On Harmonically Combining Active, Object-Oriented, and Deductive Databases

David W. Embley‡ (embley@cs.byu.edu)
Stephen W. Liddle† (liddle@byu.edu)
Yiu-Kai Ng‡ (ng@cs.byu.edu)

†Information Systems Group ‡Department of Computer Science
Brigham Young University
Provo, Utah 84602, U.S.A.

Abstract

Although there are a number of issues to resolve, active databases, object-oriented databases, and deductive databases can be smoothly integrated. We present the integration challenges, provide a resolution to the issues, and show a way to achieve this integration by describing the active, object-oriented, and deductive features of a model/language called Harmony. Harmony lets us see the integration issues in a different light, because it provides an alternative way of modeling an application. It thus lets us give an alternate solution for smoothly combining event-condition-action rules, object orientation, and logic, that better supports the development of advanced applications.

1 Introduction

Over the past decade, there has been growing interest in combining object-oriented languages, logic languages, and active event-condition-action processing with database systems. Progress has been made on several fronts. Several object-oriented database systems have become commercially available (e.g., [4, 25]) and efforts have been made to unify this integration [7]. Also, several recent prototypes have been made available for both deductive objected-oriented databases (e.g., [20, 22]) and active object-oriented databases (e.g., [1, 11]), and several proposals have been made for active deductive databases (e.g., [5, 10, 12, 28, 29]). Integration issues have been plentiful and there is a wide variety of solutions ranging from obviously ad hoc (e.g., early embeddings of SQL in C) to formal and sophisticated (e.g., [15]). The integration issues are highly complex, especially for combining all three types: active, object-oriented, and deductive. These issues warrant further investigation. The eventual goal is an elegant, fully integrated, formally defined active and deductive object-oriented database system.

The research community has identified several issues that inhibit the integration of active, object-oriented, and deductive databases. (1) Impedance mismatches occur between declarative database systems and imperative programming systems [30]. (2) Object identity conflicts with least-fixed-point semantics [24]. (3) Extensible, dynamic type systems interfere with query formulation and query optimization [24]. (4) Imperative components have side effects that incrementally change the state of a system, but logic components do not have side effects [27]. (5) The absence of declarative processing semantics prevents logic systems from behaving procedurally [6]. (6) Ad-hoc integration lacks a unified formal foundation [15].

Commonly accepted definitions of active, object-oriented, and deductive databases are mostly inadequate for smoothly resolving these issues and usually lead to partially, if not completely, ad-hoc solutions. These ad-hoc solutions are not only inelegant, but they also lack a firm theoretical foundation, preventing them from being robust, optimizable, and easily understood.[1] Using slightly different definitions, however, we can find a reasonable integration and achieve the theoretical elegance we seek.

In this paper we propose an alternative way of thinking about active, object-oriented, and deductive databases, and show how this alternative resolves these issues. To resolve the impedance-mismatch problems, we treat variables as being set-valued rather than scalar-valued. To resolve the object-identity and query-language issues, our alternative replaces the instance-variables view of object orientation by an object-relationship view. To resolve the side-effects and procedural-processing issues, our alternative replaces the method-behavior view of object orientation by an object-simulation view of behavior. And, to resolve the lack of procedural logic and an adequate formal foundation, we offer a formalism based on temporal, first-order predicate logic.

The solution we propose here is based on an ontological object model with both active and passive objects that obey rules and satisfy constraints. The model has a static component for representing objects and relationships, an object-behavior component for representing individual object behavior, and an object-interaction component for representing behavioral interactions among objects. The model also has both a graphical and a textual representation, which are equivalent in every respect. We originally designed the model as an ontological, object-oriented systems analysis model called OSA (Object-oriented Systems Analysis) [9], and we later extended it to be a model for all phases of the development process and called it OSM (Object-oriented Systems Model) [19]. To make the model fully executable, we added an implementation language component called

[1]The approach in [6, 14, 15] is an exception, but we take a different approach, basing our formalism on a temporal first-order predicate calculus rather than on a new non-standard logic.

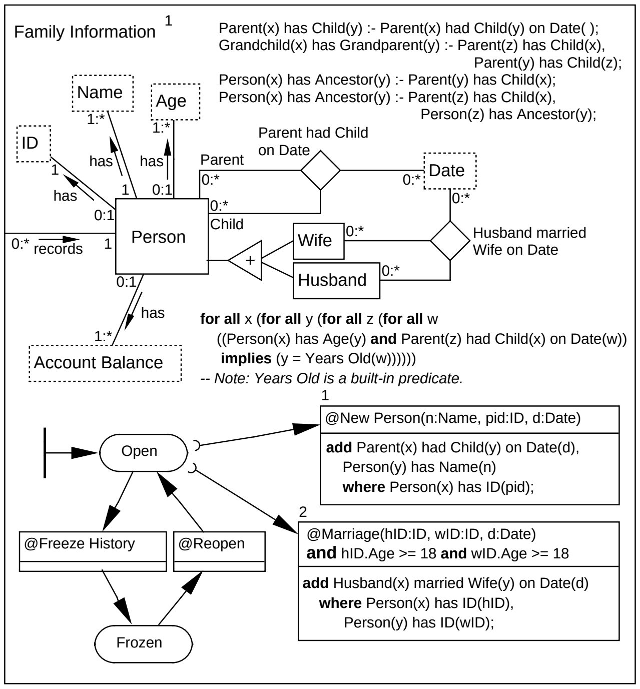
Family Information 1
Parent(x) has Child(y) :- Parent(x) had Child(y) on Date( );
Grandchild(x) has Grandparent(y) :- Parent(z) has Child(x), Parent(y) has Child(z);
Person(x) has Ancestor(y) :- Parent(y) has Child(x);
Person(x) has Ancestor(y) :- Parent(z) has Child(x), Person(z) has Ancestor(y);
Name
Age
ID
1:*
1:*
has
has
1
has
1
0:1
0:1
Parent
0:*
Parent had Child on Date
Date
0:*
0:*
0:*
Child
Person
0:* records 1
Wife
0:*
Husband married Wife on Date
+
Husband
0:*
0:1
has
1:*
Account Balance
for all x (for all y (for all z (for all w ((Person(x) has Age(y) and Parent(z) had Child(x) on Date(w)) implies (y = Years Old(w))))))
-- Note: Years Old is a built-in predicate.
1
@New Person(n:Name, pid:ID, d:Date)
add Parent(x) had Child(y) on Date(d), Person(y) has Name(n) where Person(x) has ID(pid);
Open
2
@Marriage(hID:ID, wID:ID, d:Date) and hID.Age >= 18 and wID.Age >= 18
add Husband(x) married Wife(y) on Date(d) where Person(x) has ID(hID), Person(y) has ID(wID);
@Freeze History
@Reopen
Frozen

Figure 1: Sample Harmony Application Model

Melody [16] and provided a fully textual version of the model [18]. Taken all together and presented either graphically, textually, or as a mixture of graphics and text, we call the model/language Harmony.

We proceed as follows to show how Harmony provides a point of view that smoothly resolves the integration issues that have been raised. In Section 2, we describe Harmony. We are particularly interested in its unique approach to object orientation, its active components, and its deductive features. We therefore highlight these ideas and call attention to the main points of interest for our integration discussion. In Section 3 we return to the integration issues raised above and show how Harmony resolves these issues. We conclude in Section 4 with a summary and a discussion of our current implementation status.

2 Harmony

Harmony is a *model-equivalent programming language,* which means that the constructs of its graphical model and its textual language are in a one-to-one correspondence. Both the model and language are formally defined in a model-theoretic way [8] and both are computationally complete [18]. An advantage of being model-equivalent is that the declarative aspects of the model tend to be more procedural while at the same time the procedural aspects of the language tend to be more declarative, making integration easier, more understandable, and more easily formalizable. Figure 1 shows a graphical view of a Harmony application model (Harmony also provides an equivalent purely textual representation, which we do not give here). We use this *Family Information* application model throughout this paper to illustrate our approach.

2.1 Objects and Relationships

Our definition of *object* takes an ontological perspective [26]: an object is an identifiable thing with properties. Object properties are divided into identity, structural properties, membership, and behavioral properties. As with other object-oriented systems, Harmony has *object identity* in the sense that the identity of an object is independent of the value of its other properties. An object is a member of one or more *object sets* (e.g., *Family Information*, *Person*, *Name*), and an object's membership in object sets may change dynamically. The value of structural properties for an object is represented by its relationships with other objects.

Our definition of *object set* also follows the ontological perspective. An object set is a collection of objects that share at least one property. Minimally, all objects in an object set share the property that they are members of the object set. Object sets are involved in two kinds of connections with other object sets. First, object sets may be connected by relationship sets (e.g., *Person has Name*, *Child was born to Parent on Date*). A *relationship set* is a collection of relationships that satisfy a scheme defined by the object sets to which a relationship is connected. Each *relationship* in an n-ary relationship set, $n \geq 2$, is an n-ary association among objects in the connected object sets. Relationships do not exhibit object identity directly; rather, a relationship is identified by the relationship set to which it belongs together with the identities of its related objects.

The second kind of object-set connection is *generalization/specialization.* If object set S is a specialization of object set G (or equivalently, if G is a generalization of S), then members of S are also members of G. An object set may have many generalizations and specializations. The generalization/specialization relation is transitive; that is, if S is a specialization of G, and G is a specialization of C, then S is a specialization of C (and hence C is a generalization of S). Membership in an object set implies simultaneous membership in all generalizations of the object set. A collection of object sets related by generalization/specialization is called a *generalization/specialization hierarchy.* The membership of an object is confined to a single generalization/specialization hierarchy, but an object may have the capability of becoming a member of any object set in the object's generalization/specialization hierarchy (subject to user-defined constraints). In Figure 1, *Wife* and *Husband* are mutually-exclusive specializations of *Person* (indicated by the + inside the triangle). Also, *Parent* and *Child* are *roles*, which are implicit specializations of, in this case, *Person.*

Annotations such as *0:** and *1* written near relationship-set connections are *participation constraints*, which specify the number of times an object from a connected object set must participate in the connected relationship set. For example, each *Person* object must have a single *Name* and be recorded by a single *Family Information* object. Furthermore, each *Person* may have at most one *ID*, one *Age*, and one *Account Balance*. However, a *Family Information* object may record zero or more people. Harmony also has other constraint mechanisms that we do not discuss here.

2.2 Active Objects

Behavior templates, called *state nets*, are associated with object sets, and objects act by following the rules or laws established by their state nets. An object may have multiple *threads* of control. Each thread is an independent unit of activity that is either waiting in a *state* (e.g., *Open*, *Frozen*) or actively involved in a *transition* between states. A transition has a *trigger* and an *action* (transitions are drawn as divided rectangles, with the trigger written in the upper portion, and the action written in the lower portion). A trigger is an event and/or a condition that allows a transition to execute. When a thread of control is in a transition, it executes the action specified in the transition, if any. An @ symbol denotes an event. For example, when the *Freeze History* event occurs, if a *Family Information* object is in the *Open* state, it will transition to the *Frozen* state. Arrows whose tails are disassociated from a state spawn new threads of control. If there is no arrow leaving a transition (e.g., transitions *1* and *2*), then a thread in that transition ceases to exist after executing the transition because there is no subsequent state. Thus, when the *Marriage* event occurs and the indicated husband and wife are old enough, transition *2* will fire, creating a new thread. When the **add** action associated with transition *2* is complete, the new thread will cease to exist.

A thread in transition may be associated with a nested Harmony application model. This is analogous to local variables nested within a procedure — the purpose of which is to provide temporary structures to capture the state of computation.

State nets establish two kinds of object properties: membership and behavioral properties. When an object has a thread in a state or transition pertaining to a particular object set, then the object is considered to be a member of the object set. The value of an object's behavioral properties is represented by the configuration of its threads.

Objects interact by synchronizing threads. An *interaction* involves a *send request* from one thread and a *receive request* from another. When matching send and receive requests are available, the corresponding threads synchronize, and we say that the threads interact (and hence, the objects interact). While synchronized, interacting threads may communicate (pass parameters). For example, when the *New Person send request* event occurs, a *Family Information* object in the *Open* state will spawn a new thread to handle the interaction. That new thread will synchronize with the thread that issued the *send request* event. The new thread will then receive the *Name*, *ID*, and *Date* information bundled with the event, and create appropriate objects and relationships to represent the given person in the application model.

2.3 Application Model Formalization

Harmony has a formal definition that gives precise semantics to structural and behavioral constructs, supporting both deductive and procedural paradigms within a formal, unified framework. We use a temporal, first-order predicate calculus to express this formal definition. We formalize Harmony by generating predicates and rules from a Harmony application model, and we define its semantics as the set of all *valid interpretations* (mathematical interpretations over which the generated rules hold). Following standard model theory, we specify a domain D and assign values in D to constants, functions, and predicates. We then evaluate all the rules. If they all hold, the interpretation is *valid.* We now show how to generate the predicates and rules required for this formalization. For now, we focus only on the static aspects of Harmony. We will add dynamics in Section 3.5.

2.3.1 Predicates

1. For each object set, we write a 1-place predicate that lets us assert membership of objects within object sets. The following object-set predicates come from our example:

Family Information(x)	
ID(x)	*Person*(x)
Age(x)	*Account Balance*(x)
Date(x)	*Husband*(x)
Name(x)	*Wife*(x)
Child(x)	*Grandparent*(x)
Parent(x)	*Grandchild*(x)

   Observe that *Grandchild* and *Grandparent* are derived object sets, but the others are extensional. Note that we allow white space in predicate names; furthermore, predicate names may use any combination of uppercase and lowercase letters.

2. For each n-ary relationship set, we write an n-place predicate that lets us assert the existence of relationships within the relationship set. It is significant that relationship-set names include the names of the object sets to which they are attached. Furthermore, each such n-place predicate implies n 1-place predicates corresponding to the related object sets (which may be derived or extensional). The following relationship-set predicates are derived from Figure 1:

   Family Information(x) *records Person*(y)
   Person(x) *has ID*(y)
   Person(x) *has Name*(y)
   Person(x) *has Age*(y)
   Person(x) *has Account Balance*(y)
   Parent(x) *had Child*(y) *on Date*(z)
   Husband(x) *married Wife*(y) *on Date*(z)
   Parent(x) *has Child*(y)
   Grandchild(x) *has Grandparent*(y)
   Person(x) *has Ancestor*(y)

Not only are these predicates useful in the formalization of Harmony, but these predicates are also available to the Harmony programmer for direct use in queries and (when appropriate) update operations. Harmony also has built-in predicates $=, <, \leq$, *et cetera*, and meta-predicates such as *Object Set*, *Object is running Thread*, and so forth. Meta-predicates provide powerful reflective capabilities to Harmony, so that, for example, a Harmony object could issue a query to find the object sets of which it is a member, or all threads associated with some object in the system. However, only object-set and relationship-set predicates are necessary for the formalization.

We have chosen an infix notation for predicate arguments because it matches other uses of names in Harmony better than the usual prefix notation that is conventional in deductive systems. The infix form explicitly sets off relationship-set connections, which are important because we associate constraints and roles with connections. Thus, the infix form gives our overall language a more consistent syntax. However, we observe that this decision is purely syntactical, and it is equivalent to the customary prefix notation. Thus, if we wanted we could write the infix literal *Husband(H) married Wife(W) on Date(D)* in the conventional prefix form *Husband_married_Wife_on_Date(H,W,D)*, or as would be more typical, *married(H,W,D)*. Notice that our convention connotes more of the intended meaning of a relationship set in its name, and makes it easier to see at a glance which place in the predicate encodes which information.

2.3.2 Rules

Application-model rules are closed formulas derived from model assumptions and constraints. The full set of rules is extensive [8], so we only illustrate a few rules here.

1. For each relationship set, we generate rules to ensure referential integrity. For example, to ensure that *Family Information records Person* exhibits referential integrity, we write:

$$\forall x \forall y(\mathit{Family\ Information}(x)\ \mathit{records}\ \mathit{Person}(y) \Rightarrow \\ \mathit{Family\ Information}(x) \wedge \mathit{Person}(y))$$

2. We generate a rule for each generalization/specialization to ensure appropriate set inclusion and to guarantee constraints. For example, we must guarantee that *Wife* and *Husband* are mutually-exclusive subsets of *Person*:

$$\forall x((\mathit{Wife}(x) \vee \mathit{Husband}(x) \Rightarrow \mathit{Person}(x)) \wedge \\ (\mathit{Wife}(x) \Rightarrow \not\mathit{Husband}(x)) \wedge \\ (\mathit{Husband}(x) \Rightarrow \not\mathit{Wife}(x)))$$

3. For each Harmony constraint, we must generate a rule to ensure that the constraint is satisfied. There are several kinds of constraints in Harmony, including object-set cardinality, participation, co-occurrence, generalization/specialization, and general constraints.[2] As an example, consider the participation constraint *1:** on *Name* in the *Person has Name* relationship set. This constraint implies the following rule:

$$\forall x(\mathit{Name}(x) \Rightarrow \exists y(\mathit{Person}(y)\ \mathit{has}\ \mathit{Name}(x)))$$

There are other rules dealing with aggregation hierarchies, high-level components, and so forth, that we do not illustrate here. When all the necessary predicates and rules have been written, we can test the validity of an interpretation of the corresponding application model by evaluating the rules to see if they all hold.

2.4 Variables

In procedural programming languages, variables are usually scalar, holding a single value of a particular type, whereas database queries yield sets. In Harmony, we resolve this semantic mismatch by replacing the concept of *variable* as used in programming languages with the concept of *object set*. An object set is a collection of objects, and thus can be considered a container, much like a variable. Similarly, relationship sets are containers holding sets of relationships. So a variable in Harmony is either an object set or a relationship set.

There are important differences between our approach and traditional types/variables. The most obvious is that Harmony variables are set-valued, whereas traditional variables are scalar. Traditional operators such as comparison and assignment are typically defined for scalars, not sets. Moreover, traditional operators and variables are convenient for expressing certain computations. To allow these scalar operators to apply to sets, we have redefined them to operate in a generalized context. To provide the convenience of scalar semantics, Harmony automatically treats a scalar as a singleton set and vice versa when appropriate. Thus, $x + y$ where x and y are singleton sets gives the expected addition semantics.

Another difference is that traditional variables are usually typed, but Harmony has no types. Instead, object sets can be related by generalization/specialization. We "declare a variable" in Harmony not by expressing its type, but by placing it within a generalization/specialization hierarchy. Thus, we could introduce a new variable that holds integers by making it a specialization of the *Integer* object set:

$$x\ \textbf{isa}\ \mathit{Integer};$$

With some "syntactic sugar" this becomes:

$$x\ \textbf{:}\ \mathit{Integer};$$

Since x is a specialization of *Integer* (a predefined Harmony object set whose members are predefined and cannot be changed at runtime), objects in x are also members of *Integer*. By restricting the objects that are members of *Integer*, we have also restricted the potential membership of x. Thus, we have a form of "typing" that is implied from constraints on a generalization/specialization hierarchy. We can also place constraints on object-set cardinality, and so if we wish to ensure that x is a singleton set (i.e., corresponds to a scalar), we can write the constraint:

$$x\ [1]\ \textbf{:}\ \mathit{Integer};$$

Note that in order for this constraint to be satisfied, either x must be initialized at the time of its creation, or x must exist within the scope of a transaction that defers constraint checking until the transaction is committed. Harmony supports transactions as a fundamental language and model element, but we do not explain the transaction mechanism in this paper (see [16]).

2.5 Logic Rules

Harmony supports declarative programming in the forms of logic programming (logic rules and deductive queries) and constraint specification. We have already seen several Harmony constraints, and have explored the relationship between predicates and Harmony object sets and relationship sets. We now define Harmony's logic component.

Harmony's logic component defines facts and logic rules, which are built from constants, variables, and predicates. A *constant* is any object identifier, such as an OID, a number, or a string (e.g., *-123.45*, or *"Sam"*). Syntactically, a *predicate*, as has been explained, is any valid object-set or relationship-set name, and a *variable* is any valid object-set name (within a logic rule, we can always distinguish predicates from variables according to context). A *term* is either a constant or a variable, and an atom is an n-ary predicate decorated with n terms, $n \geq 1$ (for example, *Person(x) has Name("Sally")* is a binary atom with a variable in the first argument place and a constant in the second). A *literal* is an atom or its negation (e.g., **not** *Person(x)*). A distinct

[2] *General constraints* allow Harmony programmers to directly express application-model rules as closed, well-formed formulas. Thus, a programmer can express arbitrary constraints in Harmony using predicate calculus. In our example, there is one general constraint (it begins with the phrase **for all**).

anonymous variable is assumed for any place left blank in an atom.

Logic rules are clauses of the form *head :- body* where *head* is a single n-ary atom, $n \geq 1$, and *body* is a conjunction of one or more literals, separated by commas.[3] Rules must be *range-restricted* [3], meaning that each place in a rule head has a variable, and each such variable appears in the associated rule body. Variables, which are actually temporary object sets, may be declared to be specializations of other object sets, potentially restricting the objects that a variable can hold. The following are valid rule heads:

rule1(x)
rule2(x: Integer)
Person(p) has Ancestor(a: Person)

The syntax *x: Integer* declares x to be a variable that is a specialization of the set *Integer*. Specifying a generalization g for variable x in the head of a logic rule is equivalent to conjoining the literal *g(x)* to the corresponding body. Thus, *rule2(x: Integer)* as a head is equivalent to *rule2(x)* if *Integer(x)* is found in the corresponding body. Variables have a scope that is local to each individual logic rule; thus, in our examples above, the x associated with *rule1* is distinct from the x in *rule2*. Figure 1 contains four examples of valid Harmony logic rules for deriving *Parent has Child*, *Grandchild has Grandparent*, and *Person has Ancestor*.

To ensure safety in Harmony logic rules, we require that all variables in a rule be *limited* (i.e., finite [23]). Furthermore, we restrict the use of built-in predicates and negative literals. Negative literals may appear in a rule body only if the resulting program is *stratified* [23]. Thus, variables appearing in a negative literal in a rule body must also appear in a positive literal in the same body. Moreover, each variable occurring as an argument of a built-in predicate in a body must also occur in an ordinary (non-built-in) predicate of the same body, or it must be bound (directly or transitively) by equality to an object constant or a variable that occurs in an ordinary predicate of the same body.

Recall that the predicate *Grandchild has Grandparent* from Figure 1 also implies predicates for derived object sets *Grandchild* and *Grandparent*. Now consider the predicate *Person has Child*; this predicate also implies *Person* and *Child*, which exist in the application model as extensional object sets. Since we have a strict separation between extensional and intensional predicates, we limit intensional relationship-set predicates that incorporate extensional object sets, by implicitly typing the extensional object-set places. Thus, the rule head *Parent(x) has Child(y)* is implicitly transformed to *Parent(x: Parent) has Child(y: Child)*, since *Parent* and *Child* are both extensional. This implicit typing prevents Harmony logic rules from deriving new facts for extensional object sets.

We also require that rules be partially rectified. We say that the rules for a predicate p are *partially rectified* if all their heads are identical up to variable renaming, the rules are range-restricted (i.e., for each rule, every variable in the head also appears in the body), and no constants appear in a rule head.[4]

[3]In traditional deductive-database systems, a *fact* is a rule that consists of a single positive literal for the head, and no body. However, in Harmony facts are represented by the membership of objects and relationships in object sets and relationship sets respectively. Because other representations already exist for writing facts, Harmony does not allow facts to be written as one-literal rules.

[4]The motivation for prohibiting constants in rule heads is technical, and relates to the strict separation between intensional and extensional information in Harmony, and our desire for consistent syntax.

As in Figure 1, rules deriving object sets and relationship sets may be written at the global level. Also, rules may be locally scoped by embedding them within transition actions, in which case the rules are not accessible to threads outside the transition. In such a case, logic rules may be freely interspersed with procedural statements without impacting the evaluation strategy. When an action body is parsed, Harmony first gathers together all logic rules, and then evaluates procedural statements in turn. The order of Harmony rules is insignificant.

2.6 Updates

Harmony supports updates by its **add** and **remove** statements and by a generalized assignment statement that can be rewritten as a sequence of **add** and **remove** statements. In addition, Harmony supports transaction processing, which ensures that individual update statements and explicitly specified update sequences execute atomically and ensure that the database is consistent. An **add** statement has the form

add *add-literal*$_1$, ..., *add-literal*$_n$ [**where** *body*];

where each *add-literal*$_i$, $1 \leq i \leq n$, is an object-set or relationship-set predicate with a constant or variable in each argument place, and *body* is as defined for logic rules. The interpretation of this syntax is as follows. A variable in an **add** statement is *free* if 1) it appears in an *add-literal* but does not appear in the **add** statement's body, and 2) if the variable is an object set defined outside the **add** statement, it is empty. For each free variable in an *add-literal*, the system generates an object and binds it to the variable. The body is evaluated to infer the values of bound variables in the *add-literals*. If there are m bound variables $x_1, \ldots, x_m$ in the *add-literals*, this evaluation is analogous to evaluating the rule $p(x_1, \ldots, x_m)$:- *body*. For each *add-literal*$_i$, $1 \leq i \leq n$, let $x_{i_1}, \ldots, x_{i_k}$ be the variables that appear in *add-literal*$_i$, let $y_{i_1}, \ldots, y_{i_l}$ be any free variables or constants appearing in *add-literal*$_i$, and let C be the object set or relationship set corresponding to *add-literal*$_i$. Then for each tuple t in the inferred relation p, the system inserts into C the object or relationship formed by restricting t to $x_{i_1}, \ldots, x_{i_k}$ and joining it to $y_{i_1}, \ldots, y_{i_l}$. All insertions for a single **add** statement execute as a transaction (i.e., the statement executes atomically). Naturally, predicates described by *add-literals* must be extensional predicates.

Figure 1 contains two **add** statements. For example, in transition *1*, since a **where** clause is present, the system first locates those objects in *Person* related to the objects in *pid*. If such a person is not found, the *add* statement completes without modifying the database, because the *where* clause infers an empty set. However, assuming one or more people with the given IDs are found, the system binds a new object to y and adds that object to *Person* and *Child*. Also, the objects in d are added to *Date*, and the objects in n are added to *Name*. Next, appropriate relationships in *Person has Name* are established between each object in y and each object in n. Similarly, relationships in *Parent had Child on Date* are established between the objects in x, y, and d.

Harmony's **remove** statement deletes facts from the system in an analogous way. A **remove** statement has the following form:

remove *remove-literal*$_1$, ..., *remove-literal*$_n$
[**where** *body*];

where each *remove-literal*$_i$, $1 \leq i \leq n$, is an extensional predicate, and the optional **where** clause is similar to the **where** clause of the **add** statement. For each ground substitution of the *remove-literals* and *body* such that the resulting ground literals are facts in the database, we remove the corresponding objects and relationships from the object sets and relationship sets underlying the *remove-literals*. When we remove an object from an object set, we also remove the object from all specializations, and we remove any relationships in which the object participates. When we add a relationship, objects are automatically added to the appropriate object sets, but when remove a relationship, objects are left in their respective object sets. Moreover, removing an object from a specialization object set does not force its removal from any generalization object sets.

There is one other update statement in Harmony: assignment. Harmony's assignment statement takes the form

container := query;

and is a convenient shorthand for combining common sequences of **add/remove** statements in a single transaction. Because Harmony uses object sets and relationship sets in place of the usual scalar variables, Harmony's assignment statement is necessarily generalized to deal with non-scalar semantics.

Consider the following example, which extends the application model of Figure 1 to store the ancestors of a given person in a new object set:

```
Jim, Jims Ancestors: Person;
Jim := Person() has Name("Jim").Person;
Jims Ancestors := Person(Jim) has Ancestor().Ancestor;
```

The first line is a statement that declares two new object sets, *Jim* and *Jims Ancestors*, each of which is a specialization of *Person*. A newly-declared object set is initially empty. The next statement is an imperative assignment of the expression on the right-hand side into the object set *Jim*. The right-hand-side expression is a *dot query*,[5] that returns people named "Jim". The final statement is also an imperative assignment using a dot query. Notice here that *Person*(*Jim*) *has Ancestor*() is a deductive query defined by logic rules. However, the syntax is similar to that used in the previous query, which used an extensional relationship set. Thus, it is not apparent from the query syntax whether a predicate is intensional or extensional. For a full discussion of Harmony assignment, see [16].

3 Discussion

3.1 Impedance Mismatches

As described in Section 2.4, there is a fundamental mismatch between set-oriented processing in database and logic programming systems and scalar-oriented processing in imperative programming languages. The result of a query, whether it is a relational query or a deductive query, is a set. Fundamental computations and assignment operations in procedural languages, however, apply to scalars, and, even when they apply to larger structures, they usually only apply as a shorthand for an iteration over a sequence of scalar operations.

[5]The full definition of assignment and dot queries in Harmony is quite extensive, so we omit most of the details here. A dot query automatically joins the left-hand side expression with the right-hand-side expression, traversing the object-relationship graph if necessary, and then projects on the right-hand-side expression.

Initial ad-hoc solutions to this impedance-mismatch problem provided a way to escape from procedural semantics to process a database query. In Harmony, there is no need to escape from procedural semantics, and results of queries can be stored directly into Harmony variables. Harmony does not force programmers to turn sets into sequences of scalars. Instead, *Harmony variables are fundamentally set variables*. Thus, they can directly receive the results of any query in the sense that they can appear on the left-hand side of an assignment statement whose right-hand side is a query. At the same time, Harmony does not preclude iteration through a set. A **for each** loop statement provides a general iterator for all sets. It returns elements one at a time from any set (predefined by the application model or defined dynamically by a query).

As an example, we can write Harmony code to increase the balance of John Smith's grandchildren's bank accounts by $100 each:

```
John Smith Grandchild(x) :-
            Grandchild(x) has Grandparent(y),
            Person(y) has Name("John Smith");
Richer Child : Person;
Richer Child := John Smith Grandchild;
for each Richer Child(x) do
    if x.Account Balance <> {} then
        Person(x).Account Balance := Person(x).Account
                                                          Balance + 100;
    end;
end;
```

In this program segment, this first statement defines a logic rule to extract the grandchildren of John Smith. The second statement defines an object set (or variable), *Richer Child*, that is a specialization of *Person*. The third statement gathers the set of grandchildren, defined by the *John Smith Grandchild* derived object set, and assigns that set to *Richer Child*. The fourth statement iterates over the *Richer Child* set. For each object in this set, x is bound to that object and the enclosed **if** statement is executed. If the object bound to x has an account balance (i.e., the set of related account balances is not empty), it is replaced by a new balance.

We emphasize that the distinction between Harmony assignment and traditional procedural languages is our set semantics. The fundamental advantage provided by Harmony is that its variables represent sets, not scalars. This reduces the initial mismatch, and then, since Harmony allows a query in any context where a set is expected, the impedance mismatch disappears.

On the other hand, scalar operations could be problematic in Harmony; however, they are not problematic because, with obvious adjustments, we can always use singleton object sets in any scalar context. With "syntactic sugar," we can even make them look like scalars. In the code fragment above, *Person(x).Account Balance* is a singleton object set used in a scalar context, and *100* is a singleton object set that looks like a scalar.

3.2 Object Identity

Object identity is the property of objects that divorces identity from physical attributes such as content, location, and addressability [13]. Thus, in systems with object identity, two objects are the same only if they have the same identity, regardless of any other object properties such as the value of an object's attributes. Two objects could have the same attribute values, but may still be distinct because of object identity.

Ullman pointed to a potential problem related to object identity in the following example from [24]:

r_1: *path*(X,Y) :- *arc*(X,Y).
r_2: *path*(X,Y) :- *path*(X,Z) & *arc*(Z,Y).

The claim is that if we treat object identity seriously, then *path* facts proved in different ways should have distinct identities. The consequence is that if there are any cycles in the *path* relation, the evaluation of these rules will not converge to a least fixed point, because unique objects are generated for each cyclic path (of which there may be infinitely many). Thus, there is a potential safety problem because the semantics of logic within an object-oriented context are not well defined. The fundamental problem here is that if we try to impart object identity to derived tuples, we interfere with the least-fixed-point semantics used to define logic evaluation.

To guarantee that logic evaluation eventually reaches a least fixed point, we need to be able to determine when two facts are the same. In Harmony a fact is a statement that an object or relationship is a member of a particular object set or relationship set. Since object sets and relationship sets are represented by corresponding predicates, *facts in Harmony are just like facts in traditional deductive database systems*: they are represented by ground literals, where the predicate is the name of an object or relationship set, and the arguments are object identifiers (constants in an object system).

Thus, as with other deductive database systems, in Harmony, two facts are the same if their respective predicates and arguments are the same. The only change is that in our system, the set of constants is the set of all object identifiers, rather than just, say, integer and string literals. Furthermore, our lexical object identifiers are like traditional deductive database constants (e.g., strings and numbers), and our nonlexical object identifiers are like traditional object identifiers. Thus, the set of all Harmony object identifiers encompasses traditional deductive-database and additional object-oriented requirements.

The key to our approach is that we incorporate the relation concept directly into our object model in the form of relationship sets instead of the traditional object-oriented approach that uses attributes. Thus, relationships do not exhibit object identity. Rather, two relationships are the same if they are members of the same relationship set and their corresponding related objects are respectively identical. Thus, when we infer a new relationship, rather than associating it with a unique identity and using that identity to determine sameness, we have the traditional deductive database semantics that compares the related objects to determine sameness. Hence, as with other deductive database systems, we can always tell whether a relationship represents a new fact. Finally, because we have traditional deductive database semantics, we can rely on the standard, well-developed approaches to evaluation, safety, optimization, and other important deductive database issues.

The question with respect to object identity is not so much whether Harmony has traditional deductive database semantics — it clearly does — but whether Harmony really has object identity. Our answer is that our system exhibits the fundamental concepts of object identity as presented in traditional object-oriented discussions [2, 13]. In Harmony an object has an existence independent of its value. The user cannot modify an object identifier and, indeed, cannot even create or destroy an object identifier except indirectly by requesting that the system introduce a new object or remove one. Furthermore, for situations that require relationships to be treated like objects, Harmony allows for a relational object set to provide a bijection between relationships in a relationship set and objects in the corresponding relational object set. A relational object set is an object set in every way, and may be used as a surrogate for the corresponding relationship set.

3.3 Query Formulation and Optimization

Another barrier to integration is the difficulty of mixing ad-hoc queries with dynamic typing. The reason is that an ad-hoc query might create a new type (e.g., as a result of joining tuples in a new way), but unless query operators are predefined, ad-hoc queries on new types are not possible. In relational and deductive database systems the structure is uniform, and this regularity permits the predefinition of query operators for both extensional and intensional structures. On the other hand, providing default query operators in object-oriented systems is difficult because of the rich variety of object structures, and the strong encapsulation principle that prevents external clients from directly accessing and modifying an object's private structure.

Because we reject the standard abstract-data-type model and its form of encapsulation, objects in our system are always atomic and relationships are always constructed from object associations. Thus, *Harmony constructs have a uniform structure*. The consequence is that we are able to predefine uniform default query operators for each object set and relationship set, and thus support ad-hoc queries. The extent of each object set and relationship set is accessible through its predicate, and thus, we can use these predicates to form queries in the usual way. For example, given our *Family Information* application model, we can formulate an ad-hoc query to find the names of siblings for all persons named Matt:

> *Person*(x) *has Sibling*(y) :- *Person*(x) *has Parent*(z),
Person(y) *has Parent*(z),
$x <> y$;
> *Matts Siblings*(x) :- *Person*(y) *has Name*("Matt"),
Person(y) *has Sibling*(z),
Person(z) *has Name*(x);

We can then submit this to an ad-hoc query processor as follows.

> ?-*Matts Siblings*;

Here we are entering rules and a query from the command line, as indicated by the ">" prompt.

Moreover, we provide these operators without violating the principle of encapsulation. We do so by fundamentally reexamining encapsulation and reconstituting it as object integrity, implementation independence, and information grouping and visibility [17]. *Object integrity* is the property that an object's state cannot be modified by an external entity — state changes must be completely controlled by an object. Though state changes could be triggered by events occurring elsewhere in the system, an object must effectuate all its state changes itself. *Implementation independence* is the property that an object's logical specification is completely independent of physical implementation and optimization issues. A Harmony program is abstract and does not assume a particular physical platform. Such issues may be dealt with at a lower layer of the Harmony system, but they are hidden to the Harmony program itself. *Visibility* is the property that controls how components can be made known to each other. For example, some objects may not know about all the other objects or services available in a system. Such objects have reduced visibility. By making these three encapsulation properties independent, we are able to provide uniform query operators for all object-relationship structures.

Our approach to ad-hoc queries and encapsulation also resolves the question of whether object-oriented systems are sufficiently declarative to support adequate query optimization. Traditional query optimization techniques can be used directly to optimize deductive queries in Harmony.

3.4 Side Effects

Wegner identifies software components as *reactive* units, capable of responding to stimuli and creating visible side-effects, and logic components as *retractive*, since logic computation must be revocable [27]. That is, since a logic computation must be able to back up and try a different resolution path, any side effects of the computation must be reversible and invisible. Thus, a retractive component cannot be reactive, which implies that the object-oriented and logic paradigms are fundamentally incompatible component-based models of computation.

We view objects as the primary components in our system, and object interactions as fundamental reactive units. It is the threads associated with an object that do the processing in a Harmony program, and transitions are the units of execution within a thread. When a thread fires a transition, it executes code specified in the body of the transition. The firing of a transition proceeds imperatively, but each statement may incorporate deductive queries as needed. Thus, at programmer-defined points in a thread's lifetime, the *computational model may switch back and forth from imperative evaluation to logic evaluation* in order to satisfy some query (e.g., in support of **add/remove** statements). Furthermore, the degree to which a transition uses the logic or imperative model of computation depends upon the programmer. A transition may be coded almost entirely as a logic program, almost entirely as an imperative program, or somewhere in between, in whatever fashion is most appropriate for the problem at hand.

A multiparadigm active-object/logic environment is indeed achievable. Logic rules define relationship sets that are accessed in exactly the same way as extensional relationship sets. Logic evaluation is still free of side effects, even though deductive queries can be used in the context of an imperative update operation that has side effects.

3.5 Temporal Formalization

As introduced in Section 2.3, Harmony has a formal definition that gives precise semantics to structural and behavioral constructs, supporting both deductive and procedural paradigms within a formal, unified framework. We now add the temporal aspect of Harmony's formalization that is necessary to capture dynamic semantics.

We introduce temporality by adding zero, one, or two time places to every predicate. We add two time places for predicates and rules about objects and relationships. The two time places designate the beginning and ending time of the existence of an object or relationship. We add one time place for events, indicating the time at which an event occurred, and we add no times places for any time-invariant predicates and rules. To syntactically distinguish the time places from any other places in a predicate, we add them in a pair of parentheses at the end of the predicate. Thus, for example, instead of $Person(x)$, we have $Person(x)(t_1, t_2)$, where t_1 and t_2 indicate the time interval over which x is a member of *Person*.

Participation constraints are time invariant, and thus the rule we introduced earlier, $\forall x(Name(x) \Rightarrow \exists y(Person(y)$ $has\ \ Name(x)))$, remains unchanged. However, consider referential-integrity constraints implied by our application model. Before adding temporality, a referential-integrity rule might look like this:

$$\forall x(Wife(x) \Rightarrow Person(x))$$

Expressed temporally, this becomes:

$$\forall x \forall t_1 \forall t_2 (Wife(x)(t_1, t_2) \Rightarrow$$
$$(\exists t_3 \exists t_4 (Person(x)(t_3, t_4)\ \wedge\ t_3 \le t_1\ \wedge\ t_2 \le t_4)))$$

This rule asserts that for the time an object exists as a member of *Wife*, it must also be a member of *Person*.

Consider the sample Harmony application model in Figure 1 again. There is an object set and thus an object-set predicate:

$$\textit{Family Information}(x)(t_1, t_2)$$

This predicate asserts not only that x exists as an object, but that x exists as an object between times t_1 and t_2, inclusively. Similarly to temporal object-set predicates, there can be temporal relationship-set predicates in an application model assert that a relationship holds for a time interval.

With temporal predicates, we can also generate predicates and rules for state nets and interactions. This is the interesting addition that makes it possible to formally define the behavior of objects over time. We now explain how to generate the most important predicates and rules for our application model in Figure 1.

3.5.1 Predicates

1. For each state, we write a predicate that lets us assert that an object is in the state for a time interval. One of the state predicates is *Family Information*(x) *in state Frozen*(t_1, t_2).

2. For each transition, we write several predicates that let us trace the phases (inactive, enabled, committed, executing, finishing) involved in firing a transition. For transition *1* we write

   Family Information(x) *transition 1 inactive*(t_1, t_2)
   Family Information(x) *transition 1 enabled*(t_1, t_2)
   Family Information(x) *transition 1 committed*(t_1, t_2)
   Family Information(x) *transition 1 executing*(t_1, t_2)
   Family Information(x) *transition 1 finishing*(t_1, t_2)

3. For each trigger of each transition, we write a predicate that lets us determine when the trigger is true. For the trigger in transition *1* we write *Family Information transition 1 true*(t_1, t_2). Since the trigger in transition *1* is an event, $t_1 = t_2$ must hold here and we can simplify this predicate. In general, however, we need a duration specifying when a trigger holds.

4. For each interaction, we write a predicate that lets us assert the time an interaction occurs. For the *Freeze History* interaction, we write *Freeze History*(t_1).

3.5.2 Rules

1. As a type of referential integrity, we must guarantee that any object acting according to the state net for an object set is a member of the object set when it engages in the behavior. We thus, for example, write

   $\forall x \forall t_1 \forall t_2(\textit{Family Information}(x)\ \textit{in state Frozen}(t_1, t_2)$
   $\Rightarrow \textit{Family Information}(x)(t_1, t_2)).$

   We must write similar rules for objects engaged in transitions.

2. There are five transition phases: *inactive, enabled, committed, executing,* and *finishing.* Every object in an object set must be in exactly one phase of each transition in the corresponding state net. This takes several rules. To ensure that an object is in one of the phases for transition *1*, for example, we write

   $\forall x \forall t_1 \forall t_2(\textit{Family Information(x)}(t_1, t_2) \Rightarrow$
   $\forall t_3((t_1 \leq t_3 \wedge t_3 \leq t_2) \Rightarrow$
   $\textit{Family Information}(x)\ \textit{transition 1 inactive}(t_3, t_3)$
   $\vee\ \textit{Family Information}(x)\ \textit{transition 1 enabled}(t_3, t_3)$
   $\vee\ \textit{Family Information}(x)\ \textit{transition 1 committed}(t_3, t_3)$
   $\vee\ \textit{Family Information}(x)\ \textit{transition 1 executing}(t_3, t_3)$
   $\vee\ \textit{Family Information}(x)\ \textit{transition 1 finishing}(t_3, t_3))$

   Capturing the idea of when and how a transition fires is central to defining state-net semantics. We capture this idea by requiring a rule which ensures that if a transition is enabled for an object and the transition's trigger is true, then there exists a future time at which the transition is committed for the object. We express this rule for transition *1* in Figure 1 as follows.

   $\forall t_1(\exists x(\textit{Family Information}(x)\ \textit{transition 1 enabled}(t_1, t_1)$
   $\wedge \textit{Family Information transition 1 true}(t_1, t_1)$
   $\wedge \textit{Freeze History}\ (t_1))$
   $\Rightarrow (\exists y \exists t_2(t_2 > t_1$
   $\wedge \textit{Family Information}(y)\ \textit{transition 1 enabled}(t_1, t_1)$
   $\wedge \textit{Family Information}(y)\ \textit{transition 1 committed}(t_2, t_2))))$

   Since there is only one family information object in our example (because the object-set cardinality constraint is *1*), and since *Freeze History*$(t_1) \Rightarrow$ *Family Information transition 1 true*(t_1, t_1), we can simplify this rule for this state net to

   $\forall t_1(\exists x\ (\textit{Family Information}(x)\ \textit{transition 1 enabled}(t_1, t_1)$
   $\wedge\ \textit{Freeze History}(t_1))$
   $\Rightarrow (\exists t_2(t_2 > t_1$
   $\wedge\ \textit{Family Information}(x)\ \textit{transition 1 committed}(t_2, t_2))))$

   In general, however, the more complex rule is needed to account for object sets with multiple active objects and triggers that are not just events.

Our predicates and rules give us a way to reason about the behavior of state nets. As indicated by the discussion in the last paragraph, we can be sure that a transition will fire if an object is in a prior state of a transition and the trigger for the transition holds. With a complete set of rules, we would also be able to reason about the opposite possibility. If no object is in a prior state of a transition or the trigger is not true, then the transition does not fire.

Finally, for the temporal case, *valid interpretation* is similar to the static case. Proceeding as before, we specify a domain D and assign values in D to constants, functions, and predicates. However, for a temporal application model we must, of course, also include time in D. We then evaluate all the rules; if they all hold, the interpretation is *valid.*

4 Conclusion

Harmony is an advanced model and language that provides a novel solution to the problem of integrating active, object-oriented, and deductive databases. Much of this contribution arises from the ontological perspective with which Harmony was developed from its inception. Harmony resolves impedance mismatches by using set values as its fundamental variable structure; this approach reduces to scalar semantics when appropriate. In Harmony, object identity does not conflict with least-fixed-point semantics because of our ontological view of object/relationship structure. Our approach to encapsulation provides the uniformity needed to support ad-hoc queries in the traditional sense. Since Harmony supports traditional logic rules, all the usual deductive optimization techniques available to standard deductive systems can be applied equally well in Harmony. Moreover, these rules can be freely mixed with procedural queries in Harmony programs. When side-effect-free deductive reasoning is desired, it is available; when side-effect-inducing imperative processing is desired, it is also available. Furthermore, since Harmony's formal foundation simply expresses *what* system states are valid over time and not *how* those states are reached, both declarative and imperative processing paradigms are naturally supported within a unified, formal framework.

Our implementation of Harmony has been under way for some time now. We have created a diagramming tool, called Composer, that forms the basis for our Harmony programming environment. With Composer, a user can create Harmony application models for systems analysis, specification, design, and implementation. The Composer also provides a platform for integrating other tools to assist in the development of Harmony-based systems. For example, we have a rapid-prototyping component, a graphical query language, and a database design assistant. Other related projects are also planned or under way (see [21]). We have implemented a subset of the Harmony language, and we are continuing to enlarge the implementation. Currently, we can create, store, and parse Harmony programs. We have implemented the core interpreter of our first version of the language, Melody, and we are now working on a Harmony version of this interpreter.

References

[1] R. Ariein, J. Gava, N. Gehani, and D. Lieuwen, *Ode 4.1 Ode<EOS> User Manual,* AT&T Bell Laboratories, Murry Hill, New Jersey, 1996.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. *The Object-Oriented Database System Manifesto.* Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pp. 223-240, Kyoto, Japan, December 1989. Reprinted in *Building an Object-Oriented Database System: The Story of O2*, F. Bancilhon, C. Delobel, and P. Kanellakis, (eds.), Morgan Kaufmann, San Mateo, California, 1992.

[3] F. Bancilhon and R. Ramakrishnan, "Performance Evaluation of Data Intensive Logic Programs," *Foundations of Deductive Databases and Logic Programming,* J. Minker, (ed.), Morgan Kaufmann, Los Altos, California, 1988.

[4] F. Bancilhon, C. Delobel, and P. Kanellakis (eds.), *Building an Object-Oriented Database System: The Story of O2*, Morgan Kaufman Publishers, Inc., San Mateo, California, 1992.

[5] P. Bayer and W. Jonker, "A Framework for Supporting Triggers in Deductive Databases," *Proceedings of the 1st International Workshop on Rules in Database Systems*, pp. 316-330, Edinburgh, Scotland, September 1993.

[6] A.J. Bonner and M. Kifer, "An Overview of Transaction Logic," *Theoretical Computer Science*, vol. 133, pp. 205-265, October 1994.

[7] R.G.G. Cattell (ed.), *The Object Database Standard: ODMG-93,* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[8] S.W. Clyde, D.W. Embley, and S.N. Woodfield, "The Complete Formal Definition for the Syntax and Semantics of OSA," *Technical Report BYU-CS-92-2*, Computer Science Department, Brigham Young University, 1992.

[9] D.W. Embley, B.D. Kurtz, and S.N. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach*, Yourdon Press Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[10] A. Fernandes, M. Williams, and N. Paton, "A Logic-Based Integration of Active and Deductive Databases," to appear in the *Journal of New Generation Computing*, 1996.

[11] S. Gatziu, H. Fritschi, and A. Vaduva, *SAMOS an Active Object-Oriented Database System: Manual,* Technical Report 96.02, Computer Science Department, University of Zurich, February 1996.

[12] J. Harrison and S. Dietrich, "Integrating Active and Deductive Rules," *Proceedings of the 1st International Workshop on Rules in Database Systems*, pp. 288-305, Edinburgh, Scotland, September 1993.

[13] S. Khoshafian and G.P. Copeland, "Object Identity," *OOPSLA '86 Conference Proceedings*, pp. 406-416, Portland, Oregon, September 1986.

[14] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations of Object-oriented and Frame-based Languages," *Journal of the ACM,* vol. 42, no. 4, pp. 741-843, July 1995.

[15] M. Kifer, "Deductive and Object Data Languages: A Quest for Integration," *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, Singapore, December 1995, *Lecture Notes in Computer Science*, no. 1013, Springer Verlag, New York, 1995.

[16] S.W. Liddle, "Object-Oriented Systems Implementation: A Model-Equivalent Approach," *Ph.D. Dissertation*, Computer Science Department, Brigham Young University, 1995.

[17] S.W. Liddle, D.W. Embley, and S.N. Woodfield, "Integrating Object-Oriented Type Systems and Data Models in Harmony," *manuscript submitted for review.*

[18] S.W. Liddle, D.W. Embley, and S.N. Woodfield, "Unifying Modeling and Programming through an Active, Object-Oriented, Model-Equivalent Programming Language," *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling*, pp. 55-64, Gold Coast, Australia, December 1995, *Lecture Notes in Computer Science*, no. 1021, Springer-Verlag, New York, 1995.

[19] S.W. Liddle, D.W. Embley, and S.N. Woodfield, "A Seamless Model for Object-Oriented Systems Development," *Proceedings of the International Symposium on Object-Oriented Methodologies and Systems, ISOOMS 94*, pp. 123-131, Palermo, Italy, September 1994, *Lecture Notes in Computer Science*, no. 858, Springer-Verlag, New York, 1994.

[20] M. Liu and W. Yu, *The ROL System User Manual,* Release 1.0, Department of Computer Science, University of Regina, Regina, Saskatchewan, January, 1996.

[21] *OSM Lab Home Page*, World Wide Web URL *http://www.osm7.cs.byu.edu.*

[22] *Transaction Logic Prototype, http://www.db.toronto.edu:8020/people/bonner/tr.html,* University of Toronto, Toronto, Ontario, Canada, 1996.

[23] J.D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I,* Computer Science Press, Rockville, Maryland, 1988.

[24] J.D. Ullman, "A Comparison between Deductive and Object-Oriented Database Systems," *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, Lecture Notes in Computer Science*, no. 566, pp. 263-277, Springer-Verlag, New York, 1991.

[25] *UniSQL/X Application Program Interface Reference Guide,* UniSQL, Inc., Austin, Texas, 1992.

[26] Y. Wand, "A Proposal for a Formal Model of Objects," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F.H. Lochovsky (eds.), pp. 537-559, ACM Press, New York, 1989.

[27] P. Wegner, "Tradeoffs between Reasoning and Modeling," *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and Akinori Yonezawa (eds.), pp. 22-41, MIT Press, Cambridge, Massachusetts, 1993.

[28] C. Zaniolo, "A Unified Semantics for Active and Deductive Databases," *Proceedings of the 1st International Workshop on Rules in Database Systems*, pp. 271-287, Edinburgh, Scotland, September 1993.

[29] C. Zaniolo, "Active Database Rules with Transaction-Conscious Stable-Model Semantics," *Proceedings of the 2nd International Workshop on Rules in Database Systems, Lecture Notes in Computer Science*, no. 985, pp. 55-72, Athens, Greece, September 1995.

[30] S.B. Zdonik and D. Maier, "Fundamentals of Object-oriented Databases," *Readings in Object-Oriented Database Systems*, S.B. Zdonik and D. Maier (eds.), pp. 1-32, Morgan Kaufmann Publishers, San Mateo, California, 1990.