A Framework for Constraint Management in Object-Oriented Databases

Jong P. Yoon
School of Information Technology
Goerge Mason University
Fairfax, VA 22030-4444

Larry Kerschberg
Information & Software Systems Engr
Goerge Mason University
Fairfax, VA 22030-4444

Abstract

We introduce the concept of *rule schema* in this paper in order to support constraints in the active object-oriented paradigm. The rule schema provides the meta-knowledge associated with constraints analogous to the way a database schema provides meta-data about the database objects. It is used to *compile* the constraints into clauses which are then "attached" to appropriate object attributes. The compiled constraints incorporate the semantics associated with *inheritance* over generalization hierarchies and *references* to simple objects which are constituents of complex objects. These clauses are evaluated and enforced whenever an attribute value is retrieved or updated. The update semantics for update propagation of object instances is embedded in the compiled constraints; constraints therefore play a role similar to methods in the model, but are specified explicitly rather than procedurally. They can therefore be managed by the active object-oriented database.

1 Introduction

Object-oriented database systems have advantages over relational or record-oriented database systems. They model *complex objects*, which are constituted by one or more objects of different object types. Further, in object-oriented systems one defines behavioral semantics in terms of operations, or methods, associated with objects. Our approach is to associate constraints to objects.

Constraints can be user-defined and dispersed over object types as methods associated with object types [7, 10]. Because constraints are tightly coupled with the object, it is simple to maintain object consistency within an object when the object is updated [2]. But, because of this tight coupling, it is hard to maintain overall database consistency if the effects of the update of an object are propagated and may lead to additional inconsistencies in other object types [6, 7]. This difficulty occurs particularly when (1) the constraints span more than one object along generalization hierarchies, (2) the constraints refer to complex objects, which then refer to one or more constituent objects of different types, and (3) complex objects in a generalization hierarchy refer to (simple) objects which are in another generalization hierarchy.

To overcome this difficulty, there are three well known approaches: *naive approach, limited approach*, and *expert approach*. In a naive approach, all constraints are defined in one place (e.g., a root node or the "metaclass") and all constraints are considered for enforcement. The second approach, used in [5, 6, 7, 8, 10], allows constraints to be defined and associated with an object. Only those constraints (so called *intra-object constraints*) associated with an object or inherited from a supertype object are considered when an update or a query is presented to the object. However, no propagation of update effects are considered in object-oriented databases and so overall database consistency can not be ensured. An *expert approach* allows constraints (even *inter-object constraints*) to be defined and associated with an object(s). Moreover, constraints are compiled, partly developed in [3], and "materialized."[1] In doing so, the update effects are propagated thereby ensuring database consistency. This paper describes the expert approach.

We believe that the natural way of developing an expert approach is to use *meta-knowledge*, defined in this paper as a *rule schema*, that determines the dependencies among constraints. This paper will show how to construct a rule schema and demonstrate how these schemas allow for constraint compilation. The main contribution of this paper is to materialize constraints: the applicable (both intra- and inter-object) constraints are compiled and associated with the object type; and so these constraints can be triggered *actively* over objects in generalization and aggregation hierarchies to maintain semantic integrity.

This paper is organized as follows. The remainder of this section provides an illustrative example and reviews the literature. In Section 2, structural and behavioral models are briefly reviewed. The concept of *rule schema* is introduced in Section 3. Syntax and semantics of the rule schema are presented. Using this rule schema, constraints are compiled in terms of each attribute in the object types in Section 4. Section 5 reconstitutes the object-oriented database schema with

[1] In this paper, *materialization* denotes the compilation of constraints into clauses, for both pre- and post-conditions, that are attached to attributes of objects. This can be compared to compiling the "connection graph" of predicate unifications in a deductive database.

Proj	(budget: money, manYears: int, type: char[1],	
	constraint (pb1:	Proj.budget > \$200000 $\wedge$ Proj.manYears > 1000 $\rightarrow$ Proj.type = "B"))
DefProj	(*isa*: Proj, class: char[15],	
	constraint (pc1:	DefProj.type $\geq$ "B" $\rightarrow$ DefProj.class = "top secret"))
TechMgr	(clearanceLevel: char[12]; supervises: {Proj},	
	constraint (tc1:	TechMgr.supervises.DefProj.class = "top secret"
		$\rightarrow$ TechMgr.clearanceLevel = "top secret"))

Figure 1: An Example Object-Oriented Database

the applicable constraints being compiled and associated within appropriate object types. Finally, contributions and open research issues are addressed in Section 6.

1.1 Motivating Examples

Consider an object-oriented database schema, using the generic notation of EXTRA/EXCESS [17], and IQL [1], where attributes are defined as a pair "attribute: domain" and constraints are associated with an object type. Consider an example shown in Figure 1 consisting projects (Proj), defense projects (DefProj) which are specializations of projects, and technical managers (TechMgr) who supervise projects. DefProj *isa* Proj in our example. TechMgr refers to one or more Proj's through the role of supervising, denoted by "supervises : {Proj}." In the constraint language, a *term* is one or more forms of "object.attribute" pairs. For example, the attribute "class" of a defense project supervised by a technical manager can be represented as a term "TechMgr.supervises.DefProj.class," where TechMgr and DefProj are object types, and "supervises" and "class" are attributes. Each constraint has a label, for example, pb1 represents that the project whose budget is over \$200,000 and man-year is over 1000 should be of type "B."

Suppose an update is presented to increase project budgets by 10%. The problem is to determine whether the intended update will lead to a consistent database state, and whether the effects of updates must be propagated. Conventional processing requires only that constraints associated with the object type "Proj" be applied. That is, constraint "pb1" is evaluated. However, additional resources (i.e., methods as constraints, or relationships between object types) may be impacted.

- How does the system know which constraints should be selected to check for additional inconsistencies? Since the project budget is increased, the project type may be modified according to pb1. What if its effects cause additional inconsistencies? What if Proj.class doesn't satisfy the constraint "pc1?" By the same token, how might the constraint "tc1" be activated?

- Suppose the database system is required to determine the constraints to apply for constraint execution. How does the system evaluate the term "DefProj.type" in pc1 as triggered by the effect of the execution of pb1 where "Proj.type" is evaluated? Moreover, how does the system evaluate the term "TechMgr.supervises.DefProj.class" in tc1 as triggered by the effect of the execution of pc1 where "DefProj.class" is evaluated? It must reason about both generalization and aggregation hierarchies.

$$\begin{tikzcd}
RS \arrow[r] \arrow[d, dash, "(2)"'] & R \arrow[d, dash, "(1)"] \\
DS \arrow[r] & D
\end{tikzcd}$$

Figure 2: Using Meta-Knowledge and Meta-Data for Constraint Compilation

To resolve these problems, we will present the concept of rule schema as a mechanism for an active database to determine all applicable constraints. The new mechanism is sketched in Figure 2. The horizontal arrows describe the relationships between instances and their schema, and the vertical lines describe evaluation and compilation processes. The Database Schema, DS, is an abstraction (meta-data) of database, D. The Rule Schema, RS, is an abstraction (meta-knowledge) of constraints, R. Typically, R are evaluated and triggered against D as designed by the line (1) in Figure 2. As opposed to (1), the line (2) denotes that RS can interact with DS to reason about the interaction of the constraints and the schema in terms of the object-oriented paradigm which supports *property inheritance* and *object referencing*, along generalization and aggregation hierarchies, respectively.

1.2 Related Work

There has been considerable work [13, 14, 16] dealing with the expressive power of relational constraint management. They view database relations as the predicates of constraints. Chakravarthy [3] developed the notion of constraint compilation for each database relation. The constraints are compiled by the refutation of the negation for each relation. However, the constraint compilation technique which will be presented in this paper is performed by using meta-knowledge for an update or a query.

Now, let's turn to constraint management in active databases. An *active database* is a database system enhanced with a rule processing capability in which the rules monitor the database state and automatically execute when rule conditions are satisfied.

The literature [5, 6, 12] groups constraints and rules together. By using a declarative language, constraints and rules can be specified uniformly. A drawback of these approaches is that a constraint implemented by these rules cannot activate other constraints unless those constraints are specified explicitly in the rules or changes in a database state are monitored thereby triggering another constraint. Our approach is more general in that we determine the set of constraints applicable to a database update, before effecting that update.

Finally, in object-oriented databases, constraints are associated with object types (or classes). Since constraints are scattered in an object-oriented database, inference is limited either to within an object type to which the constraints are associated or outside the scope of individual object types by means of *explicit triggers*. Propagation algorithms (e.g., in [8]) are executed in a predesignated manner for all users to make use of the same database constraints. In these algorithms, constraints are activated by explicitly predefined activators.

Although Hull and Su [8] have developed object-oriented constraint languages, the concept of *method* is ignored. The literature [2] does not explicitly address that constraints play the role of methods. However, [10] has introduced the concept of method to define object-oriented constraints. Inference is limited to using those constraints that are associated with an object type or its subtypes. This restriction is examined in [6, 7]. In this paper, we develop a rule schema to represent and reason with meta-knowledge, by which constraints may be compiled.

2 Object-oriented Database Model

Object-oriented databases are comprised of both structural and operational aspects of data. In this section, structural and behavioral models of the object-oriented database are briefly reviewed.

2.1 Structural Model – A Brief Review

We assume the existence of an infinite set of symbols, called *attribute*, and for each attribute a, of an infinite set of values, denoted dom(a), called the *domain* of a. An *object type* ($\mathcal{O}$) is defined as a finite set $(a_1, a_2, ...)$ of attributes. The instance t of $\mathcal{O}$ is a mapping from $\mathcal{O}$ to dom($\mathcal{O}$) with the restriction such that, for each a_i in $O \in \mathcal{O}$, $t(O)$ is in dom($O.a_i$). The instance t is identified by an *object identifier*(OID) as used in O-Logic [11]. The OID is a powerful programming primitive for database constraint languages [1].

A *complex object* refers to one or more objects or object fragments connected by inter-object references

Rule	::= **RID** : Antecedent $\rightarrow$ Consequent
Antecedent	::= Predicate [$\wedge$ Predicate]*
Consequent	::= Predicate $\mid$ $\perp$
Predicate	::= Term $\Theta\mathcal{D}$
Term	::= [Complex_object_path(**OID**).]* Primitive_path(**OID**)
Complex_object_path	::= Primitive_path
Primitive_path	::= **Object.attribute**
Θ	::= $= \mid < \mid \leq \mid > \mid \geq \mid \neq$
$\mathcal{D}$	::= *dom*(Primitive_path)

Figure 4: Constraint Language

[4, 9]. The references of a complex object ($O \in \mathcal{O}$) is defined as a mapping from $O.a$ to dom($O.a$) that is a set of OID's.

In our company example, a project of type Proj is supported by one or more agencies of type Agency. A complex object "Proj" is defined as follows:

Proj (	oid:	oid_type,
	supported:	{**ref** Agency},
	manYears:	int)

where OID's are generated by an object-oriented database system and invisible to users, and its type is called *oid_type*. The value of Proj's attribute "manYears" is mapped to domain "int," and the references of an attribute "supported" is mapped to one or more OID's of an object type "Agency," represented in braces.

As the properties of an object type are inherited by subtypes of objects, the references of a complex object are inherited as well. For example, suppose that DefProj *isa* Proj. An attribute "manYears" of type Proj can inherited by type DefProj unless overridden. Likewise, the type Proj is supported by one or more agencies of type Agency, so DefProj is also supported by one or more agencies again due to the inheritance. This notion is very important in managing the constraints which are defined on complex objects as we will discuss in a later section.

2.2 Behavioral Model

Constraints are defined as disjunctions of predicates in a first-order Horn clause logic: $L_1 \& L_2 \& ... \& L_n \rightarrow L_0$, where L_i is a predicate. Each predicate is of the form "*object.attribute(OID)* Θ $\mathcal{D}$," denoting the attribute of the object of OID is compared with the domain value, where Θ is a arithmetic comparison operator (e.g., $=, >, \neq$, etc), and $\mathcal{D} \in$ dom(*object.attribute*). OID can be either a variable or a value. Each constraint is identified by constraint identifier (RID). The detailed grammar is given in Table 4 by which constraints in Figure 3 are defined. For example, constraint ba1 represents that "there is no project whose budget is over 700K," and constraint pa1 represents that "any project which is supported by NSF should not have a budget over 500K."

The references being mapped between a complex object and a simple object can be represented by

```
define type Proj
Attribute ( oid:          oid_type,
            budget:       money,
            supported:    {ref Agency},
            manYears:     int,
            type:         char[1])

Rule ( pa1:  Proj.supported(x).Agency.name(y) = "NSF" →Proj.budget(x) < 500000,
       ba1:  Proj.budget(x) > 700000 → ⊥,
       pt1:  Proj.budget(x) > 200000 ∧ Proj.manYears(x) > 1000 → Proj.type(x) = "B",
       pt2:  Proj.budget(x) < 50000 ∧ Proj.manYears(x) < 500 → Proj.type(x) = "D")
end define
```

```
define type DefProj
Superclass (Proj)
Attribute ( oid:          oid_type,
            class:        char[15])

Rule ( pc1:   DefProj.type(x) ≥ "B" → DefProj.class(x) = "top secret")
end define
```

```
define type Agency
Attribute ( oid:     oid_type,
            name:    char[15] )
end define
```

Figure 3: Object Types in Company Database Example

using a *functional path* [15]. The term used in constraints is of a form "[Complex_object_path(x).]* Primitive_path(y)," denoting that a complex object x refers to one or more simple objects y. Note that we define the innermost functional path of a term as "primitive functional path" (in short, primitive path) while functional paths of the remainder are referenced to as "complex object functional path" (in short, complex object path). For example, a functional path Proj.supported.Agency.name, serves as a connection from a complex path "Proj.supported" to primitive paths "Agency.name."

3 The Rule Schema

A rule schema is a triple: $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$, where $\mathbf{R}$ is a set of RID's, $\mathbf{A}$ and $\mathbf{C}$ are respectively sets of functional paths used in the constraint and in the consequent of a constraint. Notice that a Horn clause based rule language leads to constituting set $\mathbf{A}$ and element $\mathbf{C}$.

For example, given the constraints:

pt1: Proj.budget(x) > 200000 ∧ Proj.manYears(x) > 1000 → Proj.type(x) = "B"
pt2: Proj.budget(x) < 50000 ∧ Proj.manYears(x) < 500 → Proj.type(x) = "D"

the rule schema for constraints pt1 and pt2 is: $\langle \{Proj.budget, Proj.manYears\}, \{pt1, pt2\}, \{Proj.type\} \rangle$. Rule schema represents a *skeleton* of the dependencies among object's attributes.

3.1 Syntax

The alphabet of a rule schema consists of terms such as (1) a set of functional paths, $\mathbf{A}$ and $\mathbf{C}$, used in constraint specifications, and (2) a set of constraint names $\mathbf{R}$.

The term in a rule schema is defined as follows: (1) a primitive path, *Object.attribute* $(O.a)$, is a term; (2) a complex object path $(O_2.f.O_1.g)$ referring to either a primitive path or another complex object path is a term; (3) a constraint name (r) is a term.

3.2 Semantics

Given term set $\mathbf{A}$ and term $\mathbf{C}$, we define a rule schema, $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$, for the set $\mathbf{R}$ of constraints. The individual constraint r in $\mathbf{R}$ is a mapping from the terms "Object.attribute" in a rule schema to the predicate "Object.attribute(OID) Θ dom(Object.attribute)" in a constraint. Each constraint is represented at most once in the rule schema, but more than one constraint may have the same rule schema. For example, for either pt1 or pt2, there is only one rule schema as above.

If there are constraints applicable for a given attribute, they must be a finite set of constraints $\mathbf{R}$. Consider once again a rule schema in the previous example, $\langle \{Proj.budget,\ Proj.manYears\},\ \{pt1, pt2\},\ \{Proj.type\} \rangle$, constraints pt1 and pt2 which specify the dependency from Proj.budget and Proj.manYears to Proj.type must be in the rule schema above, either pt1 or pt2. In other words, pt1 and pt2 are *instantiated* from the rule schema. All constraints to be considered are instantiated from rule schemas.

3.3 Properties

We are now ready to characterize the applicable constraints. Constraints that are represented over an object may also apply over objects of subtypes or a complex object. The rule schema can be used for various different types: (1) the rule schema of the constraints which are inherited by a subtype can be used in the objects of the subtype; (2) the rule schema of the constraints associated with a simple object type can be referenced by the complex object type. The former is possible by inheritance and the latter is possible by reference mapping. In order for constraints to apply over objects of different types, their rule schema should extend over other types. We call the rule schema extended over other types an *equivalent rule schema*.

Definition 1 (*Equivalence of Rule Schemas.*) *We define two rule schemas,* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$ *and* $\langle \mathbf{A}', \mathbf{R}', \mathbf{C}' \rangle$ *to be equivalent if* (1) *They are defined over the same set of constraints,* $\mathbf{R} \equiv \mathbf{R}'$*; and* (2) *The elements of* $\mathbf{A}, \mathbf{A}', \mathbf{C}$ *and* $\mathbf{C}'$ *are related in that there exist paths through inheritances or reference mappings.* □

The second condition means that equivalent rule schemas are constructed over generalization and aggregation hierarchies in object-oriented databases. An equivalent rule schema uses the same constraints as the original schema and has the same results, but by simpler triggering. By obtaining equivalent rule schemas, a constraint can span objects not only of the same object type but also of different types along generalization and aggregation hierarchies.

3.3.1 Inheritance

Given an inheritance hierarchy defined by the *isa* relationship, we can define object inheritance. An object inherits the properties of its super-object. Unless the inherited properties are overridden, they are available to the object. Similarly, rule schemas are also available to subtypes by typing and unifying[2] the supertype with the subtypes. The following inheritance property of rule schema is introduced:

Definition 2 (*Inheritance.*) *For* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$*, if any object type in the term* $\mathbf{A}$ *is a supertype of those in* $\mathbf{A}'$*, then its equivalent rule schema is* $\langle \mathbf{A}', \mathbf{R}, \mathbf{C} \rangle$*, available to the objects of its subtypes. Similarly,* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C}' \rangle$ *may be also an equivalent rule schema if the term in* $\mathbf{C}$ *is a supertype of* $\mathbf{C}'$. □

Example 1

Suppose a rule schema is defined at type Proj: $\langle \{Proj.budget, Proj.manYears\}, \{pt1, pt2\}, \{Proj.type\} \rangle$ is inherited to a subtype "DefProj" by unifying "Proj" with "DefProj," thereby an equivalent rule schema is: $\langle \{DefProj.budget, DefProj.manYears\}, \{pt1, pt2\}, \{DefProj.type\} \rangle$. That is, the constraints pt1 and pt2 can be invoked when an update is presented over not only a project but also a defense project. □

3.3.2 Complex Object Referencing

As discussed earlier, the rule schema over a complex object refers to one or more simple objects, while the rule schema of a simple object may also inversely refer to the complex object. We call the former *referencing* and the latter *inverse-referencing*. Referencing concatenates a complex object path with the term in a constraint, e.g., "complex_object_path.term."

[2]As in [10], the unification algorithm is used accordingly to incorporate type or class information.

Inverse-referencing is made by concatenating the inverse pointer[3] of "complex_object_path" with the term. For example, Mgr.*supervises*.Proj is inversely Proj.*supervised*.Mgr, meaning that "a manager supervises projects" is inversely "projects are supervised by a manager" where $supervised = supervises^{-1}$. From this notion, we have the following properties[4]:

Definition 3 (*Referencing.*) *Consider* $a \in \mathbf{A}$ *in* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$*. For a complex object term* a'*, if a complex object refers to simple objects through the term* $a'.a$ *then its equivalent rule schema is* $\langle \mathbf{A}', \mathbf{R}, \mathbf{C} \rangle$*, where* $a'.a \in \mathbf{A}'$*. Similarly,* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$ *can be equivalent to* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C}' \rangle$ *where* $c'.c \in \mathbf{C}'$. □

Definition 4 (*Inverse-referencing.*) *Consider* $a \in \mathbf{A}$ *in* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$*. For an inverse pointer of complex object term,* a'^{-1}*, if a complex object refers to simple objects through the term* $a'.a$ *then its equivalent constraint schema is* $\langle \mathbf{A}', \mathbf{R}, \mathbf{C} \rangle$*, where* $a'^{-1}.a \in \mathbf{A}'$*. Similarly,* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$ *can be equivalent to* $\langle \mathbf{A}, \mathbf{R}, \mathbf{C}' \rangle$*, where* $c'^{-1}.c \in \mathbf{C}'$. □

Example 2

The constraint pc1 in Figure 3 is defined at the type DefProj and can extend over to the type Mgr. $\langle \{Proj.type\}, \{pc1\}, \{Proj.class\} \rangle$ can be referred by its complex object "Mgr" by concatenating "Mgr.supervises" to "Proj.class," thereby its equivalent rule schema is $\langle \{Mgr.supervises.Proj.type\}, \{pc1\}, \{Mgr.supervises.Proj.class\} \rangle$. Consider another example: $\langle \{Mgr.supervises.DefProj.class\}, \{tc1, tc2\}, \{Mgr.clearanceLevel\} \rangle$. First, its equivalent rule schema $\langle \{DefProj.supervised.Mgr.supervises.DefProj.class\}, \{tc1, tc2\}, \{DefProj.supervised.Mgr.clearanceLevel\} \rangle$ is obtained by concatenating the inverse pointer "DefProj.supervised." By neglecting a cyclic pointer, we obtain an equivalent rule schema $\langle \{DefProj.class\}, \{tc1, tc2\}, \{DefProj.supervised.Mgr.clearanceLevel\} \rangle$ Thus, it is inverse-referenced by the simple object DefProj. One or more Proj's are connected with Mgr, meaning that a technical manager supervises one or more projects, and thereby the "project's class" is equivalent to "class of a project which is supervised by a technical manager." □

[3]Maintenance of "inverse" pointers is well described [15] and Vbase.

[4]Note that for object types $O_i, O_j \in \mathcal{O}, (i \neq j)$ and attributes $f, g, (f \neq g)$, the term $O_1.f^{-1}.O_2.f.O_1.g$ can be simplified to $O_1.g$ if the mapping f from O_1 to O_2 is "total", noting that it does not have to be "one-to-one" however. That is, for every object (instance) $o_1 \in O_1$, there exists an $o_2 \in O_2$ such that $o_2 = f(O_1)$. Therefore $o_1 \in f^{-1}(o_2)$. For example, Proj.supervised.Mgr.supervises.Proj.type is simplified to Proj.type because both terms deal with the "same objects," i.e., Proj.type, if all objects of type Proj participate in the reference mapping with objects of type Mgr.

4 Constraint Compilation

The rule schema determines the constraints to be enforced for queries or updates. Using the rule schema, constraints ($p \rightarrow q$) can be compiled into the clauses ($\neg p \vee q$) which are then associated with an object type. In this section, constraint compilation algorithm is described. Objects can be reconstituted with the compiled constraint clauses and so the constraints are "materialized."

4.1 Algorithm

Given the rule schema $\langle \mathbf{A}, \mathbf{R}, \mathbf{C} \rangle$, consider a term a which is a pair "object.attribute." Using the properties of the rule schema, the constraints are compiled as follows.

- If $a \in \mathbf{C}$, all constraints [5] in rule set $\mathbf{R}$ are compiled to ensure the correctness of updates or queries.
- If $a \in \mathbf{A}$, all constraints in rule set $\mathbf{R}$ are compiled to ensure the effects of updates.
- Suppose that $c \in \mathbf{C}$ is also an element of $\mathbf{A}'$ of a rule schema $\langle \mathbf{A}', \mathbf{R}', \mathbf{C}' \rangle$. In addition to compiling the constraints in $\mathbf{R}$, by the rule schema transitivity, the appropriate constraints in $\mathbf{R}'$ are compiled to incrementally maintain the propagation of the update effects.

The algorithm compiles not only the exactly matched constraints, but also all other constraints that must be considered. It ensures that all relevant constraints will be considered.

4.2 Object Reconstitution

In this section, the notion of materializing constraints for object types is discussed. We transform the compiled constraints into clauses such that all the clauses must hold in order to commit the update. The conjunction of clauses is then associated with an object type, similar to a *procedure attachment* (e.g., IF_NEEDED or IF_ADDED) so that the constraints can be automatically triggered as object attributes are retrieved or updated. The attribute specification with which the compiled constraints are associated is of the form

$$\text{attribute\_name: domain} \left\{ \begin{array}{l} \text{RETRIEVED\_IF} : R_i; \\ \text{MODIFIED\_IF} : R_i, R_j, R_k; \end{array} \right\}$$

where R_i is a conjunction of clauses for checking the correctness of a query or an update (Q), R_j is a conjunction of clauses for ensuring the effects of Q , and R_k is a conjunction of clauses for checking the propagation of Q's effects. This notion has been partly developed as the terms *pre-condition* and *post-condition* in the constraint language of [14].

A system allows the attribute to be **retrieved if** R_i holds when a query is presented, or to be **modified if** R_i, R_j, R_k hold when an update is presented. These conjunctions of clauses are grouped into two procedure attachments: RETRIEVED_IF and MODIFIED_IF. In doing so, not only the correctness of the modification (i.e., Update, Insert, or Delete) is checked, but also the effects of the modification and the propagation effects are ensured. Furthermore, R_i is evaluated to check the correctness of a query (i.e., Retrieve) when the query is asked over attribute_name.

[5] All accepted changes of the database should satisfy all constraints.

5 The Object-oriented Database Model – Revisited

By applying the compilation algorithm, an object-oriented schema can be reconstituted. All constraints for pre- and post-conditions related with an attribute are associated with the attribute. These constraints are then automatically triggered when the attribute is updated. Given Figure 3 and 5 which depict portions of an application, consider the following rule schemas with respect to attribute "DefProj.type."

$\langle \{Proj.budget\}, \{ba1\}, \perp \rangle$
$\langle \{Proj.budget, Proj.manYears\}, \{pt1, pt2\}, \{Proj.type\} \rangle$
$\langle \{DefProj.type\}, \{pc1\}, \{DefProj.class\} \rangle$
$\langle \{Mgr.supervises.DefProj.class\}, \{tc1, tc2\},$
$\{Mgr.clearanceLevel\} \rangle$
$\langle \{Mgr.supervises.DefProj.budget\}, \{ts1\}, \{Mgr.salary\} \rangle$
$\langle \{Proj.supported.Agency.name\}, \{pa1\}, \{Proj.budget\} \rangle$

As shown in Figure 2, the notion of inheritance and referencing is applied to reason about the database and rule schemata to compile constraints. Keeping this notion in mind, we will compile the constraints applicable to "DefProj.type" by obtaining equivalent rule schemas and applying the algorithm. Let's first find equivalent rule schemas. For $\langle \{Proj.budget, Proj.manYears\}, \{pt1, pt2\}, \{Proj.type\} \rangle$, an equivalent rule schema is $\langle \{DefProj.budget, DefProj.manYears\}, \{pt1, pt2\}, \{DefProj.type\} \rangle$ due to inheritance.

Consider $\langle \{Mgr.supervises.DefProj.class\}, \{tc1, tc2\}, \{Mgr.clearanceLevel\} \rangle$. Its equivalent rule schema is $\langle \{DefProj.class\}, \{tc1, tc2\}, \{DefProj.supervised.Mgr.clearanceLevel\} \rangle$ by inverse-referencing as shown in earlier section.

Hence, the equivalent rule schemas for the type DefProj are obtained by using inheritance and referencing:

$\langle \{DefProj.budget\}, \{ba1\}, \perp \rangle$
$\langle \{DefProj.budget, DefProj.manYears\}, \{pt1, pt2\},$
$\{DefProj.type\} \rangle$
$\langle \{DefProj.type\}, \{pc1\}, \{DefProj.class\} \rangle$
$\langle \{DefProj.class\}, \{tc1, tc2\},$
$\{DefProj.supervised.Mgr.clearanceLevel\} \rangle$
$\langle \{DefProj.budget\}, \{ts1\},$
$\{DefProj.supervised.Mgr.salary\} \rangle$
$\langle \{DefProj.supported.Agency.name\}, \{pa1\},$
$\{DefProj.budget\} \rangle$

Since all terms in these equivalent rule schemas refer to the type DefProj, it is feasible to apply the compilation algorithm. An attribute DefProj.type is the same as the c part of the rule schema of pt1 and pt2, so

```
define type Mgr
Superclass (Emp)
Attribute ( oid:               oid_type,
            supervises:        {ref Proj},
            clearanceLevel:    char[12])

Rule ( tc1:  Mgr.supervises(x).DefProj.class(y) = "top secret"
             → Mgr.clearanceLevel(x) ≥ "top secret",
       tc2:  Mgr.Supervises(x).DefProj.class(y) = "confidential"
             → Mgr.clearanceLevel(x) ≥ "confidential",
       ts1:  Mgr.supervises(x).DefProj.supported(y).Agency.name(z)
             = "DARPA" → Mgr.salary(x) > 60000)
end define
```

```
define type Emp
Attribute ( oid:     oid_type,
            name:    person_name,
            salary:  money )
end define

define type TechMgr
Superclass (Engr)
Attribute ( oid:              oid_type,
            years_of_expr:    int )
end define

define type Engr
Superclass (Emp)
Attribute ( oid:       oid_type,
            Degree:    char[3] )
end define
```

Figure 5: Object Types in Company Database Example – Extended

the constraints pt1 and pt2 are compiled into clauses. The clause R_i for ensuring the correctness of an update is obtained (or instantiated) from the equivalent rule schema above.

$$R_i : \left\{ \begin{array}{l} \neg(DefProj.budget(x) > 200000) \\ \vee\neg(DefProj.manYears(x) \\ > 1000) \vee DefProj.type(x) = \text{“}B\text{”}, \\ \neg(DefProj.budget(x) < 50000) \\ \vee\neg(DefProj.manYears(x) \\ < 500) \vee DefProj.type(x) = \text{“}D\text{”} \end{array} \right\}$$

To ensure the effects of this update, we need to select the constraints to apply. DefProj.type $\in$ {DefProj.type} holds in the rule schema, and therefore constraint pc1 is compiled. So, the clause R_j is:

$$R_j : \left\{ \begin{array}{l} \neg DefProj.type(x) \geq \text{“}B\text{”} \\ \vee DefProj.class(x) = \text{“}topsecret\text{”} \end{array} \right\}$$

Since the effects of the update propagate, additional constraints need to be evaluated. For this reason, the constraints tc1 and tc2 are compiled to clause R_k. These clauses are instantiated from the equivalent constraint schema above.

$$R_k : \left\{ \begin{array}{l} \neg DefProj.class(x) = \text{“}topsecret\text{”} \\ \vee DefProj.supervised(x).Mgr.clearanceLevel(y) \\ \geq \text{“}topsecret\text{”}, \\ \neg DefProj.class(x) = \text{“}confidential\text{”} \\ \vee DefProj.supervised(x).Mgr.clearanceLevel(y) \\ \geq \text{“}confidential\text{”} \end{array} \right\}$$

By associating these clauses with an object attribute DefProj.type, the type DefProj can be reconstituted as shown in Figure 6.

There are several advantages to this approach: All constraints associated with an object type do not have to be evaluated when an update is presented over particular attributes. Instead, only the constraints associated with the attribute of an object type are evaluated. In addition, since additional constraints related to the effects of the attribute update are compiled, overall database consistency is guaranteed. However, the redundancies should be eliminated when a query or an update is specified over many attributes and those attributes contain the same constraint clauses; this is a well known typical query optimization issue. If there are more than one rule schema to apply as traced, one may either consider all rule schemas or choose the most relevant one for the object type. Choosing the most relevant rule schema is a topic of ongoing research.

6 Contributions and Future Work

We have developed the concept of rule schema by which the constraints are compiled and associated with an object type. The constraints associated with the object type can be used to maintain: (1) Database updates, (2) The effects of the update, (3) The propagation of the update effects.

The contribution of this paper is as follows: (1) An algorithm for compiling constraints is developed. Equivalent constraints are obtained to span objects of not only the same object type but also different types. The equivalent constraint is instantiated from the equivalent rule schema which is possibly also an abstraction of that constraint. (2) Object types are reconstituted by associating constraint clauses to appropriate objects. In doing so, constraints are materialized and they span over several object types. This is a unified tool for maintaining overall database consistency as stated above.

In addition, this compilation approach can be also used for rule compilation which may correct constraint violations, which is an area of current research. Production rules for correcting constraint violations are compiled together with constraints. Both the compiled constraints and rules are associated with object types. Therefore, semantic invariants are ensured by constraints and otherwise, corrections are enumerated by production rules in a specific sequence.

Finally, the use of the rule schema concept provides a formal way to (1) reason about constraints in a database, (2) study the impact of operations and their side-effects, and (3) compile constraints and associate

```
define type DefProj
Attribute ( oid:          oid_type,
            budget:       money,
            supported:    {ref Agency},
            manYears:     int,
                                  ⎧ RETRIEVED_IF :
                                  ⎪ $\neg(DefProj.budget(x) > 200000) \vee \neg(DefProj.manYears(x) > 1000)$
                                  ⎪ $\vee DefProj.type(x) = \text{“}B\text{”}$,
                                  ⎪ $\neg(DefProj.budget(x) < 50000) \vee \neg(DefProj.manYears(x) < 500)$
                                  ⎪ $\vee DefProj.type(x) = \text{“}D\text{”}$;
                                  ⎪ MODIFIED_IF :
                                  ⎪ $\neg(DefProj.budget(x) > 200000) \vee \neg(DefProj.manYears(x) > 1000)$
            type:         char[1] ⎨ $\vee DefProj.type(x) = \text{“}B\text{”}$,
                                  ⎪ $\neg(DefProj.budget(x) < 50000) \vee \neg(DefProj.manYears(x) < 500)$
                                  ⎪ $\vee DefProj.type(x) = \text{“}D\text{”}$,
                                  ⎪ $\neg DefProj.type(x) \geq \text{“}B\text{”} \vee DefProj.class(x) = \text{“}topsecret\text{”}$,
                                  ⎪ $\neg DefProj.class(x) = \text{“}topsecret\text{”}$
                                  ⎪ $\vee DefProj.supervised(x).Mgr.clearanceLevel(y) \geq \text{“}topsecret\text{”}$,
                                  ⎪ $\neg DefProj.class(x) = \text{“}confidential\text{”}$
                                  ⎩ $\vee DefProj.supervised(x).Mgr.clearanceLevel(y) \geq \text{“}confidential\text{”}$;
            class:        char[15] )
end define
```

Figure 6: Revised Object Type in Company Database Example

them with appropriate objects. In fact, the meta-data, DS, and meta-knowledge, RS, can be used to support intelligent access to meta-information for query processing.

References

[1] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–173, Portland, OR, 1989.

[2] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 225–236, Atlantic, NJ, 1990.

[3] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):163–207, June 1990.

[4] Q. Chen and G. Gardarin. An implementation model for reasoning with complex objects. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 164–172, Chicago, IL, 1988.

[5] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long running activities with triggers and transactions. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 204–214, Atlantic City, NJ, 1990.

[6] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: a uniform approach. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 317–326, Barcelona, 1991.

[7] N. Gehani and H. V. Jagadish. Ode as an active database: constraints and triggers. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 327–336, Barcelona, 1991.

[8] Richard Hull and Jianwen Su. On accessing object-oriented databases: expressive power, complexity, and restrictions. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 147–158, Portland, Oregon, 1989.

[9] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 148–157, Denver, CO, 1991.

[10] Yanjun Lou and Z. Meral Ozsoyoglu. LLO: an object-oriented deductive language with methods and method inheritance. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 198–207, Denver, CO, 1991.

[11] D. Maier. A logic for objects. In *Proc. of the Workshop on Foudnations of Deductive Databases and Logic Programming*, pages 6–26, Washington, DC, 1986.

[12] Matthew Morgenstern. *Intelligent Database Systems*. Technical Report SRI Project 2201, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, May 1989.

[13] Geoffrey Phipps and Marcia A. Derr. Glue-Nail: a deductive database system. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 308–317, Denver, Colorado, 1991.

[14] A. Shepherd and L. Kerschberg. Constraint management in expert database systems. In L. Kerschberg, editor, *Expert Database Systems, Proc. from the First Intl.Workshop*, pages 443–468, Benjamin/Cummings, 1986.

[15] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–171, 1981.

[16] M. Stonebraker, A. Jhingran, J. Goh, and S Potamianos. On rules, procedures, caching and views in data base systems. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 281–290, Atlantic City, 1990.

[17] Scott L. Vandenberg and David J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In James Clifford and Roger King, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 158–167, Denver, CO, 1991.