A CONTROLLABLE PROLOG DATABASE SYSTEM

By
N. Minsky[1], D. Rozenshtein[2] and J. Chomicki

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

Abstract

This paper presents a model that provides a single comprehensive mechanism to control the use, operation and evolution of database systems. This model unifies several concepts generally considered to be quite distinct. In particular, it minimizes the formal distinction between the users of the database, the programs embedded in it and even the administrators and the programmers maintaining it. Furthermore, under this model, the concepts of subschema and of program module are replaced with a single concept of *frame*, which serves as the locus of power and of activity in the system. Moreover, the proposed control mechanism is closed, in the sense that the process of establishing controls is itself controllable by the same mechanism. This can be used to formalize and control managerial policies about the use and evolution of database systems.

1 Introduction

We are interested in the long-term integrity, reliability and security of database systems. One of the traditional approaches to insuring these has been through the use of various protection techniques.

First, these techniques have been used to protect the database from misuse by its users. For that, there are the standard authorization mechanisms [8, 24, 19, 13] that restrict the ability of users to operate on the database.

Secondly, most modern database systems are not merely repositories of data, but also have programs embedded in them. (These programs are typically used to implement transactions, to carry out inference, to perform internal bookkeeping, etc.) An important, even if less common, use of protection in database systems is the imposition of control on the activity of these programs. Regimes that constrain activity of programs in general are provided by many current programming languages; they are usually based on *scope rules*, sometimes enhanced by additional constructs such as Ada *packages* and by *module interconnection languages* used in a variety of software development environments [12, 5, 20, 9, 14]. Analogous control regimes have recently been introduced into database systems as well [17, 1, 18].

Unfortunately, as has been pointed out in [16], such protection regimes are inadequate in the context of *evolving* database systems. By evolution one usually understands any change in the way the system operates and/or the way it interacts with its environment. For example, any change in the set of database users or in the authorization structure imposed on them is considered to be evolutionary in nature. Likewise, any modification to the database system code, either as a result of routine maintenance, or more importantly to make the database conform to the ever-changing real world situation it is modelling, also serves as an example of database evolution.

Evolution of a database system is usually affected by its *builders*. These include all people responsible for the operation and maintenance of the database system and, in particular, include system programmers and the database administrator together with the members of his staff.

It is quite clear that in the context of evolving systems, it is not sufficient to have constraints on what each user and each module of the system can do; one must also control the activity of its builders. After all, it would be quite difficult to say anything about the long-term behavior of a database system if everyone working for the database administrator could, at will, grant users permissions to access the database, or if every programmer could arbitrarily change any piece of code in the system.

[1]Research supported in part by NSF grant MCS 82-03954.

[2]Research supported in part by NSF grant IST 84-08970 and by Henry Rutgers Research Fellowship

Note, however, that as part of their activity, builders provide other builders (or themselves, for that matter) with the power to update the data in the database and/or the power to program. Therefore, it is not sufficient to control what the constraints on the various builders of the system are; one must also control the very process of establishing and changing these constraints.[3]

This paper describes a database model that provides precisely such a *comprehensive* control regime. The regime is *closed*, in the sense that the process of establishing controls is itself controllable by the same regime. Furthermore, the regime is *unified*, in the sense that it controls the activities of the users of the database system, its own program modules, and the builders who develop and maintain it in a uniform way.

We note that the need to control the operation and evolution of systems is not unique to databases. Indeed, this work is based in part on the ideas used in DARWIN [15, 16], a programming environment that attempts to provide such control for large-scale systems in general.

We have chosen to base our model on Prolog. Our reasons for this are twofold. First, Prolog provides us with an elegant and general formalism for expressing our ideas and a convenient vehicle for testing them. Second, we believe that our approach can be used to enhance many of the current attempts to use Prolog in the context of databases.

The remainder of this paper is organized as follows. In Section 2, we briefly present a naive view of Prolog as a database language. In Section 3, we describe our model. In Sections 4 and 5, we discuss the behavior of the database system operating under this model.

2 Prolog as a Database Language: A Naive Approach

Prolog[4] is a programming language based on logic. There is a natural correspondence between Prolog programs and databases (in particular, relational ones). Consider the Prolog clause

```
p(a,b).
```
[5]

This clause (called a *ground unit* clause), which is interpreted declaratively as "p is true of a and b" or procedurally as "goal p is satisfied with respect to a and b," can also be taken to represent a tuple, while a collection of such clauses can be thought of as a relation.

Likewise, consider a clause

```
r(X,Y) :- p(X,Z), q(Z,Y).
```

This clause (called a *rule*, with `r(X,Y)` called the *head* and the rest of the clause called the *body*) is interpreted declaratively as "r is true of some individuals X and Y if there exists some individual Z such that p is true of X and Z and q is true of Z and Y." Procedurally, it is interpreted as "to satisfy goal r with respect to parameters X and Y satisfy goal p with respect to X and some Z and then satisfy goal q with respect to that Z and Y." However, this clause can also be thought of as defining a relation r as a *join* of relations p and q. Other relational operations can also be easily expressed as Prolog clauses. For example, the following clause

```
s1(X) :- p(X,Y).
```

defines relation s1 as a *projection* of relation p onto its first attribute, while the clause

```
s2(X,Y) :- p(X,Y), X > 100.
```

defines relation s2 as an appropriate *selection* of p. Thus, Prolog rules can be interpreted as defining relational *views*.

User interacts with a Prolog system by submitting goals or conjunctions of goals for evaluation. Consider, for example, the following goal

```
?- r(X,Y).
```

This goal is interpreted declaratively as a question "is r true with respect to some individuals X and Y, and if yes, then what these individuals are?" It is

---

[3]Such "meta-control" is usually viewed as a managerial concern, which does not require any formal, computer-based, support. We believe, however, that this view is inappropriate in the context of truly large, evolving database systems.

[4]In this paper, we assume the readers' basic familiarity with Prolog, as described, for example, in [6].

[5]We adhere to the now standard DEC-10 Prolog syntax, where variables are denoted by alphanumeric strings starting with an upper-case letter, while constants are either integers, or are denoted by alphanumeric strings starting with a lower-case letter. As usual, the underscore character '_' may be embedded into a constant or into a predicate name to improve readability. A single underscore character, however, denotes an anonymous variable that will unify with anything. Also note that a period must terminate every Prolog clause.

interpreted procedurally as a command "satisfy goal r with respect to some individuals X and Y, and if successful, then tell who these individuals are." However, it can also be interpreted as "retrieve relation r" and, thus, can be thought of as a *query*.[6] (If relation r is defined as above, then this clause can effectively be interpreted as "retrieve the join of relations p and q.")

Prolog provides a variety of built-in predicates (such as **assert** and **retract**) whose evaluation leaves side effects.[7] For example, evaluation of the following goal

```
?- assert(p(c,d)).
```

results in clause

```
p(c,d).
```

being added to the Prolog program, while the evaluation of the goal

```
?- retract(p(a,b)).
```

results in the deletion of the corresponding clause from the program.

We note, that **assert** and **retract** do not have to be invoked directly by the user, but can appear as subgoals in the body of some rule. For example, consider the following rule

```
go :- p(_,Y), retract(q(Y,_)), fail.
```[8]

This rule can be thought of defining a *transaction*, which, when invoked, will delete all those tuples from q where the value for the first attribute appears as a value for the second attribute in some tuple in p.

Built-in predicates **assert** and **retract** can also be used to add and delete rules, thus in effect giving the user ability to reprogram the database. For example, the evaluation of the following conjunction of goals

```
?- retract((s1(X) :- p(X,Y))),
   assert((s1(Y) :- p(X,Y))).
```[9]

will change the definition of s1 to be the projection of relation p onto its second attribute.

To summarize, Prolog provides facilities to support representation of data in the database and to perform such standard database functions as data retrieval and modification, and definition of views and transactions. Furthermore, it does this in a uniform way, which is considered an advantage. However, the above approach to using Prolog as a database language is actually quite naive. First, the resulting database system would be quite slow. Second, since Prolog provides no support for persistent data, the database would disappear at the end of the session. These and many other problems are well recognized and have been addressed in the literature. For example, compilation and optimization of Prolog queries has been studied in [10, 23, 11], while special computer architectures to speed-up the retrieval from a Prolog database have been considered in [21]. On the other hand, the problem of providing efficient storage for persistent Prolog databases has been discussed in [2, 22].

The problem to be addressed in this paper is that the naive model described above provides absolutely no facilities to control either the users of the database or the programs embedded in it. Indeed, any user can directly retrieve every fact stored in the database and can also add and delete facts at will. Although, as we have seen, Prolog provides facilities to define views and transactions, users are in no way restricted to them. Moreover, the body of Prolog code, which constitutes the semantics of the database, cannot be modularized. And finally, Prolog imposes no restrictions on who can introduce and remove rules. This allows anyone, who has any access to the system, to reprogram the database in arbitrary and unpredictable ways.

The difficulties with controlling the activity of Prolog programs stem from the fact that different pieces of Prolog code cannot be hidden from one another. Hence, such useful facilities as modules and abstract data types, to name just a few, cannot be properly supported in standard Prolog.

In the next section we present our solution to these control-related problems in Prolog databases. It

---

[6]Strictly speaking, this is true only if the user forces backtracking to obtain all different combinations of X and Y that satisfy r.

[7]For the sake of simplicity, in this paper we assume that the only way of modifying a Prolog program is through the use of these two built-in predicates. We also assume that the effects of their execution are *persistent*, i.e., kept between different Prolog invocations.

[8]Subgoal **fail** forces backtracking to try all possible ways of satisfying p.

[9]The extra pairs of parenthesis around arguments to **assert** and **retract** are required in this case by Prolog syntax.

should be pointed out that there have been attempts to introduce modularization into Prolog, by means of conventional *name visibility* constraints [4, 7]. The problem with such constraints is that because they apply to code only, they are too limited in scope and, therefore, are unsatisfactory in our context. There have been no attempts to provide a *comprehensive* control mechanism of the kind discussed in the previous section.

3 The Model of a Controllable Prolog Database System

The database system is defined as a quadruple: <**D**, **P**, **F**, **R**>, where **D** is a *database*, **P** is the set of *people* interacting with it, **F** is the set of *frames* and **R** is the set of *restrictions* on **D** and **P**.

Basically, database **D** is a set of Prolog clauses. The (syntactic) form of these clauses will be restricted by **R**, in a manner shown below.

The set **P** of people encompasses all persons that have access to our system. In particular, this set includes both the users and the builders of the system.[10]

Frames will play a central role in our model. Among other things, they will serve as subschemas and as modules. For the time being, however, it is sufficient to say that each frame in **F** is a triple <name, body, interface>, where name is a unique identifier of the frame. The nature of the body and the interface of a frame will be explained later. While the choice of frames and their names in general depends on a particular application, we always require the presence of a distinguished *origin* frame, to be named o.

Finally, **R** is the set of restrictions. We distinguish between two kinds of restrictions: on database **D** and on people **P**.

3.1 Restrictions on the Database

These restrictions are basically the following rules of correspondence between **D** and **F**.

First, every predicate in database **D** has the form p_f, where f is the name of some frame in **F**. In effect, we are changing the syntax of Prolog predicate names.[11] Furthermore, every built-in Prolog predicate q (such as `assert` and `retract`) is replaced by q_o, o being the name of the distinguished origin frame. Thus, only those predicates that have a subscript of o retain their built-in Prolog meaning in our system.[12] Finally, we note that Prolog operators (such as :-, !, etc.) are not subscripted.

Second, only the following three types of Prolog clauses are allowed.

1. *Facts.* These are ground instances of unit Prolog clauses, which have the following structure

   $p_f(\ldots)$.

2. *Local Rules.* These are Prolog rules where both the head predicate and all predicates in the body have the same subscript. Thus, local rules have the following structure

   $p_f(\ldots)$:-
   $s1_f(\ldots)$, ..., $sk_f(\ldots)$.

3. *Connections.* These are Prolog rules where the subscript of the head predicate is different from the subscript of at least one predicate in the body. The general form of connections is presented below

   $p_f(\ldots)$:-
   $s1_{f1}(\ldots)$, ..., $sk_{fk}(\ldots)$.

No other kinds of clauses are allowed in **D**.

For each clause in database **D**, we define the notion of the *home frame* in the following way. A

[10] In the context of database systems, application programs are also often considered to be users. While we do not discuss them explicitly in this paper, controlling their activity in our system is straightforward. Also, recall that builders include the database administrator and his staff, that is the people who affect the maintenance and evolution of the system.

[11] We note that the subscript notation is used for the sake of convenience only. In the actual implementation [3], frame names are appended in an unforgeable fashion to the end of predicate names.

[12] This requires a minor modification to the scanner part of Prolog interpreter.

home frame of a fact is determined by the subscript of the predicate involved. A home frame of a local rule or connection is determined by the subscript of the head predicate. We also say that a clause *resides* in its home frame. Finally, we say that the facts and local rules residing in a frame make up its *body*, while the connections residing in the frame make up its *interface*.

Note that all we do here is syntactically partition the name space of the database into single-subscript regions of Prolog clauses (these correspond to the bodies of the various frames) and provide connections between these regions by means of frame interfaces. The significance of this partitioning will be clarified later.

3.2 Restrictions on People

For a person to operate on the system, he must associate himself with a specific frame. He does so by *entering* the frame.[13] This frame is called the *host* of the person for the duration of the interaction.

While in a frame, all the person can do is to present goals (or conjunctions of goals) for evaluation. Furthermore, the person is restricted to presenting only those goals whose predicates are subscripted with the name of his host frame. Thus, while in frame f, a person can only present goals of the form

$$?\text{-}\ p_f(\ldots).$$

We note that it is not really necessary for a person to actually subscript predicates in his goals with the name of his host frame. This can (and, in fact, should) be done automatically by the system. It is essential to note, however, that it is *impossible* for any person, even the database administrator himself, to issue goals not appropriately subscripted.

3.3 Frame as a Locus of Power and Activity

As we have already seen, frames serve to partition the name space of the database. This fact has several significant implications.

First, frames can be viewed as the *loci of all activities* in the system. Indeed, the evaluation of any given goal

$$?\text{-}\ g_f(\ldots).$$

can be viewed as being done in the context of frame f, since this goal can invoke only clauses residing in f. This is true regardless of whether the goal in question is presented by some person or is part of the body of a previously invoked clause. Thus, a frame can serve as the locus of the activity of *both* people and programs.

Second, under our model, an execution can be thought of as travelling between frames. Note, however, that for goal

$$?\text{-}\ g_f(\ldots).$$

to trigger the evaluation of clauses defined in frames other than f, it is necessary for f to contain appropriate connections. Therefore, we can view a frame as providing a specific *power* to any actor operating from it with respect to the rest of the system. This power is determined by the interface of the frame, which is, of course, the set of all its connections.

To further appreciate why a frame can indeed be viewed as a *locus* of this power, consider some person who enters a frame, whose body and interface are empty. In this case, every (positive) goal posed by this person will fail. Thus, a person acting from an empty frame cannot do anything at all and, therefore, has no power whatsoever.

These aspects of frames allow us to use them in several different roles:

1. As the context for the activity of *users*, thus serving the traditional role of database *subschemas*.

2. As the context for the definition of *facts* and for the activity of *local rules*, thus serving the traditional role of *modules* in programming languages.

3. As the context for the activity of *builders* of the systems, thus allowing us to control them the same way we control users and programs.

Because our model treats users, programs and builders in a uniform way, we find it convenient to define a notion of *actor* to encompass all of them.

[13] In this paper, we do not concern ourselves with the very process of a person entering a frame. However, we take it for granted that there exists some external security scheme, based perhaps on passwords, that allows people to enter frames in a correct way.

Also note the distinguished role played by the frame names and the connections. The former are used to partition the predicates, while the latter, in effect, determine the power invested in each frame. This motivates us to define the notion of *power structure* as a pair <N, C>, where N is the set of frame names and C is the set of connections in the system.

In the next section, we consider the behavior of our system under a fixed power structure. The creation and manipulation of the power structure itself is discussed in Section 5.

4 Operating under a Fixed Power Structure

4.1 An Example of an Airline Reservation System

In this section, we discuss the behavior of a database system operating under a fixed power structure. To illustrate the discussion, we consider an airline reservation system that maintains, among other things, information about which flights are run by which airlines and about passenger reservations made by airline reservation clerks.

As an example, assume that we want our system to always satisfy the following two requirements:

1. All reservations on flights run by a particular airline can only be made by reservation clerks of that airline; and

2. Only one reservation can exist for a specific seat on a particular flight on any given date.

We model such a reservation system in the following way. For each airline we shall have a frame. This frame will be used by the reservation clerks of that airline, and it is from this frame that they will interact with the rest of our system (in order, for example, to make passenger reservations). For the sake of specificity, assume that one of these frames, to be named t, is assigned to TWA reservation clerks.

In addition, our system will have the following three frames: frame r, frame g, and the distinguished origin frame o (whose presence is, of course, mandatory).

Frame r, will be used as the encapsulation of the reservation information. In particular, it will contain a collection of facts of the following structure

```
reservation_r(customer_name,
              reservation_date,
              flight_number,
              seat_number).
```

Frame g, on the other hand, will serve as the encapsulation of various other flight-related information. In particular, let frame g contain facts of the following structure

```
run_g(flight_number, airline_company).
```

(In this particular example, each of the frames r and g contain facts over a single predicate. In general, of course, a frame may contain facts over any number of predicates.)

We now consider a sequence of examples illustrating the dependency between the body and the interface of a frame, and the resulting power provided by the frame to any person acting from it.

4.1.1 The Power to Retrieve Local Information

Consider a certain TWA reservation clerk (say, Mr. Smith) who enters frame t mentioned above. Suppose that the body of frame t contains some facts of the following structure

```
sales_t(month, number_of_tickets_sold).
```

In this case, our clerk Mr. Smith can successfully pose the following query

```
?- sales_t(Month, Number).
```

and thus, in effect, has the power to ask direct questions about facts residing in his host frame.

Suppose, next, that in addition to these facts, the body of frame t also contains the following local rule

```
L1:: big-sales_t(Month) :-
        sales_t(Month, Number),
        Number > 100.
```

In this case, our TWA reservation clerk can pose queries that involve the use of local rule L1, and thus, in effect, has the power to use *local views*.

4.1.2 The Power to Retrieve Global Information

So far, Mr. Smith was able to ask questions only about information local to his host frame. Suppose, however, that, being a reservation clerk, he also needs

to know the details of reservations, but only on flights run by TWA. This power would be available to him if the following connection is present in the database

```
C1:: reservation_t(Name, Date, Flight, Seat) :-
        run_g(Flight, twa),
        reservation_r(Name, Date, Flight, Seat).
```

Note that C1 connects frame t to both predicate run_g in frame g and predicate $reservation_r$ in frame r. When invoked, this connection will, in effect, move the computation from frame t to frames g and r.

4.1.3 The Power to Update the Host Frame

Until now, we have only dealt with the *power to retrieve*; we now turn to the *power to update*, starting with the updates to the body of the host frame. Suppose that we wish to allow Mr. Smith to add new facts into his host frame, say, in order to enable him to record monthly sales information himself. Recall that the only way one can add new clauses, of any kind, into the database is by means of predicate $assert_o$, which resides in the origin frame o. It is, therefore, necessary to have in frame t a connection which would allow the invocation of $assert_o$. Note, however, that we do not want to allow our TWA reservation clerk unrestricted use of $assert_o$; he is only to be allowed to assert facts, and only into his host frame. Precisely this power is provided to him by the following connection

```
C2:: insert_t(X) :-
        fact_o(X),
        transform_o(X, t, Y),
        assert_o(Y).
```

Connection C2 operates as follows. First, the subgoal $fact_o(X)$ is invoked. We assume that predicate $fact_o$, which resides in the origin frame o, is defined to succeed only if its argument is indeed a fact (i.e., a ground unit clause). Second, the subgoal $transform_o(X, t, Y)$ is invoked. We assume that predicate $transform_o$, also residing in the origin frame, subscripts the predicate name of the fact instantiating the variable X with the frame name t, and associates the resulting structure (which is also a fact) with variable Y. This fact, thus transformed to be local to frame t, is finally asserted by the last subgoal of the connection.

The presence of connection C2 gives our TWA reservation clerk the power to assert a fact such as "the monthly sales for December were 90 tickets" simply by presenting the following goal

```
?- insert_t(sales(december, 90)).
```

Let us now consider a situation where we want to give our clerk the power to introduce local rules (*note*, not connections) into his host frame. This will provide him with the ability to program his host frame, and to define his own private views. Such power can be provided by the following connection

```
C3:: insert_t(X) :-
        rule_o(X),
        transform_o(X, t, Y),
        assert_o(Y).
```

The only difference between C3 and C2 is that C2 checks for the argument being a fact, while C3 checks for it being a rule.

We note that connections similar to C2 and C3 to $retract_o$ would allow our clerk to delete facts and local rules from his host frame.

4.1.4 The Power to Update Other Frames

One can distinguish between two ways for an actor, operating from a given frame, to update the body of other frames: *directly* and *indirectly*. For example, suppose for a moment that our database contains the following connection

```
direct_insert_t(X) :-
        fact_o(X),
        transform_o(X, r, Y),
        assert_o(Y).
```

The above connection differs from connection C2 only in its $transform_o(...)$ subgoal, which subscripts the fact instantiating variable X with the frame name r. Its presence would enable any actor operating from frame t to introduce arbitrary new facts into frame r.[14] While such direct updates may be appropriate sometimes, in our case they could lead to violations of both of the requirements that we want our airline reservation system to satisfy. In particular, our TWA reservation clerk would be able to make reservations on flights run by other airlines. Furthermore, he would be able to make reservations for the already reserved seats.

[14] Note, however, that the above connection *will not* allow an actor operating from frame t to introduce any facts into frame t itself.

It follows, then, that the above connection should not be included in the database. And yet, a TWA reservation clerk should be able to reserve seats on TWA flights. The obvious solution is to provide the clerk with the power to invoke an appropriate transaction defined in frame r itself.

In particular, assume that frame r contains the following local rule

```
L2:: reserve_r(Name, Date, Flight, Seat) :-
       not reservation_r(_, Date,
                         Flight, Seat),
       insert_r(reservation(Name, Date,
                         Flight, Seat)).[15]
```

and the following connection

```
C4:: insert_r(X) :-
       fact_o(X),
       transform_o(X, r, Y),
       assert_o(Y).
```

(Connection C4 is identical to connection C2, except for the use of r as the subscript and as the argument to $\texttt{transform}_o$.)

Note that local rule L2 together with connection C4 provide safe and general purpose means for introducing reservation facts into the database, which is consistent with the second requirement from Section 4.1. In order for our TWA clerk to be able to make TWA reservations, his host frame t should contains the following connection

```
C5:: reserve_t(Name, Date, Flight, Seat) :-
       run_g(Flight, twa),
       reserve_r(Name, Date, Flight, Seat).
```

The presence of this connection would allow him to invoke local rule L2, and thus to insert new reservation facts. Note, however, that the presence of $\texttt{run}_g\texttt{(Flight, twa)}$ as a condition in the body of C5, would allow for the introduction of TWA reservations only, thus fulfilling the first requirement from Section 4.1.

In this section, we have only discussed facilities for the introduction of new facts. It is straightforward to provide analogous facilities for the introduction of new local rules, and for the deletion of facts and local rules.

[15]We consider `not` to be an operator, and therefore do not subscript it.

4.2 Programs, Builders and Users

One of the main advantages claimed for our approach is the unification of control over the activities of the three types of actors operating on a database: the users of the database, the programs embedded in it, and the database builders. In this section, we show how the traditional distinctions between these types of actors are reflected in our model.

We say that a frame is a *module-frame* if it *cannot* be entered by any person.[16] Module-frames cannot serve as loci of person's activity and play the role of program *modules*. In particular, such frames can be used to encapsulate data with a set of procedures defined on it, and thus serve as a vehicle for defining *data abstraction* (in the conventional sense of this term in programming languages). For example, our frame r can be viewed as the implementation of the seat reservation abstraction.

A person can be considered a *user* of a given module-frame m if he can directly or indirectly see the facts or invoke transactions residing in m. On the other hand, a person can be considered a *builder* of m, if he can enter some frame that carries the power to update the body of m. Note that, in principle, there may be *several* different users and builders of any given module-frame; likewise, the same individual may be the user and/or builder of *more than one* module-frames. This relationship between people and modules, normally established only by managerial means, can now be formalized and enforced through our concept of power structure. Also note that, in contrast to the common practice, we do not have to make a sharp distinction between users and builders. Indeed, under our model, the same individual may have the power of a builder with respect to some frames and the power of a user with respect to others. For example, consider our TWA reservation clerk, who is a builder with respect to his host frame t, but a user with respect to frames r, g and o.

Moreover, even a module-frame itself can be considered a "builder," if it contains the power necessary to program other frames (or even itself). This is useful, especially in the context of modern programming environments, where programs are often manipulated by other programs.

[16]While we do not discuss it further in this paper, our model does provide means for specifying that a given frame is indeed a module-frame.

To conclude, the distinction between programs, builders and users is, in our system, purely the matter of the power provided to an actor by the frame from which he, she or it operates. In the following section, we briefly discuss the formation of this power.

5 Managing the Power Structure

Recall, that by power structure we mean a pair <N, C>, where N is the set of frame names and C is the set of connections. Thus, managing the power structure reduces to managing frames and connections. Also recall that in Section 1 we have discussed the need for a comprehensive closed control regime. In other words, the process of establishing the controls should itself be governed by the same regime. As we show next, our model indeed provides such a regime.

By their nature, connections are just Prolog rules. Frames, on the other hand, can easily be represented by means of predicate $\mathtt{frame}_0$; any identifier f, such that the fact

$$\mathtt{frame}_0(\mathtt{f}).$$

appears in the database, is considered to be a name of existing frame. Once a power structure is represented by a set of Prolog clauses, all manipulations of it are subject to the same control regime as the rest of the system.

The implementation of the power structure is conceptually very simple. Specifically, the origin frame o, that exists when the system is first booted up, provides an abstraction of the power structure. In other words, it contains transactions for creating and destroying frames and connections. Then any frame in the system, that has access to these transactions can affect the power structure.

We note that traditionally, it is the database administrator who is responsible for setting and up maintaining the overall policies on the use and maintenance of the database. We can support this notion of the database administrator in the following way. We set up a distinguished frame a, invest it with whatever connections to o are necessary for frame a to have sufficient power with respect to the rest of the system, and allow some selected person to enter it. It would then be up to this person to establish the overall power structure of the system.

In particular, by not supplying any other frame with connections necessary to affect the power structure, the administrator in frame a, can maintain absolute control over the power structure of the entire system. Alternatively, by appropriately distributing these connections, the administrator can achieve a variety of distributed control arrangements.

In this section, we have addressed only the issue of *how* the power structure is set up and maintained. Many related questions remain to be addressed. For example,

1. What is the relationship between various managerial policies and the corresponding power structures?

2. What are the protocols for requesting, obtaining and granting power?

3. What are the mechanisms, that will guard against actors usurping the power given to them? In other words, how can an actor, who has been given some power, be obligated to return it?

These aspects of power have already been addressed in [16] and will be discussed in the context of database systems in a forthcoming paper.

6 Conclusion

In this paper, we have presented a model that provides a single comprehensive mechanism to control the use, operation and evolution of database systems. This model unifies several concepts generally considered to be quite distinct. In particular, it minimizes the formal distinction between various kinds of actors operating on a database, namely: the users, the programs and the builders. Furthermore, it replaces the concepts of subschema and of program module with a single concept of frame, which serves as the locus of power and of activity in the system. Moreover, the proposed control mechanism is closed, in the sense that the process of establishing control is itself controllable by the same mechanism.

We believe that this model has a number of advantages, some of which are presented below.

1. The unification provided by our model results in a considerable conceptual parsimony.

2. The ability to formalize and enforce a wide range of managerial policies could enhance the reliability of database systems.

3. The ability to formalize and control the

very process of database evolution should help in integrating a database system into its operating environment.

Due to the obvious space limitations, in this paper, we have only been able to present a limited view of what our model actually provides. In particular, the model has extensive facilities for creation and destruction of frames and connections. Furthermore, it provides means for assigning various properties to frames and for grouping them based on these properties; this, in turn, allows for simultaneous creation of multiple connections. Finally, the model supports the declarative notion of connections. The full description of the model will be presented in a forthcoming paper.

Finally, we note that although our database model has been formulated entirely in terms of Prolog, it does not really depend on this language. Indeed, the principles that underlie the control mechanism employed in our model have been adapted from a general purpose and language independent[17] DARWIN programming environment [15, 16].

References

[1] Albano, A., Cardelli, L. and Orsini, R.
Galileo: A Strongly Typed, Interactive Conceptual Language.
ACM TODS 10(2):230-260, June, 1985.

[2] Chomicki, J. and Grudzinski, W.
A Database Support System for Prolog.
In *Proceedings of the Logic Programming Workshop.* Aldeia das Acoteias, Portugal, June, 1983.

[3] Chomicki, J. and Minsky, N.H.
Towards a Programming Environment for Large Prolog Programs.
In *Proceedings of the 2nd International Symposium on Logic Programming*, pages 230-241. Boston, Massachusetts, July, 1985.

[4] Clark, K.L. and McCabe, F.G.
Micro-Prolog: Programming in Logic.
Prentice-Hall, 1984.

[5] Clarke, L.A., Wileden, J.C. and Wulf, A.L.
Precise Interface Control: System Structure, Language Constructs, and Support Environment.
Technical Report COINS TR 83-26, University of Massachusetts, Amherst, August, 1983.

[6] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[7] Eggert, P.R. and Schorre,D.V.
Logic Enhancement: A Method for Extending Logic Programming Languages.
In *Proceedings of the Conference on Lisp and Functional Programming*, pages 74-80. ACM, 1982.

[8] Griffiths, P.P. and Wade, B.W.
An Authorization Mechanism for a Relational Database System.
ACM TODS 1(3):242-255, September, 1976.

[9] Haberman, A.N. and Notkin, D.S.
The Gandalf Software Development Environment.
Technical Report, Carnegie-Mellon University, January, 1982.

[10] Henschen, L.J. and Naqvi, S.A.
On Compiling Queries in First-Order Databases.
Journal of the Association for Computing Machinery 31(1), January, 1984.

[11] Jarke, M., Clifford, J. and Vassiliou, Y.
An Optimizing Prolog Front-End to a Relational Query System.
In *Proceedings of the ACM SIGMOD Conference.* 1984.

[12] Lauer, H.C. and Satterthwaite E.H.
The Impact of Mesa on System Design.
In *Proceedings of the 4th International Conference on Software Engineering*, pages 174-182. IEEE, September, 1979.

[13] Minsky, N.H.
Synergistic Authorization in Database Systems.
In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 543-552. IEEE Computer Society, September, 1981.

[14] Minsky, N.H.
Locality in Software Systems.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 299-312. January, 1983.

[15] Minsky, N.H. and Borgida, A.
The Darwin Software-Evolution Environment.
In *Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments*, pages 89-95. April, 1984.

[17] We note, however, that the current prototype of this environment is implemented in Prolog.

[16] Minsky, N.H.
Controlling the Evolution of Large Scale Software Systems.
In *Proceedings of the Conference on Software Maintenance-1985*, pages 1-16. IEEE, November, 1985.
(To be published. Also published, in the Proceedings of the International Workshop on Software Engineering Environment for Programming in the Large, Cape Cod.).

[17] Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM TODS 2, September, 1977.

[18] Smith, J.M., Fox, S. and Landers, T.
Reference Manual for ADAPLEX.
Technical Report CCA-81-02, Computer Corporation of America, January, 1981.

[19] Stonebraker, M. and Wong, E.
Access Control in a Relational Database Management System by Query Modification.
In *Proceedings of the ACM Annual Conference*, pages 180-186. 1974.

[20] Strom, R. E.
Mechanism for Compile-Time Enforcement of Security.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 276-284. January, 1983.

[21] Taylor, S. et al.
Logic Programming Using Parallel Associative Operators.
In *Proceedings of the 1984 International Symposium on Logic Programming.* Atlantic City, New Jersey, February, 1984.

[22] Wise, M.J. and Powers, D.M.W.
Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words.
In *Proceedings of the 1984 International Symposium on Logic Programming.* Atlantic City, New Jersey, February, 1984.

[23] Yokota, H. et al.
An Enhanced Inference Mechanism for Generating Relational Algebra Queries.
In *Proceedings of the ACM Symposium on Principles of Database Systems.* 1984.

[24] Zloof, M.M.
Security and Integrity within the Query-by-Example Database Managment.
Technical Report IBM RC 6982, IBM, 1978.