An Active Object-Oriented Database: A Multi-Paradigm Approach to Constraint Management

Hiroshi Ishikawa and Kazumi Kubota

Fujitsu Laboratories Ltd., Software Laboratory
1015 Kamikodanaka, Nakahara-Ku, Kawasaki 211, Japan

Abstract

We describe the design and implementation of a constraint management facility for our active object-oriented database system called Jasmine/A. The facility includes integrity constraints, events/triggers, and constraint rules. The facility enables the user to handle both interobject and intraobject constraints, to define both primitive and composite events, and to populate databases with values satisfying specified constraints. We have taken a multi-paradigm approach to constraint management. All the paradigms are integrated into object-oriented databases. We describe the semantics of the constraint management facility by extending the conventional terms of transactions and consistency. Evaluation is done efficiently using page buffers for constraints associated with set-oriented access and object buffers for those associated with individual object access. Users are also able to control the constraint evaluation.

1. Introduction

We developed a prototype object-oriented database system called *Jasmine* [ISHI91][ISHI93] for advanced applications such as engineering design support and structured document management. Such advanced applications require more active functions than are needed by conventional applications. For example, we must not only model complex structures and relationships of design objects, but we must also handle design constraints as design specification and geometric constraints between components. Some constraints can be reduced to a collection of constraints on single object attributes. Other constraints inherently span several object attributes. Since we must represent both types of constraints, we discuss *constraint management* as a new database technology to attain this goal.

Constraint management includes integrity constraint enforcement, an event/trigger mechanism to propagate updates, and constraint rules to generate values satisfying specified constraints. We call databases with such constraint management facilities *active databases* by extending the original term [MORG83]. There have been attempts to generalize the event/trigger mechanism such as HiPAC [MCCA89] and SAMOS [GATZ91]. Systems such as Postgres [STON90] and Starburst [LOHM91] aim to extend relational databases by introducing production rules. Works on derived attributes and integrity constraints in object-oriented databases include Cactis [HUDS89] and ODE [AGRA89]. We developed a research system called *HyperCAD* [ISHI91], which supports engineering tasks by using Jasmine to implement the constraint management facilities.

We propose a general framework for constraint management based on the experiences of engineering applications. From this viewpoint, there are some problems with the previous approaches. First, conventional production rules are weak both in generating values which satisfy specified constraints such as design constraints and in describing complex events such as design processes. Such facilities are essential to improve reliability and productivity in design. Second, a single-paradigm approach is not always better than a multi-paradigm approach for representing a variety of purposes such as integrity constraints, triggers, and constraint rules. Third, the semantics of active databases is not so clear in terms of transactions and consistency. Lastly, functions, in particular, triggers and constraint rules must be implemented efficiently. High performance is vital for engineering applications. We have implemented an active object-oriented database system called *Jasmine/A* (Jasmine Active database system) by extending Jasmine, a kernel object-oriented database system.

In this paper, we describe the constraint management facility of Jasmine/A and its implementation. The facility includes integrity constraints, events/triggers, and constraint rules. We take a multi-paradigm approach to constraint management. We describe the semantics in terms of transactions and consistency. Evaluation is done efficiently using page buffers

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

for constraints associated with set-oriented object access and object buffers for those associated with individual object access. Users are also able to control the constraint evaluation. The paper is organized as follows. Section 2 describes active database issues and compares Jasmine/A with related work. Section 3 briefly describes Jasmine, Section 4 discusses our constraint management facility and its semantics, and Section 5 describes its implementation.

2. Requirements and Related Work

2.1 Requirements

Our work aims to satisfy the following requirements. First, constraints must be maintained for object attributes. Such constraints include those on single attributes and those which span multiple attributes of one or more objects, where the latter cannot be reduced to a collection of the former. Second, the system must not only provide direct support for generic integrity constraints, such as mandatory and multiple constraints, but the system must also allow the user to specify application-specific constraints. Third, the constraint management facilities must include check and enforcement of constraints on attributes associated with value updates as well as attribute value generation satisfying user-specified constraints, that is, *automatic database population* based on constraint rules. In particular, such automatic database population is important for engineering applications.

Next, we must provide the user with multiple paradigms which are appropriate for all of these purposes. The appropriate paradigm must be available for each facility. These paradigms must be naturally integrated into the object-oriented databases which we take as a basic framework. The semantics of constraint management must be described in terms of transactions and consistency, requiring more generalized concepts of transactions and consistency. Lastly, we must efficiently implement the constraint management facilities. In particular, we must carefully trade off the expressive power of constraint management against the performance. Since the system cannot know all the information available for optimization, the user must also be able to explicitly control the methods of constraint evaluation, making up for any limitations in the system. From these points of view, we describe Jasmine/A in this paper.

2.2 Related Work

Systems such as HiPAC [MCCA89] and SAMOS [GATZ91] attempt to integrate active database concepts into object-oriented databases. HiPAC introduces the event (E)-condition (C)-action (A) paradigm. In addition to primitive events, the user can define time events and composite events. The E-C and C-A couplings can take immediate, deferred, and separate as evaluation modes. HiPAC rules mainly provide support for a trigger mechanism, but unlike Jasmine/A, provides no direct support for constraint rules to automatically populate databases. Like HiPAC, SAMOS allows time and composite events based on the E-C-A paradigm. It provides E-C and C-A couplings which use the three evaluation modes. SAMOS focuses on triggers, but not on constraint rules.

Systems such as Postgres [STON90] and Starburst [LOHM91] aim to add production rules to extend relational databases. Unlike Jasmine/A, Postgres rules provide a unified approach to integrity constraint checks, update propagation, and view facilities. Postgres does not, however, provide any facility for automatic database population based on constraint rules. In general, there are two methods for rule evaluation: forward and backward chaining. Postgres uses optimization to choose between them. Production rules are implemented at tuple and query levels, which correspond to Jasmine/A's evaluation of constraints on object buffers and on page buffers. Postgres does not provide composite events. Examining Starburst, it also provides for production rules. Events are insertion, replacement, and deletion of values. Conditions and actions can take queries specified in extended SQL and database commands. Alert, a Starburst subsystem, allows the user to define views as a kind of production rules. Starburst provides no facility for composite events or constraint rules.

ODE [AGRA89] [GEHA92] provides integrity constraints and triggers as separate functions of object-oriented databases. Cactis [HUDS89] aims at active object-oriented databases. Cactis proposes rule evaluation schemes based on the dependency relationships of method (rule) definitions. ODE provides composite events, but Cactis doesn't. ODE and Cactis provide no direct support for constraint rules.

3. Overview of Jasmine

3.1 Functionality

To describe Jasmine's object model, objects are a collection of *attributes*, which are categorized into *properties (enumerated attributes)* and *methods (procedural attributes)*. Properties are object structures and methods are operations on those objects. Objects are categorized into *instances* and *classes*. Instances denote individual data and classes denote types (i.e., structures) and operations applicable to instances of the class. Instances consist of a collection of attribute names and values. Classes consist of attribute names, definitions, and associated information such as demons. Objects are identified by values of the system-defined attribute *object identifier* (OID). Therefore, objects with the same object identifier in a consistent database

have the same values. On the other hand, while values such as numbers and character strings have no OIDs, they do have system-defined classes. Objects with OIDs are called *reference objects* and values with no OIDs are called *immediate objects*. Objects can include other objects (i.e., OIDs) as attribute values. This enables the user to directly define *complex objects* (*composite objects*) [KIM90].

Classes are organized into a hierarchy (more strictly, a lattice) by *generalization* relationships. This hierarchy is called a *class hierarchy*. A superclass in a class hierarchy is denoted by the system-defined attribute, *Super*. Classes (i.e., *subclasses*) can inherit attribute definitions from their superclasses. Unlike the features provided for in Smalltalk-80 [GOLD83], the user can make instances (i.e., *instantiate*) from any class in a class hierarchy. Such instances are called *intrinsic instances* of the class.

In addition to defining object types and methods, classes are also interpreted as sets of instances. That is, the instances of a class are the union of all the intrinsic instances of the class and all its subclasses. This differentiates Jasmine from other OODBs such as GemStone [MAIE86] where the user must define separate classes both as a type and as a set. Objects can have a set of objects or just a single object as an attribute value. The former are called *multiple-valued attributes* and the latter *singleton-valued attributes*. Specialized functions, called *demons*, can be attached to attributes to enable the user to flexibly implement active databases.

In Jasmine, the user manipulates objects by sending messages to objects just as in object-oriented programming languages. This is called *singleton access*. The user can assign values to attributes and reference attribute values. Jasmine allows *set-oriented access* in addition to singleton access. Set-oriented access is done by object queries. The basic unit of an object query is an *object expression*, a class name followed by a series of attribute names delimited with periods. Object expressions eliminate most of the need for equijoin predicates in relational databases. The user can also specify methods in object expressions. An object query consists of a target and a condition. The target part is a list of object expressions. The condition part is a logical combination of simple conditions comparing object expressions by comparison operators. For example, the following query finds the name of coworkers of an employee who works in the shoe department and is over 30 years of age:

```
EMP.Coworkers.Name where EMP.Dept = "shoe"
                    and  EMP.Age > 30
```

To make application programs, users can combine singleton-access and set-oriented access. An element of a set of objects is assigned to an *object variable* and is manipulated by sending messages to the object variables. The introduction of object variables reduces *impedance mismatch* between a programming language and a database language [MAIE86]. As users can specify object queries as well as simple manipulation of attributes in methods, virtual objects can be defined using methods. Since a query on a class returns all the instances of the class and its subclasses, a single Jasmine query can retrieve what would take multiple relational database queries to retrieve. By specifying methods in a query, users can retrieve and manipulate objects in a set-oriented manner. If a superclass is specified with a method in a query, methods dedicated to instances of the class and its subclasses can be invoked simultaneously. This facilitates *polymorphism* [STEF86] in a set-oriented manner. A query can also make new instances from more than one class like joins in a relational database.

Application programs written with Jasmine are precompiled into C programs. During this process, references to attributes are statically resolved to reduce the burden of a dynamic search, allowing the C programs to execute efficiently. A set-oriented query can also be interpreted interactively. Although objects are basically persistent, since they exist over program execution, though users can make temporary objects as in conventional programming languages, which exist only during program execution. A Jasmine database usually consists of several classes and users can access several databases concurrently or switch among them. Jasmine provides basic database facilities such as transaction management.

3.2 Implementation

The Jasmine system has a layered architecture consisting of *object management* and *data management* (See Figure 1). The object management layer allows modeling and manipulation of objects. In particular, this layer has *object buffers* that efficiently manage objects in main memory. The data management layer allows transaction management and page buffer management as database functions.

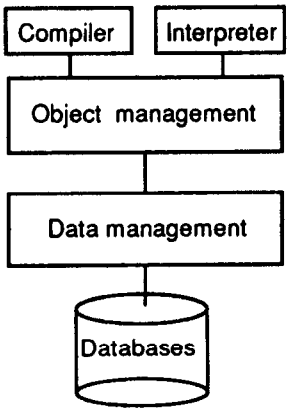

Figure 1. System architecture.

The data management layer is a general-purpose database management system that extends relational databases [YAMA89]. This enables the user to define and access *nested relations* [ROTH88] as well as flat relations. This layer provides *nest* and *unnest operations* for relations, in addition to reference and update. The system provides sequential, B-tree-based, and hash-based access to both flat and nested relations. A *clustered index* can be implemented by storing whole tuples into B-tree relations. A *nonclustered index* can be implemented by storing only keys and tuple identifiers into B-tree relations.

Objects are mapped into relations as follows. All intrinsic instances of a class are stored in a relation by having attributes correspond to fields. Intrinsic instances include inherited and non-inherited attributes. Multiple values are stored in multiple-valued fields, the simplest form of nested relations. Classes are stored in nested relations because they have nested structures.

The user can specify logical page sizes for each relation. Each class has its own page size. A class normally inherits the page size of its superclass. If necessary, however, the page size can be enlarged. There is no limitation to the number and length of tuples and fields although whole tuples must be contained in one page. This enables the user to optimally store and access large-scale data such as images. Operations and tests on fields of relations are treated as user-defined functions in the data management layer, called *manipulation* and *predicate functions*, and compiled into operations on data in page buffers.

Object queries are translated into relational operations such as selection and join. During this process, they are optimized. Object expressions generate several joins whose execution order is determined dynamically. Joins are usually processed based on hashing. If an index is attached to fields, it is used for selection and join.

Page buffers are appropriate for access to homogeneous data, but inappropriate for access to related heterogeneous data such as complex objects. Therefore, the object management layer provides object buffers. Objects, when accessed for the first time, are fetched from databases in secondary memory to page buffers in the data management layer. Only the required data comes to the object buffers from the page buffers. Object identifiers are represented as a triplet of database, class, and instance numbers. The identifiers of objects fetched into object buffers are translated into addresses in main memory. This eliminates the need for joins of relations and enables direct access of complex objects. The objects in object buffers also have tuple identifiers. If there are any updated objects in the object buffers, they are written back to the page buffers using the tuple identifiers at the end of the transaction.

Before a set-oriented query is evaluated, any updated objects associated with the query, that are in the object buffers are moved to the page buffers. The query is then evaluated against the page buffers. Unlike Jasmine, Orion [KIM90] evaluates the same query for both object buffers and page buffers and integrates the results. Because our approach needs only one evaluation scheme, the system is more compact.

A query on nonleaf classes in a class hierarchy is translated into multiple queries on relations. Simple methods specified in a query, such as manipulation of attributes, are transformed into operations on fields of relations. These can be executed more efficiently on page buffers because unnecessary data transfer between page and object buffers is reduced. On the other hand, more complex methods, such as manipulation of heterogeneous objects of complex objects, are more efficiently evaluated in object buffers. Methods appearing in the condition part are similarly processed. Unlike other OODBs, Jasmine efficiently executes methods by combining object and page buffers.

4. Constraint Management

4.1 Integrity Constraint

We now describe integrity constraints supported by Jasmine/A. Basically, constraint management is modeled on the E-C-A paradigm. We realize integrity constraints by Jasmine *demons*, which are essentially the conditions and actions described in Section 4.2. First, we provide the system-defined demons for properties such as *mandatory* and *multiple*. They take the following syntax:

ClassName
  PropertyName mandatory
  multiple

If a value is inserted into a mandatory property at instantiation, that is, if *insert* and *instantiate* events occur at the same time, then the instantiate event is successful. Otherwise, the instantiation is aborted. If a value is deleted from a mandatory property, that is, a *delete* event occurs, then an error is induced. Of course, the replacement of a mandatory property value, or a *replace* event, is possible.

If a new value is added to a multiple-valued property, then the insertion is successful. The insertion to a single-valued property is prohibited when the property already has a value. Of course, the replacement of a singleton-valued property is possible. All the events, conditions, and actions associated with mandatory and multiple demons are system-defined.

Next we describe constraint demons. The demon has the following syntax:

ClassName
PropertyName constraint {*condition* }

If the user-defined condition is true when an insert or a replace event occurs, the transaction for the event is committed. Otherwise, the transaction is aborted. The event and action in this case are system-defined while only the condition is user-defined. The condition is specified by a subset of the query language, that is, the condition part of a query. For example,

EMP
Dept mandatory
Coworkers multiple
Age constraint {Value >= 15}

where the variable *Value* is bound to the value being inserted or replaced. The constraint demon is mainly for expressing user-defined constraints on single attributes, that is, *intraobject constraints*. Constraints spanning multiple attributes, that is, *interobject constraints*, are supported by triggers and constraint rules as described later.

4.2 Triggers and Primitive Events

We describe our basic event and triggering mechanisms called demons. Jasmine/A allows the user to define demons for properties as follows:

ClassName
PropertyName if-referenced {*if-referenced_demon* }
if-inserted {*if-inserted_demon* }
if-deleted {*if-deleted_demon* }
if-replaced {*if-replaced_demon* }

Property demons such as if-referenced, if-inserted, if-deleted, and if-replaced demons modularize user-defined conditions and actions as follows:

if *condition* then *action*
else if *condition* then *action*
...

The condition syntax corresponds to that of the condition part of a query. The conditions, however, can access database *transitions* as well as database states. The actions include both set-oriented and individual object access. If system-defined events such as reference, insert, delete, or replace occur, the corresponding user-defined demons are invoked. In general, production rules consist of conditions and actions, so demons can represent a set of production rules. Constraint rules for automatic value generation are described in Section 4.4

The system-defined variables *Self* and *Value* can be used in the demon definitions. The variable Self is bound to the instance where the event occurs. The variable Value is bound to referenced, inserted, deleted, or replaced values depending on the events. An existing value before replacement is bound to the variable OldValue.

Methods can also take the following demons:

before_demon
method
after_demon

Invocation of user-defined methods correspond to user-defined events. Method demons, that is, before and after demons also modularize user-defined conditions and actions like property demons. Before demons are invoked before the main methods; after demons are invoked after. Usually, before demons are used to check or establish the preconditions of the method invocation. After demons are used to propagate the effects of the method invocation.

The integrity constraints and user-defined demons described above are specified in the class and are activated on its instances where associated events occur. Of course, they are invoked when instances are set-theoretically retrieved, inserted, deleted, replaced, or accessed with methods invocation. If there are multiple rules, that is, pairs of conditions and actions associated with the same event for one attribute, they are usually prioritized by using if-then-else constructs within demons. Note that we presently do not provide support for simultaneous firing of multiple rules.

Objects are accessed through system-defined and user-defined methods. System-defined methods include start, commit, and abort of transactions in addition to instance operations such as reference, insert, delete, replace. In general, method invocation corresponds to event occurrence, so the system can directly recognize the event occurrence. Method invocation corresponds to basic events. The user can combine primitive events to define composite events as described in Section 4.3.

We illustrate the demon functionality by taking some examples used in other work such as Postgres [STON90]. The following demon defined for the Attribute Salary of the class EMP(loyee) specifies the rule that if Joe's salary is updated, the new value is propagated to Sam's salary.

EMP
Salary if-replaced
{if Self.Name = "Joe" then
EMP.replace ("Salary", Value) where EMP.Name="Sam"}

The next demon specifies the rule that every time Joe's salary is referenced, Bill's salary is made equal to Joe's:

```
EMP
  Salary if-referenced
{if Self.Name = "Joe" then
  EMP.replace("Salary", Value) where EMP.Name="Bill"}
```

The following demon specifies the rule that Joe is unable to see Salaries of employees in the shoe department:

```
EMP
  Salary if-referenced
{If Self.Dept = "shoe" and user() = "Joe" then Value = Null}
```

Exceptions to rules can be realized by combining if-then-else constructs within the rules. The following includes an exception for the above rule for Sam:

```
EMP
  Salary if-referenced
    {If Self.Dept = "shoe" and user() = "Joe" then
       { if EMP.Name = "Sam" then Value = 1000
        else Value = Null}}
```

The next rule registers security audits every time somebody references salaries:

```
EMP
  Salary if-referenced
    {<AUDIT>.instantiate("Accessor":user(),
    "Object":Self.Name, "Value":Value)}
```

Note that rules for handling rules themselves can be realized by using event objects as described in Section 4.3.

The Postgres rule system can also provide a view facility within the same mechanism. We realize views as a separate mechanism in *Jasmine/M* (Jasmine Multidatabase system) [ISHI92]. That is, we provide objects for view definition. For example, the following view class defines TOY_EMP as employees in the toy department:

```
TOY_EMP
  BaseClass    EMP
  Property      *
  Method        *
  Condition     Dept = "toy"
```

The "*" entries in Property and Method specify that this view class inherits all the properties and methods of the base class EMP. Condition specifies the filtering condition against the base class EMP. Like this, Jasmine takes a multi-paradigm approach to constraint management because we think there is a separate paradigm well suited for each purpose. Note that all the paradigms including integrity constraints, user-defined demons, and even views are integrated into object-oriented databases. Demons are inherited through a class hierarchy. Polymorphism is also available. If the same event occurs to instances of different classes in a query including multiple classes, each of the demons associated with the same event is invoked. Users can also activate and deactivate demons.

Next we take some examples of Date's Hypothetical Integrity Language [DATE90].

```
S
   Status if-replaced { if Value <= OldValue then
                Self.replace("Status", OldValue)}
          if-inserted  { if S.Status.avg() <= 25 then
                Self.delete ("Status", Value)}
```

The if-replaced and if-inserted demons defined for Status of S(upplier) compensate for the effects of the replace and insert events. Note that the condition of the if-referenced demon is checked against the database transition rather than the database state. Jasmine/A maintains primary key constraints through OIDs. Foreign key constraints are partially maintained by validating object references on object buffers with object descriptors. Thus, constraints with fixed semantics can be elegantly supported by system-defined integrity constraints. The user has only to describe application-specific semantics by specifying user-defined demons.

4.3 Composite Events

Composite event specification extends triggers by combining primitive events described above. The facility enables users to flexibly describe engineering processes, such as design change notification and propagation, and design tool invocation. Composite events consist of one or more primitive events. Primitive events include reference, insert, delete, and replace of attribute values; start, commit, and abort of transactions; and user-defined methods. We provide composing operators such as conjunction (&), disjunction (|), negation(~), and sequence (;). The composite event expressions have the following syntax:

```
event_expression = primitive_event |
                   (event_expression) |
                   event_expression & event_expression |
                   event_expression | event_expression |
                   ~event_expression |
                   event_expression time_spec
time_spec = before time | after time | at time |
            before time after time
time = YMDHMS
```

Assume that E1, E2, and E3 are primitive events; F1 and F2 are composite events; and T1 and T2 are times. For example,

E1 and E2 occur simultaneously:
F1 = E1 & E2
E3 and at least one of E1 and E2 occur simultaneously:
F1 = (E1 | E2) & E3
E1 occurs but E3 doesn't:
F1 = E1 & ~E3
E1 is followed by E2:
F1 = E1;E2
E1 occurs at T1:
F2 = E1 at T1
Both E2 and E3 occurs between T1 and T2:
F2 = E2 & E3 after T1 before T2
E2 occurs after T1 and E3 occurs before T2:
F2 = E2 after T1 & E3 before T2

We describe the timing of evaluation of composite events. The user can specify coupling modes between events and conditions and between conditions and actions in the E-C-A paradigm. The user chooses among immediate, deferred, and separate as coupling modes. The possible combinations of the E-C and C-A couplings are: (immediate, immediate), (immediate, deferred), (immediate, separate), (deferred, deferred), (deferred, separate), and (separate, separate). The immediate mode means that if an event occurs in a transaction, the associated condition or action is immediately evaluated in the same transaction. The deferred mode means that the condition or action is evaluated immediately before the triggering transaction is completed. If there is more than one deferred evaluation in the same triggering transaction, evaluations are processed on a first-come-first-served basis. Separate mode means that the evaluation is done in a transaction other than the triggering transaction. Usually, the deferred mode is specified to evaluate constraints between several attributes of one or more objects, that is, interobject constraints. This is used to ensure the global consistency described later. The separate mode is specified in cases when the triggering event and triggered action are separately evaluated, in instances like a fire alarm and fire fighting.

We do not provide logical events of ODE [GEHA92] because we do not yet have an efficient implementation of general logical expressions for databases. Of course, general logical expressions could be simulated by defining user-defined methods. Instead, we restrict the expressive power of event specification to efficiently process the event evaluation. Similarly, we do not allow regular expressions of composite events because basic events are more data-intensive operations than conventional programming operations and because we have not determined what expressive power is sufficient for database operations. Currently, we have engineering design applications in mind, which do not require the regular expressions or more expressive notations needed by real-time applications.

In general, we assume that the triggering transaction and the separated-mode triggered transaction are independent. There are cases, however, where failure of a triggered transaction requires that the triggering transaction be aborted. Furthermore, if the triggering transaction fires other triggered transactions, this may require additional transactions to be aborted. Such cascade effects are undesirable, however. That is, the unconditional abort strategy based on dependency is particularly unacceptable for engineering applications. Our approach to solve this problem allows the user to specify the transaction compensating for the effects of the committed transactions if necessary. For example, in design applications we can consider the design phase as a triggering transaction and the subsequent drawing phase as the triggered transaction. Even if the drawing phase fails due to some design errors, the whole design phase will never be aborted. Instead, the transaction is compensated to remedy the design.

We provide event objects for primitive events and composite events. Primitive events have the following structure:

PRIMITIVE_EVENT		
STRING	Name	
TIME	Time	
COMPOSITE_EVENT	Composite	multiple
OBJECT	Object	
METHOD	Method	
PROPERTY	Property	
OBJECT	Value	

The attribute Name specifies an event name. The attribute Time denotes the time when the event occurs and Composite denotes composite events whose components include this primitive event. Other attributes are used to record the context where the event occurs.

Composite events have the following structure:

COMPOSITE_EVENT	
STRING	Name
TIME	Time
PRIMITIVE_EVENT	Primitive multiple
EVENT_EXPRESSION	Event
CONDITION_EXPRESSION	Condition
ACTION_EXPRESSION	Action
ACTION_EXPRESSION	Compensation
MODE	E-C-mode
MODE	C-A-mode

The attribute Primitive denotes primitive events which constitute this composite event. Event holds an event expression and Action holds an action expression. The attribute Compensation holds an action expression compensating for the effect of the events. E-C-mode and C-A-

mode attributes denote E-C and C-A coupling modes.

Concrete events are instantiated and named in advance. Primitive events constituting composite events are recorded in demons or methods by the following query:

```
PRIMITIVE_EVENT.insert("Time", time)
where PRIMITIVE_EVENT.Name = event-name
```

For example, to ensure the rule "if Joe's salary is replaced, replacing Sam's salary with the same value is the only way to change Sam's salary," the user defines the following composite events:

```
COMPOSITE_EVENT1
  Event      { ~EVENT1 & EVENT2 }
  Condition  { True }
  Action     { Value = PRIMITIVE_EVENT.Value
               where PRIMITIVE_EVENT.Name = "EVENT2";
               deactivate_demon;
               EMP.replace ("Salary", Value)
                    where EMP.Name = "Sam";
               activate_demon }
  E-C-mode   immediate
  C-A-mode   immediate
```

where EVENT1 and EVENT2 are invoked as follows:

```
EMP
  Salary if-replaced
{if Self.Name = "Joe" then
   { EMP.replace ("Salary", Value) where EMP.Name="Sam" ;
    PRIMITIVE_EVENT.insert("Time", Time)
      where PRIMITIVE_EVENT.Name = "EVENT1"}
 else if Self.Name = "Sam" then
   { PRIMITIVE_EVENT.insert("Value", OldValue)
      where PRIMITIVE_EVENT.Name = "EVENT2" ;
    PRIMITIVE_EVENT.insert("Time", Time)
      where PRIMITIVE_EVENT.Name = "EVENT2"}}
```

Note that deactivate_demon in the composite event action suppresses the invocation of EVENT2 to avoid an infinite loop. Events are first-class objects. The user can use object-oriented facilities such as inheritance and polymorphism to customize the event mechanism.

We conclude this subsection by describing the interpretation of event expressions. We assume the following:

T_E: Time interval specified for the event E.

H_T: History or a set of events during T_E.

P: Primitive event.

E, E1, E2: Events

The interpretation I of event expressions is defined as follows:

$I(P) = \{ P \text{ belongs to } H_T \}$

$I(E1 \mid E2) = I(E1) \mid I(E2)$

$I(E1 \,\&\, E2) = I(E1) \,\&\, I(E2)$

$I(\sim E) = \sim I(E)$

$I(E1 ; E2) = I(E1) \,\&\, I(E2) \,\&\, E1.Time < E2.Time$

4.4 Constraint Rules

We describe constraint rules, a generalization of the design goals whose validity we have verified in engineering applications [ISHI91]. The main objective of integrity constraints and triggers is to check and propagate updates of property values while the main objective of constraint rules is to generate values satisfying the specified constraints. In other words, constraint rules are mainly used to automatically populate databases. Constraint rules enable users to describe constraint conditions on attributes of objects and methods for generating candidate solutions to conditions. They help explicitly describe engineering knowledge such as design constraints. The system determines a collection of database values satisfying the constraints, based on a network consisting of constraint rules and dependency relationships among them. Such automatic database population using constraint rules is vital for establishing high reliability and productivity in engineering design. To our knowledge, there is no work on automatic database population based on constraint rules. The constraint rules are first-class objects with the following structure:

```
CONSTRAINT_RULE
  STRING                Name
  STRING                Parameter multiple
  GENERATE_METHOD       generate
  CONDITION_CLAUSE      condition-action multiple
  INCREASE_METHOD       increase
  DECREASE_METHOD       decrease
```

where

```
GENERATE_METHOD = generate (init, cond, dif ) |
      calculate (exp) | retrieve (db, cond, order ) | ask ()
CONDITION_CLAUSE = condition |
                   condition advice actions
actions = action | actions | action
INCREASE_METHOD = generate_incr | retrieve_incr |
        ask_incr |  rule.increase() | rule.decrease()
DECREASE_METHOD = generate_decr | retrieve_decr |
        ask_decr | rule.increase() | rule.decrease()
```

Name is the rule name denoting the name of the property whose value this rule aims to determine. Parameter denotes the names of other rules on which this rule depends. The generate attribute

specifies methods for generating candidate values satisfying the specified constraints, which include generation based on initial and difference values, calculation based on other rules, database retrieval, and user input. Condition-action consists of zero or more condition and action pairs. Actions for failure advice are invoked when the preceding condition is not satisfied by the candidate value. The actions are invocations of their own or other rules with an increase or decrease message. They are evaluated from left to right. If one action is successful, the following actions are not evaluated. The actions correspond to user-specified backtracking of rules. When there are multiple conditions, the rule is only successful if all the conditions are satisfied. Otherwise, the rule is aborted.

Events in this case are rule invocations such as generate, increase, and decrease. Conditions and advice actions correspond to the conditions and actions in the E-C-A paradigm. While events are system-defined, conditions and actions are user-defined. As described later, the order of rule invocation is determined by the rule scheduler based on rule dependencies. During scheduling, loop detection of rules is done statically. Loop detection is also done dynamically during rule execution.

For example,

```
Name            PistonHeadThickness
Parameter       ExplosionPower  CylinderDiameter
                PistonDiameter
generate    generate (init: 3.7, cond: Value < 3.9, dif: 0.01)
condition-action
  (1) Value > 0.06* CylinderDiameter advice Self.increase()
  (2) Value < 0.065* CylinderDiameter advice Self.decrease()
  (3) power (PistonDiameter, 2) * ExplosionPower /
        power  (Value, 2)< 80.0
      advice  Self.increase() | PistonDiameter.decrease ()
increase        generate_incr
decrease        generate_decr
```

This rule determines the value of PistonHeadThickness dependent on other parameters, such as ExplosionPower and CylinderDiameter, by using the generation method. Three condition-action pairs are specified. The last condition's action for advice in the event of failure specifies disjunctions of actions.

Like triggers, our constraint rules are basically set-oriented. In general, there is more than one combination of parameters satisfying the same set of constraints. However, we don't take the approach where all solutions are automatically generated, because all of them are not always interesting. It is more desirable that the user can modify the initial solution to get an alternative if it is unsatisfactory. In a word, the user must be able to control the exploration of alternatives in a stepwise fashion. In Jasmine/A, the user can modify the initial solution by invoking the constraint rules again with some constraints changed. The user can specify constraints such as fix, increase, decrease, loose fix, and don't care for the current values.

4.5 Semantics

We describe the semantics of integrity constraints, triggers including primitive and composite events, and constraint rules in terms of transactions and consistency. In general, a transaction causes transition from one consistent database state to another consistent database state. Transactions under consideration consist of events, conditions, and actions as follows:

C { event condition action } C'

where C and C' denote consistent database states. First, we consider integrity constraints, such as mandatory, multiple, and constraint. When the event such as insert or instantiate, occurs, the transition from C to C' is committed only if the condition, system-defined or user-defined, is true. Otherwise, the transaction including the triggering event is aborted. As a result, the state C still holds. In case of primitive events such as reference, insert, delete, and replace, the action is invoked within the associated demons to result in the state C' if the condition holds. Otherwise, in case of insert, delete, and replace events, the action compensating for the effect of the event is invoked within the demons. At that time, the resultant state C' is semantically equal to the state C.

These semantics are also true for constraint rules. If the conditions of one rule are satisfied, the transaction establishing the property value as the action of the rule is committed to reflect the event effect to the database. Otherwise, the action compensating for the event effect is committed and another rule is invoked. Note that even then, all the rules are not aborted.

Until now, we have used the term transaction to mean a conventional short transaction. The consistency associated with a short transaction is application-independent, or a local consistency. In general, an application is a sequence of such short transactions. Such an application constitutes a long transaction as a whole. The associated consistency is application-dependent, or a global consistency. The purpose of applications is to establish global consistency. Integrity constraints focus more on local consistency while composite events and constraint rules focus more on global consistency. In particular, there are cases where events, conditions, and actions are separate transactions. That is, there are application-independent consistent states between C and C'. The coupling modes of the E-C-A paradigm are used to specify such cases.

5. Implementation

5.1 Integrity Constraints and Primitive Events

This section describes the implementation of system-defined integrity and user-defined demons (primitive events). For multiple integrity for a property, the following code is embedded into predicate functions of the insert operation of the data management subsystem only if multiple is not specified for the property by the user, that is, only if the property is singleton-valued:

```
if the property is empty, then return True
else return False
```

The insert operation is performed only if the predicate is true.

The insertion to a mandatory property is needed at instantiation of the instance with the property. Deletion is prohibited while replacement is allowed. So the instantiate method checks the insertion to the mandatory properties. The delete method checks the deletion of the mandatory properties. These checks can be done at the object management layer without accessing actual values. The user-defined constraint is embedded into predicate functions of the insert and replace operations of the data management. The insert and replace operations are performed only if the predicate is true.

Next we describe the implementation of user-defined demons. The conditions and actions of demons if-referenced, if-inserted, if-deleted, and if-replaced are compiled respectively into the predicate and manipulation functions of the select, insert, delete, and replace operations in data management. The predicate and manipulation functions are directly evaluated on page buffers. This reduces unnecessary data transfer between application programs and page buffers. This scheme is used for set-oriented access of objects. For individual access of objects, the conditions and actions are evaluated on the object buffers. We provide separate evaluation schemes appropriate for each of the two types of access. The demons for the user-defined methods (events) are also compiled into predicate and manipulation functions. In this way, the conditions and actions associated with the events can be efficiently processed. The detection of events themselves can be also efficiently done. That is, the system can directly detect the occurrences of events, system-defined or user-defined, because they are invoked only through method invocation.

In general, there are two evaluation schemes for triggers or production rules: forward chaining and backward chaining [STON90]. Assume that A and B are attributes of objects and that A is dependent on B. If A is rarely referenced and B is often updated, the if-referenced demon should be specified for A. Conversely, if A is often referenced and B is rarely updated, if-inserted, if-deleted, and if-replaced demons should be specified for B. The former case corresponds to backward chaining and the latter case corresponds to forward chaining. Like this, Jasmine/A allows the user to control evaluation because we assume that the user knows access patterns better than the system does.

5.2 Composite Events

We realize composite events by using methods and demons of objects in a bootstrap manner. To check events for immediate or separate evaluation, the PRIMITIVE_EVENT class has the following demon specified for the property Time:

```
PRIMITIVE_EVENT
  Property
    TIME  Time  multiple
      if-inserted
      { Self.Composite.check_immediate_or_separate ()
        where  Self.Composite.E-C-mode = immediate
          or   Self.Composite.E-C-mode = separate }
```

For deferred evaluation, the system executes the following query before the end of the transaction:

```
COMPOSITE_EVENT.sort("Time", Ascending).check_deferred()
where COMPOSITE_EVENT.E-C-mode = deferred
  or  COMPOSITE_EVENT.C-A-mode = deferred
```

The expression *sort("Time", Ascending)* means that composite events are evaluated in a first-come-first-served manner. It is implemented by embedding the above query in the before demon of the transaction commit method.

The six combinations of the two coupling modes are interpreted by the methods check_immediate_or_separate and check_deferred of COMPOSITE_EVENT, where the spawn_transaction operation creates a new transaction whose execution may be postponed until the triggering transaction commit or abort methods are activated (See Figure 2).

5.3 Constraint Rules

Constraint rule processing is divided into rule analysis, execution planning, and rule execution. Constraint rules are used to determine property values of objects satisfying the specified constraints. Basically, the execution order of rules is determined by the dependency between rules, that is, between properties. The dependency between rules is called a dependency network. The uppermost nodes in the dependency network are properties which are not dependent on other nodes.

```
COMPOSITE_EVENT
  Method
    check_immediate_or_separate()
    { if Self.E-C-mode = immediate then
        {if Self.C-A-mode = immediate then
          {if Self.event_eval() & Self.condition_eval()
          then Self.action_eval()}
        else if Self.C-A-mode = deferred then
          { if Self.event_eval() & Self.condition_eval()
          then Self.Condition_value = True
          else Self.Condition_value = False}
        else if Self.C-A-mode = separate then
          { if Self.event_eval() & Self.condition_eval()
          then spawn_transaction {Self.action_eval()}}}
      else if Self.E-C-mode = separate &
              Self.C-A-mode = separate then
        {if Self.event_eval() then
          spawn_transaction
        { if Self.condition_eval()  then Self.action_eval()}}
      else error() }

    check_deferred()
{ if Self.E-C-mode = immediate &  Self.C-A-mode = deferred
  then
   { if Self.Condition_value = True then Self.action_eval() }
 else if Self.E-C-mode = deferred & Self.C-A-mode = deferred
  then
   { if Self.event_eval() & Self.condition_eval ()
    then Self.action_eval()}
 else if Self.E-C-mode = deferred &
         Self.C-A-mode = separate then
   { if Self.event_eval() & Self.condition_eval()
     then spawn_transaction { Self.action_eval() }}
 else error() }
```

Figure 2. Methods of COMPOSITE_EVENT.

Basically, rule execution or constraint satisfaction is done from top to bottom in the dependency network. If one rule is executed successfully, another rule is fired. One rule corresponds to a short transaction and establishes local consistency. Global consistency is only established as a whole if all the rules corresponding to a long transaction are successful.

Of course, candidate values do not necessarily satisfy the constraint condition initially. That is, the backtracking is usually needed for constraint satisfaction. When backtracking occurs, the effects of unsuccessful rules are compensated and alternative rules are invoked. The backtracking method is specified in the advice part of the constraint rules by the user. Loop detection is done statically during dependency network development. As is described later, dynamic loop detection is done during rule execution. For generality, constraint satisfaction is based on the generate and test scheme.

(1) Rule analysis

The system determines the level of rules according to the dependency between rules. Rules at level 1 depend on no other rules. Rules at level 2 only depend on level 1 rules. In general, rules at level n depend on at least one rule at level n-1. We can detect loop dependencies by using developed and undeveloped rule lists. Initially, the undeveloped rule list contains all rules at level 1 and the developed rule list contains no rules. The rules at level 1 are developed into the rules dependent on the rules at level 1. The developed rules are put in the developed rule list. The dependent rules are put in the undeveloped rule list. Similarly, the rules taken from the undeveloped rule list are developed into the rules dependent on the taken rules. If, during rule analysis, the same rule appears both in the undeveloped and developed rule lists at the same time, a loop occurs. This is brought to the user's attention for further processing.

(2) Execution planning

After the levels of rules are determined, the rules are grouped into disjoint sets of related or connected rules. Within one rule group, the rules are ordered according to increasing level. If there is more than one rule with the same level, order is determined based on the number of rules upon which the rules depend. That is, the smaller the number, the higher the execution priority. In the final step, the disjoint sets of rule groups are merged into a linear list.

(3) Rule execution

A plan is a list of rules pushed onto the stack. Individual rules are popped from the stack. A candidate value is generated through the generate method of the constraint rule. If a pattern of partially determined values including the candidate value is already in the history hash table, a rule execution loop occurs. That is, the system can detect the loop dynamically. In other words, the system can guarantee the termination of rule execution. If the pattern is not in the history table, then the constraint condition is evaluated. If the condition is satisfied, then all the advice actions associated with this rule are popped off the stack. Otherwise, the advice action of the condition is pushed on the stack. If all the conditions are satisfied, the dependent rules are pushed on and the control is given to the beginning of this process.

When the user has specified alternative solutions or alternative sets of values, the system modifies the constraints according to these specifications. For example, if the user specifies "Increase the current value", the condition "Value > CurrentValue" is inserted to the constraint rules. The execution of rules is done in the same order as the initial plan.

6. Conclusion

We have described the design and implementation of the constraint management facility for our active object-oriented database system called Jasmine/A. The facility includes integrity constraints, events/triggers, and constraint rules. The facility enables the user to handle both interobject and intraobject constraints, to define both primitive and composite events, and to populate databases with values satisfying specified constraints. We have taken a multi-paradigm approach to constraint management. All the paradigms are integrated into object-oriented databases. We have described the semantics of the constraint management facility by extending the conventional terms of transactions and consistency. Evaluation is done efficiently using page buffers for constraints associated with set-oriented object access and object buffers for those associated with individual object access. Users are also able to control the constraint evaluation.

We plan to apply Jasmine/A to various practical applications to verify the validity of our approach and give experience feedback to the system. We also plan to include enhancements such as extension of composite event specification, extension of constraint rule description, and the addition of a graphical user interface.

Acknowledgments

We thank the anonymous referees for their helpful suggestions.

References

[AGRA89] Agrawal, R., et al.: ODE: The language and the data model, Proc. the 1989 ACM-SIGMOD Conference, pp.36-45(1989).

[DATE90] Date, J.C.: An Introduction to Database Systems, vol. 1, Addison-Wesley, 1990.

[GATZ91] Gatziu, S., et al.: Integrating Active Concepts into an Object-Oriented Database System, Proc. the 3rd International Workshop on Database Programming Languages (1991).

[GEHA92] Gehani, N.H., et al.: Event Specification in an Active Object-Oriented Database, Proc. the 1992 ACM-SIGMOD Conference, pp. 81-90 (1992).

[GOLD83] Goldberg, A., et al.: Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, MA., 1983.

[HUDS89] Hudson, S., et al: Cactis: A Self-Adaptive, Concurrent Implementation of An Object-Oriented Database Management System, ACM Trans. Database Syst., vol. 14., no.3, pp.291-321(1989).

[ISHI90] Ishikawa, H.: An Object-Oriented Knowledge Base Approach to a Next Generation of Hypermedia System, Proc. IEEE COMPCON Spring 90 Conference, pp. 520-527 (1990).

[ISHI91] Ishikawa, H., et al.: An Object-Oriented Database: System and Applications, Proc. the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing, pp.288-291 (1991).

[ISHI92] Ishikawa, H., et al.: An Object-Oriented Database System and its View Mechanism for Schema Integration, Proc. the 2nd Far-East Workshop on Future Database Systems, pp.194-200 (1992).

[ISHI93] Ishikawa, H., et al.: The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System, ACM Trans. Database Syst., vol.18, no.1, pp.1-50 (March 1993).

[KIM90] Kim, W., et al.: Architecture of the ORION Next-Generation Database, IEEE Trans. Knowledge and Data Engineering, vol. 2, no.1, pp. 109-124 (1990).

[LOHM91] Lohman, G., et al.: Extensions to Starburst: Objects, Types, Functions, and Rules, Comm. ACM, vol.34, no.10, pp.94-109 (1991).

[MAIE86] Maier, D., et al.: Development of an object-oriented DBMS, Proc. the 1st OOPSLA Conference, pp. 472-482 (1986).

[MCCA89] McCarthy, D., et al.: The Architecture of An Active, Object-Oriented Database System, Proc. the 1989 ACM-SIGMOD Conference, pp.215-224 (1989).

[MORG83] Morgenstern, M.: Active Databases as a Paradigm for Enhanced Computing Environments, Proc. the 9th VLDB Conference, pp. 34-42 (1983).

[ROTH88] Roth, M. A., et al.: Extended Algebra and Calculus for Nested Relational Databases, ACM Trans. Database Syst., vol.13, no.4, pp.389-417 (Dec. 1988).

[STEF86] Stefik, M., et al: Object-Oriented Programming: Themes and Variations, AI MAGAZINE, vol.6, no.4, pp.40-62 (winter 1986).

[STON90] Stonebraker, M., et al.: On Rules, procedures, caching and views in database systems, Proc. the 1990 ACM-SIGMOD Conference, pp.281-290 (1990).

[YAMA89] Yamane, Y., et al.: Design and Evaluation of a High-Speed Extended Relational Database Engine, XRDB, Proc. International Symposium on Database Systems for Advanced Applications, pp.52-60 (1989).