ELSEVIER

Data & Knowledge Engineering 22 (1997) 67–111

DATA & KNOWLEDGE ENGINEERING

CDOL: A comprehensive declarative object language☆

Susan D. Urban*, Anton P. Karadimce, Suzanne W. Dietrich, Taoufik Ben Abdellatif, Hon Wai Rene Chan

Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-5406, USA

Received 3 September 1993; revised manuscript received 25 March 1996; accepted 3 June 1996

Abstract

In this paper we present a rule-based database language known as CDOL (Comprehensive, Declarative Object Language) that is an integration of deductive, object-oriented and active database technology. CDOL provides sublanguages for the expression of derived data, constraints, updates and active rules. The rule-based query language of CDOL provides an expressive approach to extend the stored database with derived attributes and classes. The constraint sublanguage allows explicit declarative specification of integrity constraints as a basis for database consistency. The update sublanguage of CDOL enables ad-hoc declarative update requests, where updates are encapsulated in the methods associated with class definitions, thus conforming to traditional object-oriented design concepts. The active rule sublanguage provides active, user-transparent agents that support reactive behavior within CDOL applications. In particular, active rules can be used to supplement declarative updates to maintain database consistency with respect to the set of integrity constraints. Active rules in general are used to monitor the occurrence of specific events and to serve as alerters and triggers within a CDOL application. This paper presents the rule-based query language of CDOL and illustrates the manner in which the constraint, update and active rule sublanguages build on this declarative framework. The use of methods and transactions are also addressed, together with a discussion of the operational semantics of active rule processing.

Keywords: Object-oriented databases; Active rules; Deductive rules; Derived data; Constraints; Updates; Rule-based query language.

1. Introduction

Object-oriented database systems (OODBs) have been identified as the most promising platform for the increasingly complex demands of future database applications. The central OODB concept is that of an *abstract object*, such that each entity represented by an object has an immutable *identity* and a mutable *state* [9]. The concept of *object identifiers* allows data and structure sharing, thus eliminating data redundancy. An object's mutable state can be accessed

☆ This research was supported by NSF grant no. IRI-9109195 and NSF grant no. IRI-9410993.

* Corresponding author. Email: s.urban@asu.edu

PII S0169-023X(96)00041-9

(modified) only through a set of well-defined retrieval (update) *methods*. The set of methods defined for an object constitute the object's *interface*, serving as a foundation for providing appropriate encapsulation and protection capabilities.

Another important database research direction has been the incorporation of rule processing into database systems. Database systems with rule processing capabilities can be classified as *deductive databases* [25] and *active databases* [22]. Deductive database systems combine databases with logic to provide a declarative approach to database programming, using the concepts of derived data and recursive rule processing. Active databases, on the other hand, transform passive database processing into active environments by using rules as alerters and triggers and by using rules for constraint enforcement/satisfaction. Advanced applications such as engineering design and real-time systems need more active, intelligent environments rather than the passive environments provided by traditional database systems.

Although object-oriented, deductive and active database systems have primarily developed as separate areas of research, there are significant advantages that can be achieved through the integration of these concepts. As an example, one approach to the formalization of object-oriented databases has been based on the integration of deductive and object-oriented databases (DOODs) [2,3,21,31,33,40]. In a DOOD framework, basic object-oriented concepts are supported by a declarative rule-based language that serves as a primary tool for data definition and querying. Generally, declarative, object-oriented languages provide advantages such as rapid method development, associative retrieval of objects, definition of derived data and formation of user views.

Active database concepts have also been integrated with object-oriented database systems. Active rules are especially useful for the dynamic maintenance of integrity constraints as database updates occur. Active, object-oriented database prototypes such as Ode [26], Sentinel [6] and REACH [12] exemplify the recent work on building active rule processing into an object-oriented environment. Few DOOD systems, however, have addressed the issues of updates and constraint maintenance. The difficulty stems from the fact that database updates involve extra-logical operations that map one database instance into another. In other words, database updates do not fit easily into the first-order, model-theoretic framework of declarative database languages. Thus, declarative updates with active support for constraint maintenance in OODBs, complementing the use of declarative retrieval languages, has not been adequately addressed. Beeri in [8] writes: *It would be nice to have a more declarative approach to the specification of updates and some understanding of its power and limitations . . .*

In this paper we present a model and language that support the integration of object-oriented, deductive and active database concepts. The specific system is known as *CDOL* (*Comprehensive, Declarative, Object Language*). At the core of CDOL is a rule-based query language. The query language serves as a basis for the specification of derived attributes and derived classes in the schema definition process. The query language also serves as a foundation for several different sublanguages that are associated with CDOL: (1) a constraint sublanguage that allows elaborate expression of structural and application integrity constraints, (2) an update sublanguage that supports ad-hoc declarative specification of updates and (3) an active rule sublanguage, used to define active, production-type rules that serve as a bond between declarative updates and declarative integrity constraints.

Our goal in this paper is to present the complete language framework of CDOL. A formal definition of the object model of CDOL, together with an initial presentation of the update language appeared in [35]. An earlier version of the sublanguages of CDOL also appeared in [36]. Since our initial work on CDOL, the model has been redefined in [39] and the CDOL framework has been adopted as the basis for the establishment of an active rule development environment, supporting the investigation of analysis, debugging and testing issues associated with active database rules (see http://www.eas.asu.edu/~adood). CDOL has thus been transformed into a more specific, user-oriented language framework to support the implementation and research associated with the active rule development environment. In particular, the query language and update language have been extended, the constraint language has been redefined, the details of specifying methods and transactions in schema development have been clarified, and the structure and semantics of active rules has been elaborated. CDOL is currently being implemented using the Shore Storage Manager [14] for construction of the active database environment. This paper describes the language features of CDOL that we are currently implementing in support of the active database environment.

In this paper, we first present the structural features of the CDOL data model, together with the definition of the rule-based language. We then show how CDOL provides a basis for creating an application environment. In particular, we illustrate the use of CDOL for defining derived data, constraints and updates. With respect to derived data, we show how CDOL can be used as a retrieval method definition vehicle for deriving simple and parameterized virtual attributes and classes. The ability to define virtual attributes and virtual classes creates an expressive environment for extending the schema associated with the stored database with intensional aspects that would otherwise be difficult to maintain.

We then show how CDOL supports an explicit representation of schema constraints using the constraint sublanguage. In general, the declarative representation and enforcement of data semantics in OODBs has received little attention [17,52]. Procedural approaches to constraint enforcement are error-prone and provide no support to the programmer for correctly and consistently enforcing constraints, especially in constraints that involve multiple classes. In this research, we use CDOL to express constraints declaratively, where constraints include those that define the semantics of the data model as well as other application-specific constraints.

After discussing derived data and constraints, we then present the update sublanguage of CDOL. The update sublanguage is also rule-based, where the rule body specifies the object to be updated and the rule head specifies update operations. An important component of our approach to constraints and updates is found in the use of the active rule sublanguage. Active rules are entities that are automatically triggered by operations of the update sublanguage. The transparent execution of active rules within CDOL provides checking for database consistency violations and accordingly restores database integrity by invoking other update methods. Active rules, in general, are used to monitor the occurrence of specific events and to serve as alerters and triggers in CDOL database applications.

In summary, starting from a standard object model with a declarative query language, we have enhanced the model with active capabilities. We have also developed an object-oriented database model that provides a high-level, declarative retrieval and update language, while supporting integrity constraint definition, checking and enforcement transparently to the end user. As in other object-oriented database models, CDOL associated methods with classes

and all updates are encapsulated within these methods. The approach presented has considerable advantages over the case of procedural encoding of update methods. Firstly, updates are specified using the same declarative framework as retrievals. Secondly, the concept of integrity constraints is clearly separated and uniformly supported using the specific mechanism of active rules. This makes update requests and transactions simpler and less error-prone. Furthermore, changes in the schema and/or constraints can be easily accommodated, since the constraint-handling procedures are localized in declaratively-stated active rules. Thirdly, active rules enhance database flexibility by enabling the development of view update and materialized view maintenance functions.

The remainder of this paper is structured as follows. An overview of the CDOL language features is presented in Section 2, followed by a definition of the rule-based query language in Section 3. Details related to the constraint, update and active rule sublanguages are then presented in Sections 4, 5 and 6, respectively. After a discussion of related work in Section 7, the paper concludes in Section 8 with a summary of the work and a discussion of our future directions.

2. An overview of CDOL

This section provides an overview of CDOL and the different components of the language that exist for the definition of applications using CDOL. As an active, deductive, object-oriented language, CDOL was initially defined with the formal, structural definition of the language that appeared in [35,36]. The original definition of the language was based on formal definitions of object-oriented models such as O2 [42] and IQL [3]. The formal basis for CDOL has since been redefined in [39] using an order-sorted algebraic definition.

An order-sorted algebra, where sorts are partially ordered to reflect subsort inclusion, represents a natural approach for algebraic modeling of object-oriented data models, especially considering the salient features of object-oriented models such as strong typing, class hierarchies, attribute inheritance and polymorphism. In [39], Karadimce uses the order-sorted algebraic approach to develop a correspondence between a CDOL schema and an order-sorted signature and a correspondence between an instance of a CDOL database and an order-sorted algebra. It is not the purpose of this paper to present the formal, algebraic definition of CDOL; such an exercise is beyond the scope of the current paper. The algebraic definition of CDOL does, however, represent an essential prerequisite towards the development of CDOL as a declarative, logic-based query framework for an object-oriented database. The reader is referred to [39] for further details about the formal basis of CDOL. The remainder of this section provides a global view of the language features that are described in further detail in the following sections.

To support the discussion of CDOL throughout this paper, we will use a running example based on a Horse Racing Database (HRDB) application as presented in Fig. 1. The notation in Fig. 1 follows the graphical notation that is used in conjunction with the object model standard as defined in [16], where rectangles represent classes and bold arrows represent subclass relationships. Thin arrows between classes represent object relationships (single-

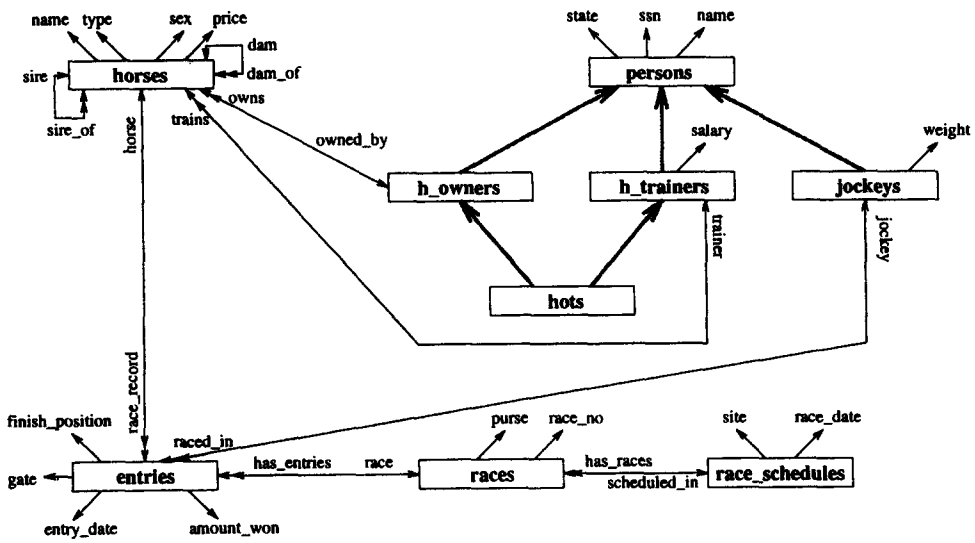


Fig. 1. Horse racing database application.

valued or set-valued as indicated by the single or double arrow head, respectively). Arrows from rectangles to text represent attributes of classes.

In particular, the diagram in Fig. 1 defines a database that is composed of horses and persons as well as additional information about horse races. The persons class is further decomposed into the h_owners, the h_trainers and the jockeys subclasses, respectively representing those individuals that own horses, those individuals that train horses and those individuals that ride horses in races. The hots class is a subclass that inherits from h_owners and h_trainers, representing horse owners that are also horse trainers.

The diagram in Fig. 1 also contains classes that represent information about races. The class race_schedules contains objects that represent race schedules for each racing day of the year for a particular race track. Each race_schedules object is composed of one or more objects from the races class. Each object from the races class is in turn composed of one or more entries objects, containing information about the horses and jockeys that are involved in each race.

Rather than create our own data definition language, we have adopted the use of the Object Definition Language of the Object Data Management Group (ODMG) [16], with appropriate extensions to support the declarative and active features of CDOL. A complete textual description of the Horse Racing Database application appears in Appendix A.[1] Relevant subparts of the description will be repeated in the main body of the paper to support the description of CDOL.

The HRDB in Appendix A is subdivided into several parts, corresponding to the main components of application specification using CDOL. Those parts include (1) type and class definition, including virtual classes, (2) constraint specification, (3) passive rules for derived attribute/class specification, (4) rule-based method and transaction definition and (5) active

[1] The horse racing schema together with a BNF definition of CDOL can be found at http://www.eas.asu.edu/~adood.

rule specification. The *rule-based query language* of CDOL is a fundamental component of the language, providing the expression of derived attributes and classes in the schema definition using passive (i.e., deductive) rules. The *constraint sublanguage*, the *update sublanguage*, and the *active rule sublanguage* are also based on the declarative query language of CDOL. Methods and transactions are composed of update rules expressed using the update rule sublanguage, following the object-oriented paradigm in which modifications to objects are encapsulated in the methods associated with each class.

Classes are defined in the standard manner as described in [16]. In particular, classes can be defined as subclasses of one or more superclasses. Classes can also have extents and keys. Each class is characterized as having specific properties which include attributes for describing the state of an object and relationships for describing 1:1, 1:M and M:N relationships between objects, including the specification of inverse relationships. Method signatures define the operations associated with class behavior. With respect to class instances, CDOL conforms to the notion of *disjoint identifier assignment* as described in [2]. In particular, an object can exist as an instance of a single class only. The notion of inheritance is then captured through the additional notion of *inherited identifier assignment* [2], where the set of all instances of a class is defined to consist of the direct instances of a class together with the instances of all subclasses of the class. The objects of a subclass therefore inherit the attributes, relationships and methods defined in its superclasses.

Our extension to class definition supports the specification of derived, or *virtual*, attributes using the CDOL rule-based query language. As an example of a class definition in CDOL, consider the specification of the horses class:

```
class horses
(
    extent horses
    key (name, type))
{
    attribute string name ;
    attribute string type ;
    attribute string sex ;
    attribute short price ;
    relationship horses sire inverse horses::sire_of ;
    relationship horses dam inverse horses::dam_of ;
    relationship set ⟨horses⟩ sire_of inverse horses::sire ;
    relationship set ⟨horses⟩ dam_of inverse horses::dam ;
    relationship h_owners owned_by inverse h_owners::owns ;
    relationship h_trainers trainer inverse h_trainer::trains ;
    relationship set ⟨entries⟩ race_record inverse entries::horse ;

    virtual short salary_of_trainer = trainer_salary_rule ;
    virtual set ⟨horses⟩ ancestors = horse_ancestor_rule ;
```

```
    virtual short total_race_count = total_race_count_rule ;
    virtual short average_win_amount = average_win_amount_rule ;

    void transfer(h_owners:New_Owner) ;
    void assign_trainer(h_trainers:New_Trainer) ;
;{
```

The horses class has four attributes defining the name, type, sex and price of a horse. The sire relationship definition provides an example of a recursive relationship with the horses class, defining the father of a horse. The sire_of inverse defined a set-oriented relationship, indicating the children that a given horse has sired. A similar definition exists for the dam/dam_of relationship. The owned_by and trainer relationships define the owner and trainer of a horse object. The race_record relationship defines the set of entries that a horse has participated in.

In addition to attributes and relationships, the horses class definition also defines four virtual attributes: salary_of_trainer (deriving the salary of the trainer of a horse), ancestors (deriving the set of horses that represent the ancestors of a horse), total_race_count (deriving the total number of races a horse has participated in), and average_win_amount (deriving the average winning amount of a horse over all the races that the horse has participated in). The declarative rules that derive each virtual attribute value, as well as rules that derive virtual class instances, are specified in the derived attributes/classes section of the application definition using the rule-based query language of CDOL. As examples of virtual classes, thoroughbreds and expensive_horses are classes that are derived from the horses class based on conditions that are defined in the body of each rule defining the class. The ability to define virtual classes provides a flexible way of extending the schema definition to support different user views, thus creating a more expressive application environment, without explicitly maintaining the extension of each class.

Two explicit methods are associated with the horses class for transferring ownership of a horse and assigning a trainer to a horse. Method definitions are specified using the rule-based update language of CDOL. The rule-based definitions of transfer and assign_trainer are defined in the methods and transactions specification section of Appendix A. In addition, several default methods are automatically associated with every class definition. For example, each class has default methods to create objects of the class, delete objects from the class, and modify attribute and relationship values. The default methods are not shown in Appendix A, but examples are presented in Section 5.3. Finally, free-standing transactions can also be defined that are not associated with any specific class. Transactions provide a way of bundling together update rules into meaningful application procedures.

An important component of application development in CDOL is the specification of constraints. Some constraints can be automatically generated from class definitions, such as constraints associated with the keys of each class. Other constraints must be explicitly stated in the constraint specification section of a CDOL schema using the CDOL constraint sublanguage. Since the constraint sublanguage is constructed from the query language, complex constraints can be expressed in CDOL using universal and existential quantifiers, as well as aggregate functions. The quarter_horses constraint in the constraint specification part of the

schema definition provides an example of a constraint stating that all horses in a specific race must be quarterhorses.

The final component of application development in CDOL is the active rule specification section. Active rules are different from the rule-based query language in that they cause reactive behavior within the database in response to the occurrence of specific events. Active rules can be used to define automated responses to constraint violations, specifying corrective actions or alerting the user of undesirable conditions. Active rules can also be used to monitor events and conditions of specific interest that are not necessarily associated with constraints and to trigger associated actions and notifications within the application. The trainer_salary and purge_horse_owner_information active rules are examples of rules that can be used to automatically respond to the occurrence of specific events.

In summary, CDOL not only supports fundamental object model features, but it also supports the definition of virtual attributes and virtual classes, the definition of constraints, the use of update rules in the definition of methods and transactions, and the definition of active rules. The expression of constraints, update rules and active rules all build on the fundamental CDOL rule-based query language. The following sections elaborate on the CDOL query language and its use within the different sublanguages of CDOL.

3. Rule-based query language of CDOL

Given the overview of CDOL in the previous section, we now present the rule-based query language that allows declarative definition of derived data as well as the declarative expression of constraints, update conditions and active rule conditions. Section 3.1 first presents the fundamental concepts of query expression in CDOL. The use of the rule-based query language in the expression of virtual attributes and virtual classes is then addressed in Sections 3.2 and 3.3, respectively. Miscellaneous features of the language are described in Section 3.4.

3.1. General query language constructs

Besides type and class names, typed constants and object identifiers, we assume countably infinite sets of function symbols and variables. By convention, variables start with uppercase letters, and constants, attribute labels, object identifiers and function symbols start with lowercase letters.

Informally, *id-terms* are basic components of the query language that serve for object identity denotation. Id-terms have the form $C: V$, denoting a variable V ranging over the class C (i.e., V represents an object of class C whose identity is not bound). Id-terms can also have the form $C: o$, denoting an object of class C with identity o.

Terms of the language can be id-terms, simple terms of the form $T = c$ or $T = V$ (where T is a type name, c is a constant, and V is a variable), or complex set or tuple terms. If t is an id-term of type τ and $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively, then $t[t_1, \ldots, t_n]$ is a term of type $[\tau_1, \ldots, \tau_n]$, called a *tuple term*. If $A_1, \ldots, A_n$ are appropriate labels (or attribute names), then $t[A_1 = t_1, \ldots, A_n = t_n]$ is a tuple term with named attributes. Finally, if T is a type name of type $\{\tau\}$, and $t_1, \ldots, t_n$ are terms of type τ, then $T = \{t_1, \ldots, t_n\}$ is a

term of type $\{\tau\}$, called a *set term*. As usual, a *ground term* is a term with no variables appearing in it. Terms that are not ground will be called non-ground terms.

Given the schema in Fig. 1, examples of id-terms are illustrated below:

h_owners:joe[name = "Joe Smith", ssn = "123456789", state = "AZ", owns = {h_1, h_2}]

horses:h_1[name = "Pegasus", type = "thoroughbred", sex = "M", price = 10000,
dam = h_3, sire = h_4, owned_by = joe, trainer = bob]

horses:h_2[name = "Oil Patch Pappa", type = "thoroughbred", sex = "M", price = 50000,
dam = h_5, sire = h_6, owned_by = joe, trainer = bill]

The above examples illustrate three tuple terms for the objects joe, h_1 and h_2. The object joe contains a set term to indicate the horses that Joe Smith owns. Each horse object references other horse objects through the sire and dam attributes. The owned_by and trainer attributes reference persons objects.

Although the above examples denote specific objects in the database, explicit references to object identifiers are not supported as part of the query language. Objects must be indirectly referenced through variables. For example, the following id-term is a non-ground term that refers to a horse owner from Arizona with a specific ssn:

h_owners:J[name = N, ssn = "123456789", state = "AZ", owns = {H}]

In the above example, J is a variable that is bound to the object that satisfies the condition. In general, there can be more than one object that satisfies the specified condition. The variable N in this case will be bound to the name of the object and the variable H will range over the elements of the set of horses owned by J. The owns relationship could have alternatively been written as owns = H, in which case H is bound to the entire set value rather than the individual elements of the set.

Given the above definitions, we can now define atoms, literals and rules. An *atom* (or *atomic formula*) is either a term (with no explicit references to object identifiers), an *equality atom* $t_1 = t_2$ (where t_1 and t_2 are terms with the same type and = is the typed equality predicate), or a *set membership atom t in* t_s (where t_s is a set term of type $\{\tau\}$ and t is a term of type τ). Built-in predicates (e.g., $t_1 \leqslant t_2$), arithmetic expressions (e.g., New_salary = Old_salary * 1.1), and externally evaluated functions (e.g., $t_1 = sum(t_1, t_2)$) can also be used as atomic formulae.

An atom is a literal, called a *positive literal*. If A is a positive literal, then *not* A is a *negative literal*. Literals can also be existentially or universally quantified. For example, assume H is bound to the entire set of horses owned by some owner. The following literal expresses that all horses H1 in the set H be female:

for_all H1 in H (H1[sex = "F"])

A *passive rule* r is a clause of the form $A \leftarrow L_1, L_2, \ldots, L_n$ where A is an atom (a non-ground term only), called the *head* of the rule and denoted as *head*(r). $\{L_1, \ldots, L_n\}$ is a set of literals comprising the *body* of the rule, denoted as *body*(r). The term "passive rule" is used to distinguish rules that define derived data from the active rules that invoke specific

actions in response to the occurrence of prespecified events. Active rules are defined in Section 6.

3.2. Virtual attributes using CDOL rules

As indicated in the overview of Section 2, the set of stored attributes (single-valued or set-valued) for a given class C can be extended by a set of virtual attributes that are defined using CDOL passive rules. The values of such attributes are not directly stored in the database but are derived when the attribute is referenced within a query language expression. Virtual attributes can be *simple* or *parameterized*. In general, the value V of a simple virtual attribute a for an arbitrary object O of class C is defined by the set of CDOL rules of the format:

$$C: O[a = V] \leftarrow L_1, \ldots, L_n ;$$

where L_i's are arbitrary literals defining the bindings for object variable O and value V. A parameterized virtual attribute p is defined by a set of CDOL rules of the format:

$$C: O[p(\textit{param-list}) = V] \leftarrow L_1, \ldots, L_n ;$$

where *param-list* is a list of typed parameters, that appear in $L_1, \ldots, L_n$.

Recall for the horses class definition that the salary_of_trainer virtual attribute is used to derive the salary of a trainer as an attribute of a horse object:

```
virtual short salary_of_trainer = trainer_salary_rule ;
```

The salary_of_trainer value is calculated using the trainer_salary_rule:

```
rule trainer_salary_rule
{
    horses:Horse[salary_of_trainer = Salary] ← horses:Horse[trainer = T], T[salary = Salary] ;
} ;
```

The strong typing requirement is obeyed sufficiently so that the system can fill out the missing type specifications using standard inference techniques and typing information from the base database schema. The use of attribute names in id-terms allows the omission of other attribute values of horses and h_trainers that are not relevant for the definition of the salary_of_trainer virtual attribute.

For single-valued properties, CDOL supports the dot notation to combine dereferencing and attribute selection. Using dot notation, the previous rule can be rewritten as:

```
rule trainer_salary_rule
{
    horses:Horse[salary_of_trainer = Salary] ← horses:Horse, Horse.trainer.salary = Salary ;
} ;
```

Dot notation is used to avoid the unnecessary introduction of "join variables" (i.e., variable T in the original version of the rule above) and to provide a more object-oriented approach to the expression of conditions.

Simple virtual attributes can also be defined by multiple rules, possibly involving recursion, as in the rules for defining the ancestors virtual attribute of a horse. The class definition of horses indicates that the horse_ancestor_rule is used to derive the value of the virtual attribute ancestors:

```
rule horse_ancestor_rule
{
   horses:Horse[ancestors = {Ancestor}] ← horses:Horse, Horse.dam = Ancestor ;
   horses:Horse[ancestors = {Ancestor}] ← horses:Horse, Horse.sire = Ancestor ;
   horses:Horse[ancestors = {Ancestor}] ← horses:Horse, Horse[ancestors = {Y}],
     Y[ancestors = {Ancestor}] ;
} ;
```

Note that the ancestors of a horse are collected into a set value. In the above example and hereafter, the notation $X[attr = \{Y\}]$ is used as an abbreviation for $X[attr = Z]$, Y *in* Z.

As examples of parameterized virtual attributes, the race_schedules class defines two virtual attributes with parameters: one to identify races with a purse over a certain value and the other to identify races with a purse in a specified range. The corresponding rules are:

```
rule purse_over_rule
{
   race_schedules:Schedule[purse_over(short:Low) = {Y}] ←
     race_schedules:Schedule, Schedule.has_races = {Y}, Y.purse ≥ Low ;
} ;

rule purse_between_rule
{
   race_schedules:Schedule[purse_between(short:Low, short:High) = {Y}] ←
     race_schedules:Schedule, Schedule.has_races = {Y}, Y.purse ≥ Low, Y.purse ≤ High ;
} ;
```

The first rule defines for each race_schedules object the set of races objects with a purse over the parameter value Low. The second rule has two parameters: for a given race schedule, it returns the set of race objects with a purse between Low and High. Parameterization is a useful feature for the expression of derived data, eliminating the unnecessary duplication of rules that calculate similar concepts using different values. Further examples of the expressiveness of parameterization appear in the following subsection on virtual classes.

3.3. Virtual classes using CDOL rules

Classes as object containers serve to organize objects into an inheritance hierarchy. Since a database serves a number of users, each having differing application demands, a predefined and stored class inheritance hierarchy would be quite complex in order to capture the

requirements of each user. A complex stored inheritance hierarchy can complicate database management functions and can also complicate the user's perception of data.

An elegant solution to the above problem is proposed in [2]: the user's customization is achieved by defining *virtual classes* on top of stored classes. Stored and virtual classes are then conceptually integrated into a single class hierarchy. Furthermore, virtual classes are explicitly specified only by defining their membership, while their position in the conceptual class hierarchy and the set of applicable methods can be inferred by the system. Work describing the positioning of classes in a conceptual class hierarchy is addressed in [23].

As described in [2], virtual classes can be defined in terms of specialization or generalization. Using specialization, a virtual class is defined as a *subset* of a previously defined (stored or virtual) class. Using generalization, a virtual class is defined as a *union* of (stored or virtual) classes. We have captured both concepts for virtual classes in CDOL.

A virtual class C can be defined by a set of CDOL rules of the format:

$$C: O \leftarrow L_1, \ldots, L_n ;$$

where L_i's are arbitrary literals defining the bindings for object variable O. A parameterized virtual class C is defined by a set of CDOL rules of the format:

$$C(\textit{param-list}): X \leftarrow L_1, \ldots, L_n ;$$

where *param-list* is a list of typed parameters, that appear in $L_1, \ldots, L_n$.

The following examples illustrate the use of specialization in CDOL:

```
rule thoroughbreds_rule
{
   thoroughbreds:Tbreds ← horses:Tbreds[type = "thoroughbred"] ;
} ;

rule male_tb_rule
{
   male_thoroughbreds:MTbreds ← thoroughbreds:MTbreds[sex = "male"] ;
} ;

rule champ_rule
{
   champions:Champ ← horses:Champ[race_record = {Race_Rec}],
      Race_Rec[finish_position = 1, amount_won = Amt_Won], Amt_Won ≥ 100000 ;
} ;
```

The thoroughbreds virtual class specializes the base horses class. The male_thoroughbreds virtual class further specializes the thoroughbreds class. The third example defines that the champions virtual class consists of horses that have won first place in a race with a first-place winning amount of at least $100,000.

Added flexibility in the definition of specialized virtual classes is achieved by the concept of

parameterized classes. A set of related parameterized virtual classes is characterized by the same virtual class name and a different (set of) parameter value(s). For example, one may need virtual classes consisting of horse objects trained by the same trainer:

```
rule trained_by_rule
{
    horses_trained_by(h_trainers:The_Trainer):Horse ← horses:Horse[trainer = The_Trainer] ;
} ;
```

The stored database schema would be hardly manageable and would be subject to continual modification if many parameterized classes were defined as stored classes. Note that some of the parameterized classes may be empty and the number of related parameterized classes can easily be very large.

Generalization allows bottom-up enhancement of the class hierarchy, by defining larger classes as unions of smaller (base or virtual) ones. For example, the class of "Little Ranch" horses can be defined as the union of horses trained by Jim and John (assuming Jim and John are the only trainers at "Little Ranch"):

```
rule little_ranch_rule
{
    little_ranch_horses:LRHorses ← h_trainers:Trainer_Jim[name = "Jim"] ,
      horses_trained_by(Trainer_Jim):LRHorses ;
    little_ranch_horses:LRHorses ← h_trainers:Trainer_John[name = "John"] ,
      horses_trained_by(Trainer_John):LRHorses ;
} ;
```

Methods applicable to all generalized classes (e.g., horses_trained_by(Trainer_Jim) and horses_trained_by(Trainer_John)) are also applicable to the generalization class (e.g., little_ranch_horses). Even if there are no methods in common, the membership in the generalization class is still well defined, though such a class has little meaning except that of a union of two object sets.

3.4. Other language features

In the definition of derived attributes and classes, it is often useful to use existential and universal quantification in the expression of rule conditions. Aggregate functions are also useful in the practical expression of conditions involving database objects.

As an example of quantification, consider the expression of the following virtual classes associated with the jockeys and horses classes. In particular, we need to create derived subclasses that represent (1) jockeys that have finished first, second or third in at least one race, and (2) horses that have finished first, second or third in all races in which they have participated. Case 1 requires the use of existential quantification over the raced_in relationship of the jockeys class. Case 2 requires universal quantification over the race_record relationship of the horses class. The specific rule definitions are shown below:

```
rule ever_win_rule
{
    ever_win_jockeys:Jockey ← jockeys:Jockey, exists Entry in Jockey.raced_in
      (Entry.finish_position ≥ 1, Entry.finish_position ≤ 3) ;
} ;

rule never_lose_rule
{
    never_lose_horses:Horse ← horses:Horse, for_all Entry in Horse.race_record
      (Entry.finish_position ≥ 1, Entry.finish_position ≤ 3) ;
} ;
```

In each case, the quantifier is defined over a set-valued attribute. Following the quantifier specification is a list of literals in parentheses that are evaluated under the scope of the quantified expression.

CDOL also supports aggregate functions (such as avg, count, sum, product, min and max) for the expression of aggregate values over set-valued properties or for the calculation of aggregate values over single-valued properties of class objects.

As an example, suppose that we want to define two virtual attributes for the horses class that give (1) the total number of races a horse has participated in, and (2) the average winning amount of a horse. The rule definitions for such virtual attributes are:

```
rule total_race_count_rule
{
    horses:Horse[total_race_count = Count] ←
      horses:Horse, Count = count(Horse.race_record) ;
} ;

rule average_win_amount_rule
{
    horses:Horse[average_win_amount = Amt] ←
      horses:Horse, Amt = avg(Entry.amount_won over Entry in Horse.race_record) ;
} ;
```

In the above examples, the count function is applied over a set-valued relationship (race_record of the horses class), whereas the avg function is applied over a single-valued attribute (amount_won) of the entries class objects corresponding to a horses object's race_record. The semantics of the count function are straightforward, computing the cardinality of the relationship. However, the semantics of the avg function deserve more clarification. When aggregates are applied over a single-valued attribute of class objects, the aggregate function specification consists of two parts separated by the reserved word over. The first part is a single-valued path expression that must begin with a new variable. The new variable must be

bound in the second part of the aggregate function specification that takes the form of a rule body. The semantics for this form of aggregate function is that the single-valued path expression is evaluated for each binding of the new variable bound by the second part of the aggregate function specification to produce a collection. Then the aggregate is applied over the collection.

Aggregate functions by default assume multiset (bag) bindings in the domain being aggregated. Therefore the average_win_amount_rule will give the correct result even if a horse wins two or more races with the same amount. If a set binding is desired, the set construct (braces around the aggregate path expression) must be used. For instance, to create a virtual attribute for the race_schedules class that gives the number of different jockeys participating in a race schedule, the following rule can be used:

```
rule jockey_count_rule
{
    race_schedules:Sched[jockey_count = J_Cnt] ←race_schedules:Sched,
      J_Cnt = count({Entry.jockey} over Race in Sched.has_races, Entry in Race.has_entries) ;
};
```

Here, the set construct around Entry.jockey guarantees that if a jockey races more than once in one race schedule, he/she will be counted only once.

4. The constraint sublanguage of CDOL

An important aspect in the description of any practical database application is the expression of constraints. Constraints provide additional semantic knowledge for an application by restricting the set of allowed instances of the database. In an object-oriented environment, constraints can be encapsulated in the methods associated with each class. Constraints, however, also represent a source of knowledge about objects and their associations that ideally should be available to the database management system to support intelligent reasoning capabilities. Since CDOL represents an integration of deductive and object-oriented concepts, our approach is to represent constraints in a declarative form, rather than to embed knowledge of application constraints within code. This section presents the constraint sublanguage of CDOL and the way in which the basic CDOL query language is used in the expression of constraints. The active rule language for the automatic maintenance of integrity constraints will be addressed in Section 6.

Informally, constraints are used to represent arbitrary assertions that must be true in the database instance. In CDOL, a *constraint* is expressed in the form:

constraint *cname*
  if $L_1, L_2, \ldots, L_n$
  then $M_1, M_2, \ldots, M_m$;

where *cname* is the name of the constraint and L_i and M_j are arbitrarily literals of the CDOL query language. A CDOL constraint is interpreted as "if the condition in the if clause is true,

then the condition in the then clause *must* be true." If $m = 1$ and M_1 is the distinguished constant false, then the constraint is called a *denial*, indicating that the condition in the if clause cannot be true in any database state.

As an example of a CDOL constraint specification, consider the inverse relationship between the owns and owned_by relationships in Fig. 1. An inverse relationship can be expressed using two separate constraints to express the constraint from the point of view of each relationship involved:

```
constraint owner_owns_1
  if h_owners:Owner[owns = {Horse}]
  then horses:Horse[owned_by = Owner];

constraint owner_owns_2
  if horses:Horse[owned_by = Owner]
  then exists H in Owner.owns(H = Horse);
```

As another example, consider the external key associated with the entries class in Fig. 1, consisting of the horse and jockey relationships and the entry_date attribute. This constraint can be expressed in two different forms, where one form expresses the constraint as a denial. Both forms of the constraint are shown below:

```
constraint entries_key_denial
  if entries:E1[horse = H, jockey = J, entry_date = D],
    entries:E2[horse = H, jockey = J, entry_date = D], not (E1 = E2)
  then false;

constraint entries_key
  if entries:E1[horse = H, jockey = J, entry_date = D],
    entries:E2[horse = H, jockey = J, entry_date = D]
  then E1 = E2;
```

Constraints such as the inverse and key constraints above can be automatically generated from the schema definition. Other constraints must be explicitly stated by the application designer. Since the basic CDOL query language supports the use of quantification, aggregation and arithmetic expressions, these features are also available to support the explicit statement of complex application constraints. As an example, the following constraint uses the count aggregate function to express the cardinality constraint that each race_schedules object must be composed of 10 races:

```
constraint number_of_races
  if race_schedules:Schedule[races = Race]
  then count(Race) = 10;
```

As an example of a more complex constraint involving quantification, the following constraint expresses the fact that all horses in race 1 on May 16, 1996 must be quarterhorses:

```
constraint quarterhorses
  if races:Race[race_no = 1, scheduled_in = Sched], Sched[race_date.day = 16,
    race_date.month = "May", race_date.year = 1996]
  then for_all Entry in Race.has_entries (Entry.horse.type = "quarterhorse") ;
```

Constraints such as those defined above can be expressed to form the application semantics that define the valid relationships that must exist in a consistent database state. In Section 6, we will return to the issue of constraints to address the use of active rules in CDOL for dynamic maintenance of constraints as updates occur. Section 5, however, first presents the update sublanguage of CDOL.

5. The update sublanguage of CDOL

In this section we present a rule-based update sublanguage that builds on the CDOL query language to provide a basis for the construction of methods and transactions that create, modify and delete objects in a CDOL application. Section 5.1 presents the fundamental concepts of update rules, together with several examples. The execution semantics of update rules is then presented in Section 5.2. The use of update rules to construct methods and transactions is addressed in Section 5.3.

5.1. Update rules and update transactions

A CDOL update rule is an augmentation of the rule-based query language, where the rule body specifies the object to be updated and the rule head specifies the update operations. In order to define the semantics of update rules, we first define *primitive update terms*. Primitive update terms can be either object update terms or property update terms. Recall that the term "property" in CDOL refers to attributes of objects as well as relationships between objects. Intuitively, an object update term represents an object-create, object-destroy, or object-migrate request for a single object. A property update term represents an update request for a single property of a single object. A property update term can be single-valued or set-valued, depending on the property type.

If C: O is an id-term, then C: *new* O and C: *destroy* O are *primitive object-create* and *primitive object-destroy update terms*, respectively. A primitive object-create term generates a new object identifier and disjointly assigns the object to the specified class. A primitive object-delete operator removes the object from the specified class and from all of its subclasses. A *primitive object-migrate update term* has the form C: *migrate* O. Intuitively, if O denotes an existing object, C: *migrate* O is an object-migrate request that O be disjointly assigned to C.

If t is an id-term, A_i is a single-valued property label and t_i is a term of an appropriate type, then $t[A_i := t_i]$ is a *primitive single-valued property-modify* operation. Note the use of the $:=$ assignment operator to represent modification of A_i. A value of a single-valued property

can be deleted by using a property-modify operation to set the value of an attribute to *nil*, as in $t[A_i := nil]$.

If t is an id-term, A_i is a set-valued property label and t_i is a term of an appropriate type, then $t[A_i := \{+t_i\}]$ and $t[A_i := \{-t_i\}]$ are *primitive set-valued property-insert* and *primitive set-valued property-delete update terms*, respectively.

Primitive update terms for set-valued properties allow for insertion or deletion of a single element at a time. To express multiple insertions or deletions from a set-valued property, we use a *composite update term* of the form $t[A_i := t_i]$, where A_i names a property of type $\{\tau_i\}$ and t_i is of type $\{\tau_i\}$. A composite update term of the form $t[A_i := \{\}]$ assigns a set-valued property to the empty set. Set operations on set-valued properties are also supported with composite update terms of the form $t[A_i := t_i \; setop \; t_j]$, where t_i and t_j are of type $\{\tau_i\}$ and *setop* is either *union*, *diff* or *int*, representing the union, difference and intersection operators, respectively.

We generalize *composite update terms* to a set of multiple atomic update requests to a single object. Informally, a composite update consists of a combination of at most one object update, multiple primitive single-valued updates, and multiple primitive and/or composite set-valued updates. Formally, a composite update term has the form $t[A_{i_1} := u_1, \ldots, A_{i_k} := u_k]$, where

(i) t is an id-term, an object-create update term, or an object-migrate update term,

(ii) $\{A_{i_1}, \ldots, A_{i_k}\}$ is a subset of the property labels of t,

(iii) if A_{i_j} $(1 \leq j \leq k)$ is a single-valued property label, then u_j is a property-modify update term of the appropriate type,

(iv) if A_{i_j} $(1 \leq j \leq k)$ is a set-valued label naming a property of type $\{\tau\}$, then u_j is

$\{+t_j\}$ or $\{-t_j\}$, where t_j is of type τ, or

t_j, where t_j is of type $\{\tau\}$, or

$t_i \; setop \; t_j$, where t_i and t_j are of type $\{\tau\}$,

(v) if t is an object-create update term, then no primitive set-valued update term is a delete term, and,

(vi) if t is an object migrate term, then no primitive set-valued update term for a new property assignment is a delete term.

An *update rule* r_u is a clause of the form $U \leftarrow L_1, L_2, \ldots, L_n$, where $n \geq 0$, U is a primitive or composite update term and $L_1, L_2, \ldots, L_n$ are literals that serve as qualifying conditions for the occurrence of the update U. Based on the condition of the update rule, U is a *set-oriented update* if more than one object satisfies the qualifying condition.

The following examples illustrate the use of CDOL update rules:

(a) Create a jockey object with ssn = "700092233" and name = "Jim Rider".

```
jockeys:new Jim[ssn := "700092233", name := "Jim Rider"] ;
```

(b) Remove John from the h_trainers class. The object will remain in the persons class.

```
h_trainers:destroy John ← h_trainers:John[name = "John"] ;
```

(c) Destroy all entries objects corresponding to the third race at the New Orleans race track on May 24, 1996.

```
entries:destroy Entry ← race_schedules:RSched[site = "New Orleans",
  race_date.day = 24, race_date.month = "May", race_date.year = 1996,
  has_races = Race], races:Race[race_no = 3, entries = {Entry}] ;
```

(d) Migrate the owner of the horse named "Sting" into the hots class.

```
hots:migrate Sting_Owner ← horses:Sting[name = "Sting", owned_by = Sting_Owner] ;
```

(e) Modify the name property of the horse named "Sting" to "Stinger".

```
horses:Horse[name := "Stinger"] ← horses:Horse[name = "Sting"] ;
```

(f) Delete the trainer property of "Stinger".

```
horses:Horse[trainer := nil] ← horses:Horse[name = "Stinger"] ;
```

(g) Insert the finishing position (2) and the amount won (150000) for a specified entry object.

```
entries:Entry[finish_position := 2, amount_won := 150000] ←
  entries:Entry[horse.name = "Stinger", jockey.name = "Jim Rider",
  entry_date = today( )] ;
```

(h) Insert "Stinger" into the owns property of the horse owner "Jim K.".

```
h_owners:Jim[owns := {+Stinger}] ←
  h_owners:Jim[name = "Jim K."], horses:Stinger[name = "Stinger"] ;
```

(i) Insert all horse objects owned by "Joe T." into the owns property of "Jim K.".

```
h_owners:Jim[owns := Jim.owns union Joe.owns] ←
  h_owners:Jim[name = "Jim K."], h_owners:Joe[name = "Joe T."] ;
```

(j) Delete the owns property of "Joe T.".

```
h_owners:Joe[owns := { }] ← h_owners:Joe[name = "Joe T."] ;
```

In the above cases, (b), (c), (d), (e), (f), (h), (i) and (j) are primitive updates, while (a) and (g) are composite updates. All cases except (a) are also potential set-oriented updates since the update conditions may dictate that the primitive update be applied to more than one object.

5.2. *Execution semantics for update rules*

In general, an update rule represents a set of uniform composite updates, where each composite update is uniquely applicable to a single distinct object of the same class. Specifically, let $r_u = U \leftarrow L_1, \ldots, L_n$ be an update rule, where U is the composite update term $t[A_{i_1} := u_1, \ldots, A_{i_k} := u_k]$. To establish the qualifying composite updates for r_u, first the

query $? - L_1, \ldots, L_n$ is evaluated against the database instance, and then the bindings for variables that do not appear in U are projected out. The evaluation of r_u essentially produces a $k + 1$-ary *answer* relation with schema $R_u(O_{id}, A_{i_1}, \ldots, A_{i_k})$. R_u is a nested relation if there are multivalued attributes in $A_{i_1}, \ldots, A_{i_k}$. The O_{id} attribute of R_u plays the role of the relation key, essentially specifying which objects are to be updated, while the other attributes carry the updating action for each updated property of the qualifying objects. For single-valued attributes, the update value replaces the old value. For set-valued attributes, primitive property-insert update terms amount to set union with the old value while primitive property-delete update terms translate to set difference. Composite set-valued update terms replace the old set value.

We say that update rule r_u is *admissible* against the database instance if the functional dependency $O_{id} \rightarrow (A_{i_1}, \ldots, A_{i_k})$ holds for each tuple in R_u. If r_u is an admissible update rule, each tuple of R_u represents a deterministic composite update of a different database object, due to the aforementioned functional dependencies.

All of the update rules presented in Section 5.1 are admissible. The following update rule is not admissible:

```
horses:H[trainer := T] ← horses:H[name = "Secretariat"], h_trainers:T ;
```

Since h_trainers is a set of objects, this rule defines multiple updates of the trainer property of H. The update is nondeterministic – the final database instance depends on the T value that is accessed last.

An admissible update rule r_u represents s composite updates, where $s = |R_u|$. If R_u is a singleton, then r_u represents a single composite update. If $R_u = \{\}$, then r_u has no effect on the database instance. Finally, if r_u is not an admissible update rule, then it is rejected (aborted) since it would cause nondeterministic update effects.

5.3. Methods and transactions

The previous subsections provided examples of update rules within CDOL. However, since CDOL supports an object-oriented model of data, modifications to objects must be encapsulated within the methods associated with a class. Updates are then performed by calling the appropriate class methods. CDOL update rules therefore provide a means for constructing the update methods associated with a class.

Methods are actually transactions that encapsulate the behavior of a class. Transactions can also be defined as independent procedures that are not associated with any specific class. In this case, transactions represent application-oriented procedures that make calls to class methods and/or other application-oriented transactions. CDOL also supports calls to externally-defined functions and procedures that may be written in a language such as C or C++. Such procedures may be necessary for complex calculations or for the use of input/output statements.

As indicated in the CDOL overview in Section 2, classes have several default methods that are automatically generated as a result of the schema definition. These default methods support object creation and deletion, update operations on single and set-valued properties, and insert/delete operations on set-valued properties.

A constructor method for a class has the name: create_*class_name*. Constructor methods, by default, do not have any parameters and therefore the properties of the treated object are automatically assigned default values. For example, the predefined constructor method for the persons class is:

```
method persons::create_persons( )
{
    persons:new P[name :="", ssn :="", state :=""] ;
} ;
```

This method can be redefined by the user to the following method that creates an object and assigns specific values to its properties:

```
method persons::create_persons(string:Name, string:SSN, string:State)
{
    persons:new P[name := Name, ssn := SSN, state := State] ;
} ;
```

A destructor method has the name: ~*class-name*. For example the destructor for the races class is method races::~races( ) and its implementation is:

```
method races::~races( )
{
    races: destroy This ;
} ;
```

The variable This in the ~races destructor is a system supported variable that is used to refer to the object that receives the message. Given the above definition of the races destructor, assume that a user transaction needs to delete all races that are schedules for 1/24/96 and have a purse less than $10,000. The following rule would be included in the transaction to delete the appropriate objects:

```
races:Race[~races( )] ←
  race_schedules:RS[race_date.month = "Jan", race_date.day = 24, race_date.year = 96,
  has_races = {Race}], Race[purse = Purse], Purse < 10000 ;
```

This approach to referencing class methods respects the encapsulation concept of object-orientation and also provides a uniform environment to access both properties and methods.

An update method has the name: update_*property_name*, for updating single or set-valued properties. As an example, the following method is automatically generated for the persons class:

```
method void persons::update_state(string:State)
{
    persons:This[state := State] ;
} ;
```

The update_state method can be referenced in the body of a user transaction or another class method as follows:

```
persons:Joe[update_state("Nevada")] ← persons:Joe[ssn = "123456789"] ;
```

When an update method is generated for a set-valued property, the parameter of the method is a set value that replaces the current value of the set-valued property. The names for insert and delete operations on set-valued properties are insert_*property_name* and delete_*property_name*, respectively. In addition, since the granularity of insertion or deletion can be a single element or a set of elements, these operations are overloaded to handle either granularity. As an example, the following four operations are defined for the relationship owns:

- **Inserting**
  - *Inserting single elements*

```
method void h_owners::insert_owns(horses:Horse)
{
  h_owners:This[owns := {+Horse}] ;
} ;
```

  - *Inserting a set of elements*

```
method void h_owners::insert_owns(set⟨horses⟩:Horse_set)
{
  h_owners:This[owns := Old_set union Horse_set] ← This.owns = Old_set ;
} ;
```

- **Deleting**
  - *Deleting a single element*

```
method void h_owners::delete_owns(horse:Horse)
{
  h_owners:This[owns := {−Horse}] ;
} ;
```

  - *Deleting a set of elements*

```
method void h_owners::delete_owns(set⟨horses⟩:Horse_set)
{
  h_owners:This[owns := Old_set diff Horse_set] ← This.owns = Old_set ;
} ;
```

Methods are inherited by subclasses. As a result, all of the methods defined for the persons class are also valid for the h_owners class. The signatures for default methods do not have to be defined in the class definitions of a CDOL schema; only the signatures of user-defined methods appear in the class definition. However, a subclass can override the definition of any inherited default method, in which case the definition of the signature for the overridden

method appears in the class definition to signify that the method has a new definition at the subclass level.

As an example of a user-defined method, consider the horses class, which provides a method signature for transferring ownership of horses:

```
void transfer(h_owners:New_Owner);
```

This method involves deleting the horse from the previous owner's owns property, inserting the horse into the new owner's owns property, and setting the horse's owned_by property to the new owner.

```
method void horses::transfer(h_owners:New_Owner)
{
    h_owners:Old_Owner[delete_owns(This)] ← horses:This[owned_by = Old_Owner];
    h_owners:New_Owner[insert_owns(This)];
    horses:This[owned_by := New_Owner];
};
```

In the above method, the third statement that updates the horses owned_by property is the only statement that can directly use a primitive update operation; since the method is defined for the horses class, the horse object that receives the message can directly modify itself with primitive updates. The first two statements, however, involve modification of the old and new horse owner which requires changes to objects of the h_owners class. As a result, modifications of h_owners objects can only be made through the predefined methods associated with the h_owners class.

The above method can also be specified as a transaction that is not related to any specific class:

```
transaction void transfer(horses:Horse, h_owners:New_Owner)
{
    h_owners:Old_Owner[delete_owns(Horse)] ← horses:Horse[owned_by = Old_Owner];
    h_owners:New_Owner[insert_owns(Horse)];
    horses:Horse[update_owned_by(New_Owner)];
};
```

In this case, all three statements are required to express updates through the methods associated with the classes involved. Since this update also involves the maintenance of the inverse relationship between the owns and owned_by properties, a third alternative for transfer of ownership is to simply modify the owned_by attribute (or the owns attribute) and make use of the inverse constraints in Section 4 together with active rules to trigger the appropriate updates. The following section addresses the use of active rules in CDOL applications for the maintenance of constraints, specifically describing the integrity maintenance approach for the transfer of horse ownership example. Section 6 also addresses the specification of reactive behavior, in general.

6. The active rule sublanguage of CDOL

The final sublanguage to be presented as part of CDOL is the active rule sublanguage. Active rules provide a means of automatically initiating repair actions for constraint violations. Active rules can therefore be associated with constraints as described in Section 4, where each constraint may have several active rules. The different rules associated with each constraint may depend on the specific operation violating that constraint as described in the work in [51]. Active rules, in general, can also be used to monitor specific conditions to serve as alerters of special circumstances or triggers of prespecified operations. Active rules thus provide CDOL applications with reactive behavior, responding automatically to situations that would not be allowed in traditional passive systems, or triggering operations that would otherwise have to be manually initiated by users. Active capabilities are becoming increasingly important for advanced database applications.

In the following subsections, we present the CDOL active rule sublanguage together with an informal description of the operational semantics of the rule language. As with the previous sublanguages, the active rule language builds on the query language of CDOL. In addition, active rules make use of the update sublanguage as well as methods and transactions for the specification of the actions to be triggered.

6.1. Active rules in CDOL

The general form of a CDOL active rule is

active *arname* in *rsname*
{
event E_1 or E_2 or ... or E_n
condition *event_condition_coupling_mode* $L_1, L_2, \ldots, L_m$
action *condition_action_coupling_mode* $A_1 A_2 \ldots A_k$
$\rangle$ *rules_and_rulesets_list*
$\langle$ *rules_and_rulesets_list*
};

In the above rule format, *arname* is the name of the active rule and *rsname* is the optional rule set name. Rule sets allow CDOL rules to be grouped into logical sets to support rule management activities in an active database environment, such as activation and deactivation of rules to support rule tracing and rule debugging [34]. Rule sets are formed by grouping together rules that target the same database information or that handle similar events.

A CDOL active rule is interpreted as follows. Upon triggering an active rule by executing one of the specified events E_i in the event clause, the query $L_1, L_2, \ldots, L_m$ in the condition clause is evaluated. The evaluation returns *true* if there is at least one set of variable bindings that is an answer to the query. In this case we say that the condition part is *satisfied*. If the condition is satisfied, the action part is executed separately for each answer variable binding.

Each E_i in the event clause is the name of a CDOL method or transaction. Each A_i in the action part is a CDOL update rule.

If the evaluation of the condition part returns an empty answer, we say that the condition is *not satisfied* or *false*. If the condition is false, the update terms in the action part of the rule are not executed, and the active rule terminates without making any updates. In the special case when the action is *abort*, the triggering event is aborted. Note that the condition of a rule is optional, thus supporting event-action rules. We plan to eventually support condition-action rules by making the event optional and using condition monitoring techniques such as that described in [29,30] as an additional technique for triggering rules.

In the event specification, each E_i is of the form: before|after *event_name*, where the before and after directives are used to specify the time the rule should be triggered with respect to the occurrence of the event. Specifically, a before directive indicates that the condition and action of the rule should be evaluated before the execution of the event. Likewise, an after directive indicates that the condition of the rule is executed after the execution of the event. The after directive is assumed as a default.

CDOL also provides for the use of three different *coupling modes* between event and condition and between condition and action as originally defined in [22]: immediate, deferred and decoupled. Coupling modes indicate when a condition is executed relative to the event, or when the action is executed relative to the execution of the condition. Using event-condition coupling as an example, a mode of immediate means that the condition is executed immediately after the occurrence of the event. deferred indicates that execution of the condition is deferred until the commit of the transaction in which the event occurred. A decoupled mode defines that the condition will be executed in a separate transaction than the transaction containing the event. In the case of immediate and deferred coupling modes, we assume the use of a nested transaction model in which immediate and deferred parts of rules are executed as subtransactions within the parent transaction that triggered the rule. Further details about the semantics of coupling modes appears in the following subsection addressing the semantics of the execution model.

In addition to coupling modes, CDOL supports the partial ordering of rules through the use of the $\rangle$ and $\langle$ specifiers. These specifiers can be used to resolve rule conflicts when more than one rule is scheduled to execute at the same time. For example, the list of rules and rule sets associated with a $\rangle$ specifier in rule R indicates the rules that should execute before the execution of rule R.

As an example of CDOL active rules, consider again the inverse constraint between the owns and owned_by relationships in Fig. 1:

```
constraint owner_owns_1
  if h_owners:Owner[owns = {Horse}]
  then horses:Horse[owned_by = Owner];

constraint owner_owns_2
  if horses:Horse[owned_by = Owner]
  then exists H in Owner.owns (H = Horse);
```

Four separate active rules can be constructed to automatically maintain the inverse relationship. Insert and delete atomic updates on the owns or the owned_by properties are the triggering events that violate the above constraints, leading to the following five active rules:

```
active owner_rule_1     // assign the horse a new owner – previous owner exists
{
    event       before horses:Horse[update_owned_by(New_Owner)]
    condition   immediate h_owners:New_Owner, New_Owner <> nil,
                  horses:Horse[owned_by = Old_Owner], New_Owner <> Old_Owner
    action      immediate h_owners:Old_Owner[delete_owns(Horse)] ;
                  h_owners:New_Owner[insert_owns(Horse)] ;
} ;

active owner_rule_2     // delete the horse owner
{
    event       after horses:Horse[update_owned_by(Owner)]
    condition   immediate h_owners:Owner, Owner = nil,
                  not (h_owners:H_Owner, Horse in H_Owner.owns)
                h_owners:Old_Owner[owns = Horses_Owned], Horse in Horses_Owned
    action      immediate h_owners:Old_Owner[delete_owns(Horse)] ;
} ;

active owner_rule_3     // Assign the horse a new owner – no previous owner
{
    event       after horses:Horse[update_owned_by(Owner)]
    condition   immediate h_owners:Owner, Owner <> nil,
                h_owners:Old_Owner[owns = Horses_Owned], Horse in Horses_Owned
    action      immediate h_owners:Owner[insert_owns(Horse)] ;
} ;

active owns_rule_1      // include the horse in the set of owned horses
{
    event       after h_owners:Owner[insert_owns(Horse)]
    condition   immediate horses:Horse[owned_by = Horse_Owner], Owner <> Horse_Owner
    action      immediate horses:Horse[update_owned_by(Owner)] ;
} ;

active owns_rule_2      // remove ownership of the horse
{
```

```
    event       after h_owners:Owner[delete_owns(Horse)]
    condition   immediate horse:Horse[owned_by = Horse_Owner], Owner = Horse_Owner
    action      immediate horses:Horse[update_owned_by(nil)] ;
} ;
```

As an example of an active rule for a more complex constraint, consider the following constraint which states that a horse must be trained by a trainer with a salary of at least $75,000 if the price of the horse is greater than $250,000:

```
constraint trainer_salary
  if horses:H[price = P], P > 250000
  then H.salary_of_trainer > 75000 ;
```

The above constraint can be used to define the following CDOL rule which is triggered when the price of a horse changes:

```
active trainer_salary
{
    event       after horses:Horse[update_price(New_Price)]
    condition   deferred New_Price > 250000, Horse.salary_of_trainer < 75000
    action      immediate horses:Horse[update_trainer(New_Trainer)] ←
                  New_Trainer = get_trainer_over_75K( ) ;
} ;
```

In the above rule, a change in the price of a horse will cause the condition of the rule to be executed at the end of the transaction in which the price change occurs. If the current trainer of the horse does not satisfy the condition, the action will execute a rule that calls the transaction get_trainer_over_75K. This transaction will execute external functions that interact with the user to identify a trainer with the appropriate salary. The result returned from the transaction in New_Trainer is then used to update the trainer attribute value of the horse.

As a final rule example, the following rule monitors the destructor method of the h_owners class. Upon occurrence of the event, all horses owned by the deleted owner, along with their race entries, are deleted from the database.

```
active purge_horse_owner_information
{
    event   before h_owners:Owner[~h_owners( )]
    action  immediate
              entries:Entry[~entries( )] ←
                h_owners:Owner[owns = {Horse}], horses:Horse[race_record = {Entry}] ;
              horses:Horse[~horses( )] ←
                h_owners:Owner[owns = {Horse}] ;
} ;
```

Table 1
E-C-A coupling modes

Event-condition coupling mode	Condition-action coupling mode		
	Immediate	Deferred	Decoupled
Immediate	ok	ok	ok
Deferred	ok	ok	ok
Decoupled	ok	no	ok

6.2. *Execution model issues for CDOL active rules*

The semantics of rule execution in active database systems are complicated by the variety of alternatives for rule execution behavior [47]. In CDOL, we have defined three specific aspects of the execution model: coupling modes, binding patterns, and rule processing granularity. More specifically, the event-condition and condition-action coupling modes as defined in the previous subsection can only be used together in certain combinations, especially considering the before and after directives of the event specification. Binding patterns address the data that are transferred between the event and condition and between the condition and action, thus defining the valid values that can be referenced in the specification of rule conditions and actions. Rule processing granularity defines the instance-oriented versus set-oriented nature of the rule execution process. In the following subsections, we address each of these issues in further detail with respect to CDOL active rules.

6.2.1. *Coupling modes*

Although three different coupling modes are supported in CDOL, certain combinations of coupling modes are not valid. The specification of coupling modes is also affected by the before and after specifications for events. Table 1 presents a matrix showing the valid combinations of event-condition and condition-action coupling modes. As indicated in Table 1, the only combination that is not allowed is a decoupled condition with a deferred action. A decoupled condition implies that the condition will be executed in a separate transaction. This separate transaction will only contain the condition of the rule. As a result, if the action is not decoupled, it will always, by default, be executed immediately following the execution of the condition. A deferred coupling mode for the action in this case is meaningless.

Coupling mode specifications must also be coordinated with the specification of the rule event. Table 2 summarizes the coupling modes that are feasible in conjunction with the time of the event occurrence. If the event includes a before specification, then the condition/action

Table 2
Rule triggering with respect to event occurrences

Event-condition/event-action coupling mode	Before	After
Immediate	ok	ok
Deferred	no	ok
Decoupled	ok	ok

cannot be deferred. Otherwise a deadlock situation is created: the condition must be executed at the end of the transaction but the end of the transaction will never be reached since the event will not execute until after execution of the condition. Note that this restriction applies to event-condition and event-action coupling modes. Condition-action coupling modes under these conditions follow the same restrictions as defined in Table 1.

6.2.2. Binding patterns

It is important in rule specification and in rule execution to define the information, if any, that flows from one part of a rule to another. In CDOL, any parameters of the event can be referenced in the condition or the action of the rule. Objects bound by variables at the time of the condition evaluation are also passed to the action, as long as the variables represent atomic values or set objects. Variables that range over the instances of a set (i.e., variables that are introduced in set braces) are not passed from the condition to the action. Passing bindings through such variables necessitates the introduction of grouping into the language and thus complicates the expression and semantics of rules actions. Note that Chimera [18] restricts parameter passing to atomic values only. CDOL is more flexible in that set objects can be passed between the different parts of an active rule.

When the rule condition is executed, the result of condition evaluation may produce several solutions. The bindings that are produced as a result of condition evaluation can be viewed as a set of tuples that correspond to the bindings that are eligible to be referenced in the action, where each tuple represents a different solution to the condition evaluation. There is an implied iteration feature in the action of each rule in that the action will be executed for every tuple of bindings that is produced by the condition.

6.2.3. Rule processing granularity

As defined in [47], an important rule processing characteristic to define is whether rules are processed in an instance-oriented and/or set-oriented fashion. Traditionally, most active, relational database systems process rules in a set-oriented mode, although Postgres is an exception [50]. Most active, object-oriented systems use an instance-oriented approach to the execution of rules. In instance-oriented rule processing, a rule is executed once for each database instance triggering the rule. In set-oriented rule processing, a rule is executed once for a set of objects that have triggered the rule. There are two important distinctions here between instance-oriented and set-oriented rule processing. First of all, in an instance-oriented approach, the rule condition will potentially be evaluated on a different database state for each event that triggers the rule. In a set-oriented approach, the rule condition will be evaluated on the same database state for all events that have triggered the rule. Furthermore, in a set-oriented approach, the cumulative results of events are considered. For example, a creation of an object followed be a deletion of an object is ignored, as if the object was never created. As a result, a rule with a deferred condition that is triggered by the creation of an object that is deleted before the end of the transaction would have no effect.

CDOL active rules support both modes of processing. A rule condition with an immediate coupling mode implies an instance-oriented execution of the rule in that the rule condition is to be immediately executed on the parameters of the event that triggered the rule. A rule

condition with a deferred coupling mode is automatically executed in a set-oriented fashion. In this case, the rule is scheduled to execute at the end of the transaction. The first event that triggered the rule, together with all subsequent events that trigger the same rule, are then collected together for processing at the end of the transaction. At the end of the transaction, the rule condition is then collectively executed over the same database state for all events associated with the rule. If the condition-action coupling is immediate, then the actions are immediately executed for each event (after the condition has been evaluated for all events). If the condition-action coupling is deferred, then the action for each event is deferred for execution until the end of the current rule execution waiting list.

7. Related work

As a model and language that integrates deductive, object-oriented, and active database concepts, CDOL builds on past work in all three of these areas. This section reviews the work that has influenced the design of CDOL. In particular, we review work in the area of DOOD (Deductive, Object-Oriented Databases) systems and active database systems.

7.1. Research on deductive, object-oriented database systems

The CDOL basic data model and rule language are similar to those of models such as IQL [3], LLO [43], C-Logic [21], and F-Logic [40], representing the initial work in defining the integration of deductive and object-oriented concepts. In all of these DOOD systems, a clear separation is made between object identity and state. Tuple and set type constructors are used to build composite types, and the class lattice is basically a specialization hierarchy. Our presentation of virtual attributes and virtual classes was fully motivated by [2], where SQL-like declarations are used for definition of object views. We believe that the research in [2] provides the framework for the definition of a customizable database environment for end-users, while maintaining a reasonably simple stored object database. More recent examples of DOOD systems are found in research prototypes such as Rock&Roll [7] and Logres [13]. The system most closely related to CDOL is the Chimera system [18], which also integrates active rule processing into a DOOD environment as in CDOL. Chimera is addressed in more detail in the following subsection on active database research.

One difference between CDOL and other DOOD systems is that constraints are not extensively addressed in the context of most DOOD's [15,46], although researchers acknowledge their importance as a separate concept [9]. One possible reason for the lack of attention to constraints is the tight connection between constraints and updates, which are also infrequently addressed.

The problem of updating relational, deductive and object-oriented databases has been addressed from a variety of aspects. From a *practical* point of view, almost all database products have an elementary update interface, allowing more complex update transactions to be developed as procedural programs. This approach has two major drawbacks: first, update transactions have to be anticipated and programmed in advance, hence limiting the ad-hoc update capability; and second, database application semantics is buried into procedural code.

From a *theoretical* point of view, database updates have been addressed primarily for deductive or logical databases [55]. While the logical foundations of relational and deductive databases are well understood, the update aspect has not been easy to formalize, since first-order logic does not capture the notion of a dynamically changing world [20]. As a consequence, dynamic logic has been used to formally describe database updates.

Declarative languages for updating logical databases have been developed by [4,20,44,45]. Abiteboul et al. [4] concentrate on the expressive power of the declarative update languages or their procedural counterparts. They investigate the issue of nondeterministic updates and isolate sublanguages that possess the desirable deterministic update property. Manchanda et al. [44] enhance a Datalog language with update rules, using dynamic logic for the underlying theory that captures the update transitions between database instances. This framework can also be used to map view updates onto base database updates. Naqvi et al. [45] describe a similar update language, but with more control constructs, such as sequencing, looping and if-then. Again, updates are viewed as describing the accessibility relation over database instances. The work of Chen [20] considers database updates as binary relations over partial interpretations of a database. By using this approach Chen develops an update algebra and an update calculus, and shows their equivalence.

7.2. Research on active database systems

Major research efforts that have focused on the development of active database prototypes include HiPAC [22], POSTGRES [50], Starburst [27], Alert [49], Ode [26], Sentinel [6], and Ariel [28], as well as many other recent systems described in [48,54]. An excellent overview of current work in active database systems can be found in [53].

HiPAC was one of the first active systems to outline the details of rule execution semantics using the notion of coupling modes [22]. Subsequent projects in active database systems have used execution models that are variations of that originally described in the HiPAC paper [22]. Gehani's work with Ode, for example, provides separate execution specifications for general rules and constraint rules [26]. Gehani makes a distinction between hard constraints and soft constraints, which conforms to the notion of immediate and deferred coupling modes in HiPAC. Recently, Beeri presented a logical model for active rule execution that subsumes most of the execution models presented to date and refines the notion of an execution interval with the specification of starting and ending events [11]. The work in [32] also presents a language for describing the semantics of rule execution. Paton's work provides an overview of the main issues that must be considered in the definition of active rule execution models [47].

An advantage of active database systems is that they provide an effective means of dealing with the maintenance of integrity constraints. The work in [52] for example describes how to automatically derive active integrity-preserving rules from general constraints. In [19] a framework for integrity maintenance of a relational database is developed using the Starburst active database system, where rules are generated from constraints. More recently, the Chimera system [18] provides an example of an active DOOD system supporting constraints and an active rule generation system for generating rules from constraints. Chimera also supports tools to analyze the behavior of rules, both at compile time and at run time. Many of the objectives of the Chimera system are similar to our own objectives in the use of CDOL as

the basis for the construction of an environment to support the analysis, testing and debugging of active rules. Our work differs from Chimera in the support that we provide for the rule-based update language and use of the update language in the definition of methods and transactions. The bindings that we provide within active rules are also more flexible since CDOL supports bindings for set-valued objects. In addition, our work also diverges from Chimera in the incorporation of condition monitoring into the CDOL execution model (an issue that is not addressed in this paper) as well as our focus on architectural issues for more effective construction of active systems for run-time rule analysis, with specific emphasis on support for rule testing. These are issues that represent our future directions using the CDOL language as described in this paper.

8. Summary and future research

This paper has presented CDOL, a comprehensive language for managing objects using a declarative approach. An important aspect of CDOL is the uniform manner in which the rule-based query language provides a basis for expressing derived data, constraints and updates. We have also presented an active approach to constraint enforcement in CDOL. By introducing active rules on top of a standard structural object-oriented database and a declarative query language, we have developed a more sophisticated object-oriented database environment that supports a high-level declarative update language and provides integrity enforcement transparently to the end user. The development of active rules from constraints is a one-time process that can be supported by automated tools [52]. An important contribution of CDOL is the framework it has established for studying the interaction between constraints, derived data, and declarative updates using active database concepts.

There are several outstanding issues regarding the language definition that still need to be addressed. One issue is related to the use of update rules in transactions. There is a potential need in transaction specification to pass information from one update rule to another. Currently, variable bindings only exist within the scope of an individual rule, except for those variables that are: (1) passed as parameters in transactions, or (2) passed between the event, condition and action components of active rules. We still need to examine how the bindings of variables from one update rule can be passed to subsequent update rules to provide additional variables that are global to the scope of an entire transaction. A second issue is related to the introduction of relations into CDOL. There is an advantage to supporting both object-based and valued-based approaches within a database environment, as already addressed in languages such as [42]. The introduction of relations into CDOL will provide an expressive alternative for the specification of complex intensional data such as queries and conditions. Relations are particularly important in the definition of queries that do not necessarily return objects as results. Resolution of these outstanding issues will contribute to the practical use of CDOL as an active database language for our future research efforts.

Within the active database framework, we are experimenting with the execution model for active rule processing described in this paper together with an approach for analyzing active rule behavior. As active rules become an integral art of future database systems, it is important to provide tools to assist in understanding and debugging the use of active rules. In

particular, it is important to determine whether a set of active rules possess desirable properties such as termination and confluence [5]. Our initial results on the analysis of active rules are reported in [37,38]. We are also using CDOL as the basis for the development of an environment that supports runtime debugging of active rules, with specific emphasis on testing methodologies. Our initial results on active rule debugging are reported in [34].

Acknowledgments

We would like to thank Amy Sundermier, Michael Tschudi, Lorena Gomez and Michael Tjahjadi of the A-DOOD (Active, Deductive, Object-Oriented Databases) research group for their careful review of the paper. We would also like to thank the referees for their helpful comments on an earlier version of this paper.

A. The schema definition of the horse racing database application

This appendix presents the application definition of the Horse Racing Database (HRDB) application. The listing is not an exhaustive definition of the HRDB application, but is intended to illustrate the various components of a CDOL schema. Implicit components of the schema, such as default method definitions and inherent constraints within the schema definition (e.g. key and inverse relationship constraints) are not included here as they will be automatically generated by the system without the need of explicit specification in the application definition.

A.1. Type definition

```
typedef struct date_struct {
    short day ;
    string month ;
    short year ;
} date_type ;
```

A.2. Class definition

```
class persons
(
    extent persons
    key ssn)
{
    attribute string ssn ;
```

```
    attribute string name ;
    attribute string state ;
} ;

class h_owners: persons
(
    extent h_owners)
{
    relationship set ⟨horses⟩ owns inverse horses::owned_by ;
} ;

class h_trainers:persons
(
    extent h_trainers)
{
    attribute short salary ;
    relationship set ⟨horses⟩ trains inverse horses::trainer ;
} ;

class jockeys: persons
(
    extent jockeys)
{
    attribute short weight ;
    relationship set ⟨entries⟩ raced_in inverse entries::jockey ;
} ;

class hots: h_owners, h_trainers
(
    extent hots)
{
} ;

class horses
(
    extent horses
    key (name, type))
```

```
{
    attribute string name ;
    attribute string type ;
    attribute string sex ;
    attribute short price ;
    relationship horses sire inverse horses::sire_of ;
    relationship horses dam inverse horses::dam_of ;
    relationship set ⟨horses⟩ sire_of inverse horses:sire ;
    relationship set ⟨horses⟩ dam_of inverse horses::dam ;
    relationship h_owners owned_by inverse h_owners::owns ;
    relationship h_trainers trainer inverse h_trainer::trains ;
    relationship set ⟨entries⟩ race_record inverse entries::horse ;

    virtual short salary_of_trainer = trainer_salary_rule ;
    virtual set ⟨horses⟩ ancestors = horse_ancestor_rule ;
    virtual short total_race_count = total_race_count_rule ;
    virtual short average_win_amount = average_win_amount_rule ;

    void transfer(h_owners:New_Owner) ;
    void assign_trainer(h_trainers:New_Trainer) ;
} ;

class race_schedules
(
    extent race_schedules
    key (site, race_date))
{
    attribute string site ;
    attribute data_type race_date ;
    relationship set ⟨races⟩ has_races inverse races::scheduled_in ;

    virtual set ⟨races⟩ purse_over(short:Low) = purse_over_rule ;
    virtual set ⟨races⟩ purse_between(short:Low, short:High) = purse_between_rule ;
    virtual short jockey_count = jockey_count_rule ;
} ;
```

```
class races
(
   extent races
   key race_no)
{
   attribute string race_no ;
   attribute short purse ;
   relationship race_schedules scheduled_in inverse race_schedules::has_races ;
   relationship set ⟨entries⟩ has_entries inverse entries::race ;
} ;

class entries
(
   extent entries
   key (horse, jockey, entry_date))
{
   attribute string gate ;
   attribute short finish_position ;
   attribute short amount_won ;
   attribute date_type entry_date ;
   relationship horses horse inverse horses::race_record ;
   relationship jockeys jockey inverse jockeys::raced_in ;
   relationship races race inverse races::has_entries ;
} ;
```

A.3. Virtual class definition

```
vclass expensive_horses = expensive_rule ;
vclass thoroughbreds = thoroughbreds_rule ;
vclass male_thoroughbreds = male_tb_rule ;
vclass champions = champ_rule ;
vclass horses_trained_by(h_trainers:The_Trainer) = trained_by_rule ;
vclass little_ranch_horses = little_range_rule ;
vclass ever_win_jockeys = ever_win_rule ;
vclass never_lose_horses = never_lose_rule ;
```

A.4. Constraint definition

```
constraint number_of_races
  if    race_schedules:Schedule[races = Race]
  then  count(Race) = 10 ;

constraint quarterhorses
  if    races:Race[race_no = 1, scheduled_in = Sched],
          Sched[race_date.day = 16, race_date.month = "May", race_date.year = 1996]
  then  for_all Entry in Race.has_entries(Entry.horse.type = "quarterhorse") ;
```

A.5. Passive rule definition

A.5.1. Rules for derived attributes

```
rule trainer_salary_rule
{
   horses:Horse[salary_of_trainer = Salary] ←
     horses:Horse, Horse.trainer.salary = Salary ;
} ;

rule horse_ancestor_rule
}
   horses:Horse[ancestors = {Ancestor}] ←horses:Horse, Horse.dam = Ancestor ;
   horses:Horse[ancestors = {Ancestor}] ←horses:Horse, Horse.sire = Ancestor ;
   horses:Horse[ancestors = {Ancestor}] ←horses:Horse, Horse.ancestors = {Y} ,
     Y.ancestors = {Ancestor} ;
} ;

rule purse_over_rule
{
   race_schedules:Schedule[purse_over(short:Low) = {Y}] ←
     race_schedules:Schedule, Schedule.has_races = {Y}, Y.purse >= Low ;
} ;

rule purse_between_rule
{
   race_schedules:Schedule[purse_between(short:Low, short:High) = {Y}] ←
     race_schedules:Schedule, Schedule.has_races = {Y},
```

```
        Y.purse  >= Low, Y.purse <= High ;
    } ;

    rule total_race_count_rule
    {
        horses:Horse[total_race_count = Count] ←
            horses:Horse, Count = count(Horse.race_record) ;
    } ;

    rule average_win_amount_rule
    {
        horses:Horse[average_win_amount = Amt] ←
            horses:Horse, Amt = avg(Entry.amount_won over Entry in Horse.race_racord) ;
    } ;

    rule jockey_count_rule
    {
        race_schedules:Sched[jockey_count = J_Cnt] ←race_schedules:Sched,
            J_Cnt = count({Entry.jockey} over Race in Sched.has_races,
            Entry in Race.has_entries) ;
    } ;
```

A.5.2. Rules for derived classes

```
    rule expensive_rule
    {
        expensive_horses:Exp_Horse ←
            horses:Exp_Horse[price = The_Price], The_Price > 1000000 ;
    } ;

    rule thoroughbreds_rule
    {
        thoroughbreds:Tbreds ←horses:Tbreds[type = "thoroughbred"] ;
    } ;

    rule male_tb_rule
    {
        male_thoroughbreds:MTbreds ←thoroughbreds:MTbreds[sex = "male"] ;
```

```
};

rule champ_rule
{
    champions:Champ ←
      horses:Champ[race_record = {Race_Rec}],
      Race_Rec[finish_position = 1, amount_won = Amt_Won],
      Amt_Won >= 100000 ;
};

rule trained_by_rule
{
    horses_trained_by(h_trainers: The_Trainer): Horse ←
      horses:Horse[trainer = The_Trainer] ;
};

rule little_ranch_rule
{
    little_ranch_horses:LRHorses ←
      h_trainers:Trainer_Jim[name = "Jim"],
      horses_trained_by(Trainer_Jim):LRHorses ;
    little_ranch_horses:LRHorses ←
      h_trainers:Trainer_John[name = "John"],
      horses_trained_by(Trainer_John):LRHorses ;
};

rule ever_win_rule
{
    ever_win_jockeys:Jockey ← jockeys:Jockey, exists Entry in Jockey.raced_in
      (Entry.finish_position >= 1, Entry.finish_position <= 3) ;
};

rule never_lose_rule
{
never_lose_horses:Horse ←
      horses:Horse, for_all Entry in Horse.race_record
      (Entry.finish_position >= 1, Entry.finish_position <= 3) ;
};
```

A.6. Method and transaction definition

```
method void horses::transfer(h_owners:New_Owner)
{
    h_owners:Old_Owner[delete_owns(This)] ←horses:This[owned_by = Old_Owner] ;
    h_owners:New_Owner[insert_owns(This)] ;
    horses:This[owned_by := New_Owner] ;
} ;

method void horses::assign_trainer(h_trainers:New_Trainer)
{
    h_trainers:Old_Trainer[delete_trains(This)] ←horses:This[trainer = Old_Trainer] ;
    h_trainers:New_Trainer[insert_trains(This)] ;
    horses:This[trainer := New_Trainer] ;
} ;

transaction void transfer(horses:Horse, h_owners:New_Owner)
{
    h_owners:Old_Owner[delete_owns(Horse)]
        ←horses:Horse[owned_by = Old_Owner] ;
    h_owners:New_Owner[insert_owns(Horse)] ;
    horses:Horse[update_owned_by(New_Owner)] ;
} ;
```

A.7. Active rule definition

```
active owner_rule_1
{
    event       before horses:Horse[update_owned_by(New_Owner)]
    condition   immediate h_owners:New_Owner, New_Owner <> nil,
                  horses:Horse[owned_by = Old_Owner], New_Owner <> Old_Owner
    action      immediate h_owners:Old_Owner[delete_owns(Horse)] ;
                  h_owners:New_Owner[insert_owns(Horse)] ;
} ;

active owner_rule√3
{
```

```
    event       after horses:Horse[update_owned_by(Owner)]
    condition   immediate h_owners:Owners, Owner = nil,
                  h_owners:Old_Owner[owns = Horses_Owned],
                  Horse in Horses_Owned
    action      immediate h_owners:Old_Owner[delete_owns(Horse)] ;
} ;

active owner_rule_2
{
    event       after horses:Horse[update_owned_by(Owner)]
    condition   immediate h_owners:Owner, Owner <> nil,
                  not (h_owners:H_Owner, Horse in H_Owner.owns)
    action      immediate h_owners:Owner[insert_owns(Horse)] ;
} ;

active owns_rule_1
{
    event       after h_owners:Owner[insert_owns(Horse)]
    condition   immediate horses:Horse[owned_by = Horse_Owner],
                  Owner <> Horse_Owner
    action      immediate horses:Horse[update_owned_by(Owner)] ;
} ;

active owns_rule_2
{
    event       after h_owners:Owner[delete_owns(Horse)]
    condition   immediate horses:Horse[owned_by = Horse_Owner],
                  Owner = Horse_Owner
    action      immediate horses:Horse[update_owned_by(nil)] ;
} ;

active trainer_salary
{
    event       after horses:Horse[update_price(New_Price)]
    condition   deferred New_Price > 250000, Horse.salary_of_trainer < 75000
```

```
    action      immediate horses:Horse[update_trainer(New_Trainer)] ←
                  New_Trainer = get_trainer_over_75k( ) ;
} ;

active purge_horse_owner_information
{
    event       before h_owners:Owner[~h_owners( )]
    action      immediate
                  entries:Entry[~entries( )] ←h_owners:Owner[owns = {Horse}],
                    horses:Horse[race_record = {Entry}] ;
                  horses:Horse[~horses( )] ←h_owners:Owner[owns = {Horse}] ;
} ;
```

References

[1] S. Abiteboul, Towards a Deductive Object-Oriented Database Language, in [41] 453–472.

[2] S. Abiteboul and A. Bonner, Objects and views, in: *Proc. 1991 ACM SIGMOD Conf.* (1991) 238–247.

[3] S. Abiteboul and P.S. Kanellakis, Object identity as a query language primitive, in: *Proc. 1989 ACM SIGMOD Conf.* (1989) 159–173.

[4] S. Abiteboul and V. Vianu, Procedural and declarative update languages, in: *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1988) 240–250.

[5] A. Aiken, J. Widom and J.M. Hellerstein, Behavior of database production rules: termination, confluence and observable determinism, in: *Proc. 1992 ACM SIGMOD Conf.* (1992) 59–68.

[6] E. Anwar, L. Maugis and S. Chakravarthy, A new perspective on rule support for object-oriented databases, in: *Proc. ACM SIGMOD Int. Conf. on the Management of Data* (Washington, D.C., May 1993) 99–108.

[7] M.L. Barja, A. Fernandes, N. Paton, M. Williams, A. Dinn and A. Abdelmoty, Design and implementation of rock and roll: a deductive object-oriented database system, *Information Systems* 20(3) (1995) 185–211.

[8] C. Beeri, New data models and languages – the challenge, in: *Proc. 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (ACM Press, June 1992) 1–15.

[9] C. Beeri, Formal Models for Object Oriented Database, in [41] 405–430.

[10] C. Beeri and Y. Kornatzky, Algebraic optimization of object-oriented query languages, in: *Proc. Int. Conf. on Database Theory* (*Lecture Notes in Computer Science 470*) (Springer-Verlag, 1990) 72–88.

[11] C. Beeri and T. Milo, A model for active object-oriented database, in: *Proc. 17th Int. Conf. on Very Large Data Bases* (1991) 337–349.

[12] H. Branding, A. Buchmann, T. Kudrass and J. Zimmerman, Rules in an Open System: The REACH Rule System, in [48] 111–126.

[13] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca and R. Zicari, Integrating object-oriented data modelling with a rule-based programming paradigm, in: *Proc. 1990 ACM SIGMOD Conf.* (1990) 225–236.

[14] M. Carey, D. DeWitt, J. Naughton, M. Solomon et al., Shoring up persistent applications, in: *Proc. 1994 ACM SIGMOD Conf.* (1994) 383–394.

[15] Y. Caseau, Constraints in an Object-Oriented Deductive Database, in [24] 292–311.

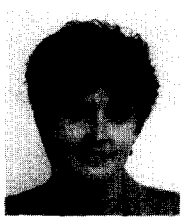
[16] R. Cattell (ed.), The Object-Oriented Database Standard: ODMG-93 (Morgann Kaufmann Publishers, San Francisco, CA, 1994).

[17] S. Ceri, A declarative approach to active databases, in: *Proc. 8th Int. Conf. on Data Engineering* (1992) 452–456.

[18] S. Ceri, P. Fraternali, S. Paraboschi and L. Branca, Active Rule Management in Chimera, in [53] 151–176.

[19] S. Ceri and J. Widom, Deriving production rules for integrity maintenance, in: *Proc. 16th Int. Conf. on Very Large Data Bases* (1990) 566–577.
[20] W. Chen, Declarative specification and evaluation of database updates, in [24] 147–166.
[21] W. Chen and D.S. Warren, C-logic for complex objects, in: *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1989) 369–378.
[22] U. Dayal, Active database management systems, in: *Proc. Third Int. Conf. on Data and Knowledge Bases* (Jerusalem, June 1988) 150–170.
[23] L. Delcambre and K. Davis, Automatic validation of object-oriented database structures, in: *Proc. Fifth Int. Conf. on Data Engineering* (Los Angeles, CA, 1989) 2–9.
[24] C. Delobel, M. Kifer and Y. Masunaga (eds.), Deductive and object-oriented databases, in: *Lecture Notes in Computer Science*, vol. 566 (Springer-Verlag, 1991).
[25] H. Gallaire, J. Minker and J.-M. Nicolas, Logic and databases: A deductive approach, *ACM Computing Surveys* 16(2) (1984).
[26] N. Gehani and H.V. Jagadish, Ode as an active database: constraints and triggers, in: *Proc. 17th Int. Conf. on Very Large Data Bases* (1991) 327–336.
[27] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey and E. Shekita, Starburst mid-flight: as the dust clears, *IEEE Transactions on Knowledge and Data Engineering* 2(1) (1990) 146–160.
[28] E. Hanson, Rule condition testing and action execution in Ariel, in: *Proc. ACM SIGMOD* (San Diego, June 1992) 49–58.
[29] J. Harrison, Condition Monitoring in an Active Deductive Database, Ph.D. Dissertation, Department of Computer Science and Engineering, Arizona State University, August 1992.
[30] J. Harrison and S.W. Dietrich, Integrating active and deductive rules, in: *Proc. 1st Int. Workshop on Rules in Database Systems* (Edinburgh, Scotland, 30th August–1st September 1993, Springer Verlag in Collaboration with the British Computer Society) (1994) 288–305.
[31] A. Hauer and P. Sander, Preserving and generating objects in the LIVING IN A LATTICE Rule Language, in: *Proc. Seventh Int. Conf. on Data Engineering* (1991) 562–569.
[32] R. Hull and D. Jacobs, Language constructs for programming active databases, in: *Proc. 17th Int. Conf. on Very Large Data Bases* (1991) 455–467.
[33] R. Hull and M. Yoshikawa, ILOG: Declarative creation and manipulation of object identifiers, in: *Proc. 16th Int. Conf. on Very Large Data Bases* (1990) 455–468.
[34] A. Jahne, S. Urban and S. Dietrich, PEARD: A prototype environment for active rule debugging, to appear in *J. Intelligent Information Systems* (1996).
[35] A. Karadimce and S. Urban, A framework for declarative updates and constraint maintenance in Object-Oriented Databases, in: *Proc. Ninth Int. Conf. on Data Engineering* (Vienna, April 1993) 391–398.
[36] A. Karadimce and S. Urban, CDOL: A declarative platform for developing OODB Applications, in: *Proc. Int. Phoenix Conf. on Computers and Communications* (Tempe, AZ, 1993) 224–230.
[37] A. Karadimce and S. Urban, Conditional term rewriting as a formal basis for analysis of active database rules, in [54] 156–162.
[38] A. Karadimce and S. Urban, Refined triggering graphs: a logic-based approach to termination analysis in an active object-oriented database, in: *Proc. 12th Int. Conf. on Data Engineering* (New Orleans, LA, 1996) 384–391.
[39] A. Karadimce, Termination and confluence analysis in an active object-oriented database, Ph.D Dissertation, Department of Computer Science and Engineering, Arizona State University, Fall 1996.
[40] M. Kifer and J. Wu, A logic for object-oriented logic programming (Maier's O-logic revisited), in: *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1989) 379–393.
[41] W. Kim, J. Nicolas and S. Nishio (eds.), *Deductive and Object-Oriented Databases* (North-Holland, The Netherlands, 1990).
[42] C. Lecluse, P. Richard and F. Velez, O_2, an object-oriented data model, in: *Proc. 1988 ACM SIGMOD Conf.* (1988) 424–433.
[43] Y. Lou and Z.M. Ozsoyoglu, LLO: An object-oriented deductive language with methods and methods inheritance, in: *Proc. 1991 ACM SIGMOD Conf.* (1991) 198–207.
[44] S. Manchanda, Declarative expression of deductive database updates, in: *Proc. Eighth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1989) 93–100.

[45] S. Naqvi and R. Krishnamurthy, Database updates in logic programming, in: *Proc. Seventh ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1988) 251–262.
[46] A. Olivé and J.A. Pastor, Integrity constraint checking in deductive databases with the internal events method, in: J. Goers and A. Heuer, eds., *Proc. Second Workshop on Foundations of Models and Languages for Data and Objects* (Aigen, Austria, TU Clausthal, September 1990) 139–168.
[47] N. Paton, O. Diaz, M. Howard, J. Campin, A. Dinn and A. Jaime, Dimensions of Active Behavior, in [48] 40–57.
[48] N. Paton and M. Howard (eds.), *Rules in Database Systems* (Scotland, Springer-Verlag, Great Britain, 1994).
[49] U. Schreier, H. Piradesh, R. Agrawal and C. Mohan, Alert: an architecture for transforming a passive DBMS into an active DBMS, in: *Proc. 17th Int. Conf. on Very Large Data Bases* (1991) 469–478.
[50] M. Stonebraker, E. Hanson and S. Potamianos, The POSTGRES rule manager, *IEEE Trans. Software Engineering* 14(7) (1988) 897–907.
[51] S.D. Urban and B.B.L. Lim, An intelligent framework for active support of database semantics, *Int. J. Expert Systems* 6(1) (1993) 1–38.
[52] S.D. Urban and M. Desiderio, CONTEXT: A constraint explanation tool, in: *Data and Knowledge Engineering*, vol. 8 (North-Holland, 1992) 153–183.
[53] J. Widom and S. Ceri (eds.), *Active Database Systems: Triggers and Rules for Advanced Database Processing* (Morgan Kaufmann Publishers, San Francisco, CA, 1996).
[54] J. Widom and S. Chakravarthy (eds.), *Proc. Fourth International Workshop on Research Issues in Data Engineering: Active Database Systems* (IEEE Computer Society Press, Houston, TX, 1994).
[55] M. Winslett, *Updating Logical Databases* (Cambridge University Press, 1990).

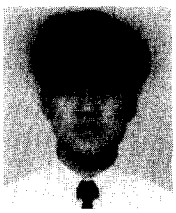


Susan D. Urban received the B.S., M.S. and Ph.D. degrees in computer science in 1976, 1980 and 1987, respectively, from the University of Southwestern Louisiana, Lafayette. She is currently an Associate Professor in the Department of Computer Science and Engineering at Arizona State University. Her research interests include object-oriented database systems, active database systems, multidatabase environments, and engineering databases, with research support from the National Science Foundation and the Defense Advanced Research Projects Agency. She is currently serving as the co-director of the ADOOD (Active, Deductive, Object-Oriented Database) Research Project at ASU. Dr. Urban has served on the editorial board of the IEEE Transactions on Knowledge and Data Engineering and is currently the program co-chair for the 1998 Data Engineering Conference to be held in Orlando, Florida. Dr. Urban is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Phi Kappa Phi Honor Society.

Anton P. Karadimce received his B.S. and M.S. degrees in computer science from the University of Cyril and Methodius, Macedonia. He is currently a doctoral candidate in the Department of Computer Science and Engineering at Arizona State University. Karadimce's work on the definition of CDOL has provided the foundation for the formation of the ADOOD (Active, Deductive, Object-Oriented Database) Research Project at ASU. His research interests include active object databases, object-oriented systems and declarative programming.

Suzanne Wagner Dietrich is an Associate Professor in the Department of Computer Science and Engineering at Arizona State University. She received the B.S. degree in computer science in 1983 from the State University of New York at Stony Brook, and as the recipient of an Office of Naval Research Graduate Fellowship, earned her Ph.D. degree in computer science at Stony Brook in 1987. Dr. Dietrich's areas of teaching and research include the theoretical and practical aspects of databases. Her current research focuses on the integration of active, object-oriented and deductive/declarative databases as well as the application of this emerging database technology to various disciplines such as software engineering. She is currently serving as the co-director of the ADOOD (Active, Deductive, Object-Oriented Database) Research Project at ASU. Dr. Dietrich is a member of the Association for Computing Machinery and the IEEE Computer Society.

Taoufik Ben Abdellatif is a Ph.D. student in the Department of Computer Science and Engineering at Arizona State University, where he also received his B.S. degree in computer systems engineering and his M.S. degree in computer science. Abdellatif has been a research assistant on the ADOOD (Active, Deductive, Object-Oriented Database) Research Project at ASU since 1995. His research interests include active and object-oriented database systems with emphasis on the architectural design and run-time support for such systems. Taoufik is an active member of the ASU Chapter of the Association for Computing Machinery.



Hon Wai Rene Chan received his B.S. degree in computer science from the Department of Computer Science and Engineering at Arizona State University in 1995. He is currently an M.S. student in computer science at Arizona State University. Chan has been a research assistant on the ADOOD (Active, Deductive, Object-Oriented Database) Research Project at ASU since 1995. His research interests include object-oriented and active databases. He is also interested in object-oriented software engineering. His M.S. thesis topic is in the area of testing issues for active database rules.