Object Integrity Using Rules

Claudia Bauzer Medeiros*
DCC-IMECC-UNICAMP-CP 6065
13081 Campinas SP
Brazil
cmbmedeiros@dcc.unicamp.ansp.br

Patrick Pfeffer†
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
U.S.A.
patrick@cs.colorado.edu

Abstract

Integrity maintenance in object-oriented systems has so far received little attention. This paper is an attempt to fill this gap. It describes a mechanism for maintaining integrity in an object-oriented database, implemented for the O_2 system, and which uses the production rule approach to constraint maintenance. Object integrity is ensured by objects themselves – the *rules* which are activated when selected events take place. The approach presented is original, in the sense that it takes full advantage of the object-oriented paradigm, considering constraints as first-class citizens which can be inherited, and defined independently of any application. Furthermore, we support maintenance of not only static but also some types of dynamic constraints, as well as constraints on object behavior.

1 Introduction

As remarked by [JMSS89], a great part of the properties that define the consistency of a database can be represented by predicates on the database state (the so-called *static integrity constraints*). If all these predicates are evaluated as true for a given state, then the state is consistent. Most of the work published on database constraint maintenance is dedicated to ensuring this type of constraint.

Dynamic integrity constraints, on the other hand, are predicates specified over a *sequence of states*. Research in this area goes in the direction of transforming each dynamic constraint into a set of static constraints (e.g., [Via88]) sometimes using state transition graphs (e.g., [Kun85, Lip88]) where the terminal nodes are consistent states. Again as remarked by [JMSS89], a considerable amount of dynamic constraints can be expressed in terms of the initial and final states of a transaction. A *transaction* is a sequence of actions conducting from one initial (or input) database state to a terminal (output) state. Dynamic constraints that can be expressed in terms of only initial and final states of a single transaction are called *two-state predicate constraints*. The monitoring of other dynamic constraints is still a matter of research, and demands maintaining historical information on the database states ([HS90]).

Static constraints are usually expressed by means of first order logic expressions, and dynamic constraints by means of *temporal logic*. Temporal logic extends first order logic by incorporating modal operators *always*, *sometime* and *next*. These operators may vary, since authors who do research in the area define their own operators to better express dynamic constraints.

*This research was developed while the author was on sabbatical leave at GIP Altaïr, BP 105, 78153 Le Chesnay-Cedex, France. The support of this research was provided by GIP Altaïr and by grant CNPq - Brazil - 200.168-89.4

†This research was partially developed while the author was working for Gip Altaïr. The support for this research was provided by GIP Altaïr and by ONR under Contract N00014-88-K-0559

Most mechanisms for constraint enforcement support only a very limited set of constraints. This support consists usually in either forbidding or rolling back operations that violate constraints. A more flexible approach to enforcement is that of performing *compensating actions*, whereby the once forbidden operations are allowed to take place, followed by other operations which re-establish consistency.

Integrity maintenance in object-oriented systems is still an unexplored topic. This paper presents a solution to this problem, which has been implemented on the O_2 database system, and which can be generalized to other object-oriented database systems. The constraint maintenance mechanism takes advantage of the production rule subsystem which has been integrated into the O_2 prototype [Da90]. Though this rule mechanism is general, this paper is only concerned with the features relevant to the consistency problem. The reader is referred to [BMP91] for a more detailed description of rule treatment in O_2.

The key issues discussed here are based on the following:

- **Constraints supported.** The mechanism supports the maintenance of any constraint that can be expressed as a sequence of predicates on sequences of database states using the O_2 query language. This means that we allow static constraints and some classes of dynamic constraints. For dynamic constraints, we restrict ourselves to cases where state history can be checked in terms of one state transition (i.e., we support two-state predicate constraints). We do furthermore consider constraints on object behavior.

- **Integrity enforcement.** It is ensured by performing compensating actions, determined by database and application semantics. It uses the production rule solution to constraint maintenance.

- **Exploiting the object-oriented paradigm.** Our mechanism takes advantage of the object-oriented paradigm in three aspects. First, each constraint is transformed into a set of special objects, called *rules*, which will monitor integrity. Thus, object integrity is supported by objects themselves. Second, the system takes inheritance into account, and a constraint defined for a given class is automatically enforced in all its subclasses. Third, constraints can be inserted, deleted and modified at will, independent of any application. They are considered first-class citizens, and need not be encoded in the body of any application. The last point is an answer to the remark in [fADF90], where it is claimed that this independence cannot be supported in object-oriented systems.

The rest of this paper is organized as follows. Section 2 presents an overview of the O_2 system. Section 3 presents a brief description of the O_2 rule system. Section 4 points out some of the problems of constraint maintenance in an object-oriented environment, presents our framework and gives an overview of research in the area. Section 5 shows how the rule system is used to maintain constraints. Section 6 gives examples of constraint support. Finally, Section 7 contains conclusions and directions for future work.

2 A brief overview of the O_2 system

This section contains a short presentation of the O_2 object-oriented database system. The interested reader will find further detailed information in [Da90].

The O_2 data model [LR89] relies on two kinds of concepts: *complex values*, defined as in standard programming languages, and *objects*. Objects are instances of classes, and values are instances of types. Objects are encapsulated (i.e., their value is only accessible through methods), whereas values are not – their structure is known to the user, and they are manipulated by primitive operators. Manipulation of objects is done through *methods*, which are procedures attached to the objects. Objects sharing structure (*type*) and behavior (*methods*) are grouped into classes. Users can define *names* for given objects.

Types are constructed recursively using the O_2 atomic types (e.g., integer or bitmap), classes from the schema and the *set, list* and *tuple* constructors. An O_2 *schema* is a set of classes related by inheritance links and/or composition links, as well as the attached methods, and allows multiple inheritance.

Though O_2 is multi-language, the methods discussed here are coded in the CO_2 language. CO_2 is a C-like programming language which allows (i) class declaration, (ii) object manipulation (i.e., message passing) and (iii) value manipulation by means of primitive operators.

The O_2 system has a functional first order query language [BCD89] which can be used either in an interactive mode or included in the body of methods. This language supports access to O_2 structures, thus being able to manipulate O_2 constructors and make use of methods. The result of a query is a complex value whose type is defined by the query itself. The result of a query can be used in a program as any complex value.

3 The O_2 rule mechanism

3.1 General concepts

Active databases (e.g., [DBB+88]) are database systems that respond automatically to events generated internally or externally to the system itself, without user intervention. The nature of the response is arbitrary, and depends on application semantics. The desired behavior is commonly specified by *production rules*, which are pairs of the form < predicate → action > to be triggered at specific events.

Our solution to integrity enforcement uses the active database paradigm, where a constraint is enforced by means of *O_2 production rules*. This section gives a brief overview of the rule mechanism in O_2. A complete description of this mechanism is given in [BMP91].

Rules are considered to be schema components and can be kept independently from applications on the database. Furthermore, the inheritance property applies – i.e., rules defined for a given class are inherited by all of its subclasses. Like any other schema component, rules also suffer evolution: they can be added, deleted and modified. They differ from other schema components in the sense that they can be enabled or disabled at times, which does not apply to the other components. Some rules may be local to one transaction (which creates them at its beginning and deletes them at its end).

Rules are implemented as objects, with priorities and access rights, and are instances of O_2 builtin classes, whose root is the class *Rule*. Following the terminology of [DBB+88], their activation occurs upon some *Event*, in which case a *Condition* is checked and some *Action* is optionally undertaken.

3.2 Rules as objects

The O_2 rule system is not restricted to verifying integrity constraints. Whereas most active systems restrict events to be update requests, in O_2 they may be associated with *message sending* or with the *passing of time*.

Rule objects are tuples <**Name**, **E**, **Q**, **A**, **P**, **S**, **AP**>, where
Name is a string that identifies the rule;
E(vent): is an expression describing one event which triggers rule verification;
Q(uery): is an O_2 query. It contains the predicate to be tested in order to execute the action;
A(ction): is a sequence of CO_2 operations and corresponds to the action to be performed if the condition is met. It may itself involve operations which will trigger further rules, in nested execution;
P(riority): is an integer that ranks the rule, to be used when there is more than a rule applicable for a given event;
S(tatus): indicates whether the rule is enabled or disabled;
AP(plicability): indicates when to check the rule, e.g., pre- or post-method execution.

Message-related events are expressed as [*Receiver, Method*] – i.e., the event is signalled when there is a request for *Method* to be executed on *Receiver*. *Receiver* can be either an object name or a class name. Time-related events are specified as *TIME(value)*. Accepted time values are those of the O_2 builtin classes Date and Duration (which respectively allow expressing points in time or time intervals in years, months, days, hours, minutes and seconds). An event may trigger the verification of a set of rules, according to their priority. Verification of a rule's applicability corresponds to performing query **Q** followed by the arbitrary action **A** which is itself an O_2 method.

Rules can be examined or updated. Rule operations are methods attached to the *Rule* builtin class hierarchy. They can be invoked inside an application program or interactively using the graphical user interface. The update operations on rules are Add, Delete, Enable, Disable, Fire and Change_priority.

4 Constraints in an object-oriented DBMS

4.1 Related work

The analysis and support of constraints for object-oriented systems has so far merited little attention and has been restricted to static constraints. Most of the integrity maintenance problems already existing in the relational world can be translated into equivalent (or often the same) problems when one comes to the object-oriented model. However, the object-oriented model has brought an additional type of constraint into existence: constraints imposed on the *behavior* of objects (i.e., controlling method definition and execution). Some systems (e.g., [KGBW90]) support constraint maintenance for schema (structural) update operations; others (e.g., [DBB+88, KDM88]) describe constraint maintenance in the framework of active databases, but without details about types of constraint supported or their transformation into rules.

Recent research in constraint maintenance in relational and logic databases uses the active database approach, and is usually restricted to static constraint support. Examples are the constraint equations of [Mor84] (expressed as path expressions over relations), the POSTGRES rule system [SJGP90] (where rules are used to define views, compute aggregate fields and translate update requests), or the work of [SLR89] (concerned with different algorithms for improving the checking of conditions upon one-tuple relational updates). [CW90] describe a framework in which constraints specified in an SQL-like language can be translated into rules that detect integrity violation. Logic database rule based systems include PRISM [SK84] and TAXIS [MBW80].

In the object framework, [LR90] suggest how static constraints might be stated in a database programming language, to be checked as post-conditions to a transaction. Production rules are suggested for constraint maintenance in [DBB+88, KDM88]. In [DBB+88], ECA (Event Condition Action) rules are supported, where Event triggers the rule, Condition is a collection of queries evaluated when a rule is triggered by an Event, and Action is a sequence of database and application program operations. [KDM88] describes an extended trigger mechanism on top of the DAMASCUS system which has three components: Event, Action and Trigger. The Trigger is the means of connecting an Event with a given Action. Events have pointers to Trigger chains, organized according to Trigger priority.

Constraints and rule execution in object-oriented environments are also found in the CACTIS system [HK89] where derived attribute instances are defined in terms of triggers, and updated only on request. The work of [UD89] proposes specification and maintenance of integrity constraints as rules in an NF2 system, where constraints are defined on the schema or for specific application needs. Finally, in [NQZ90] a rule system for semantic modelling is implemented on top of Gemstone, allowing the handling of some types of static constraints.

4.2 Constraints supported in O_2 - assumptions and terminology

We support both static and two-state transition constraints. This section describes the framework of this support.

In the relational model, constraints are often classified into intra-relation (e.g., functional dependencies) or inter-relation (e.g., referential integrity). Inter and intra-relation constraints exist both at the static and the dynamic level. Analogously, we support constraints that are defined on the objects of one class (*intra-class*) or several classes (*inter-class*). We consider constraints between objects as a special case of intra-class constraint. We also support constraints defined on object behavior, as long as they can be stated as pre or post conditions to methods. Finally, constraints can be *global* – i.e., they hold for all applications that run on a database; or *local* – they are defined locally to an application, and their verification is only enabled when the application is active.

Another issue we consider is that of *flexibility*. In commercial database systems, constraint enforcement depends on the programmer adding the appropriate integrity checks throughout all applications that use the database. This type of enforcement requires a huge maintenance effort, not only to correct errors, but also to accompany evolution of the database or the applications. Furthermore, one would like flexible constraint management, allowing enabling and disabling of constraint verification.

In order to respond to this need for flexibility, we define constraints to be first-class citizens. They are conceptually considered to be *properties* of an O_2 database schema and can be kept independently from applications on the database. They may be defined over classes, methods or named objects, as well as as over *computed* or *aggregate* values, that is, values that are not effectively stored.

We handle both static constraints (expressed as first order logic predicates on a state) and some dynamic constraints (those that can be stated as two state predicates). Two-state constraints are formalized in [CCF82], where they are stated in terms of transactions that lead from an input $State_i$ to an output $State_o$, and where the predicate is transformed into a first order expression.

We thus deal with situations involving just first order logic predicates: static constraints, constraints on input and output states of methods, and two-state constraints. In the latter case, we use [CCF82]'s technique to transform the constraint into a transaction where predicates are to be verified as pre and/or post conditions.

We take the remedial approach to integrity maintenance. Thus, rather than defining constraints as assertions over database states, they are specified as *production rules*. These rules are implemented using the O_2 rule system and the predicates are expressed within O_2 queries: first order logic expressions, using methods, composition and iterators.

A constraint statement is transformed in a production rule expression as follows. Consider first static constraints, stated as first order logic predicates $\mathcal{P}$. They give origin to rule statements of the form $< \neg \mathcal{P} \rightarrow \mathcal{A} >$ for some designer-defined action $\mathcal{A}$. If one assumes the state of a database is consistent before any operation that may violate a static constraint, then this constraint need only be checked after a state change.

The dynamic constraints supported are expressed using modalities "sometime" and "always" (see [CCF82])

sometime $\mathcal{P}$i *before* Transaction
sometime $\mathcal{P}$o *after* Transaction
always $\mathcal{P}$i *before* Transaction
always $\mathcal{P}$o *after* Transaction

Each such dynamic constraint is transformed into a *set* of static constraint declarations, to be checked both as pre and post-conditions:

$$< \neg \mathcal{P}\mathrm{i}\ (\mathrm{State}_i) \rightarrow \mathcal{A}\mathrm{i} > \text{ and } < \neg \mathcal{P}\mathrm{o}\ (\mathrm{State}_o) \rightarrow \mathcal{A}\mathrm{o} >$$

where i and o indicate input and output states, $\mathcal{P}$i and $\mathcal{P}$o are first order logic predicates and $State_i$ and $State_o$ are input and output states of Transaction.

5 Maintaining constraints through O_2 rules

As pointed out in [SLR89], one of the problems in maintaining constraints through rules is how to execute rules in the appropriate order for events where more than one rule applies. We adopt the solution described in, among others, [SLR89, DBB+88, CW90, WF90, NQZ90]: rules are assigned execution priorities to disambiguate execution order.

As stated in Section 4, we only consider constraints where the predicate to be checked is a first order logic statements. Constraint statements, expressed as $< \neg \mathcal{P} \rightarrow \mathcal{A} >$, are transformed into O_2 rules where $\neg \mathcal{P}$ is the selector clause within an O_2 query statement and $\mathcal{A}$ is the name of a CO_2 method. The answer to the query gives enough information on the state of the database to indicate if the constraint has been violated. Since O_2 queries return complex objects and values, the answer is also used, in some cases, to indicate which objects satisfy ($\neg \mathcal{P}$) and thus might need to be processed by $\mathcal{A}$. For details on the syntax of the query language, see [BCD89].

5.1 Checking constraints at message passing events

Like all other researchers who have examined the problem of integrity violation, we only analyze constraints whose violation is caused by schema or by state *updates* (insertions, deletions or modifications). We only consider therefore message-related events, where the methods perform some update action.

It must be stressed that this represents a subset of the constraints that can be enforced by the O_2 rule system (we ignore, for instance, time-related events). We have not yet, however, been able to completely characterize other kinds of constraints. One such example is the class of constraints that are violated by the *absence* of an update, another corresponds to constraints on history-bound relationships, which require keeping auxiliary structures to monitor state sequences.

5.2 Transforming a constraint into a set of objects

This section describes how a constraint stated as a production rule is transformed into an initial set of O_2 rules. This process is at the moment done manually, since automation would require restricting the types of constraint that can be enforced (e.g., [Mor84, CW90]). Once this initial set of rules is specified and inserted into the database, the rule support system takes over, and further transformations are executed, automatically.

The first step consists in *determining the pair* $<$ **Q, A** $>$. Recall that the constraint is stated as (a set of) productions $< \neg \mathcal{P} \rightarrow \mathcal{A} >$. Predicate $\neg \mathcal{P}$ is transformed into an O_2 **Query**. The **Action** to be taken is determined by database and application semantics. It corresponds either to the name of a method supplied by the user, or to system-provided actions in the cases of cancelling or undoing of an operation.

The execution of the pair $<$ **Q, A** $>$ may be triggered by different update events. Thus, the next step in transforming a constraint into rule objects corresponds to *the determination of all* **Events** that require constraint verification. This is done in two stages. The first stage is *path analysis*, in a fashion similar to the one described in [CW90]. The query predicate is treated as a path expression, where all objects or class names mentioned in the path are potential sources of constraint violation. The determination of these sources is automated, since this corresponds to syntactically analyzing a query statement. Unlike [CW90]'s relational constraints, however, our predicates are on classes and objects that interact, and static analysis is not enough. We must then go through the next stage and *examine the database schema* to identify all methods which, sent to the potential sources of violation, may indeed cause violation. The event description pairs [*Receiver, Method*] are the output of this second stage.

The steps described above correspond to transforming a constraint into a set of rule objects **R1, ..., Rn**, all having the same $<$ **Q, A** $>$ and different **Event** components. Finally, some **Queries** can be simplified, given the restrictions imposed by the **Events**.

```
            Structural/Inheritance Information

class Person type
  tuple (Name:string, Birth:Date, Lives:Address)

class Employee inherits Person type
  tuple (Salary:Money, Dept:Department)

class Client inherits Person type
  tuple (Credit:Money, Status:string)

class Department type
  tuple(Name:string, Personnel:set (Employee), Manager:Employee)

                    Behavior Information

method change_birth (Newbirth:Date) in class Person;
method hire () in class Employee;
method change_sal (newsal:Money) in class Employee;
method change_dept (Oldep:Department, Newdep:Department)
                          in class Employee;
method change_manager (Newman:Employee) in class Department;
method buy () in class Merchandise;
```

Figure 1: Example Schema

Once these steps are taken, the rule objects can be inserted into the database. The rule support mechanism then takes over and creates additional rules to ensure constraint inheritance. If rule **R1** is created for an event [**Rec**, **M**], then other rules with the same components **Q** and **A** are automatically created for events [**Rec'**, **M**] for all subclasses **Rec'** of **Rec**. This corresponds to a phenomenon of *constraint propagation by inheritance*. The user may later assign different priorities to each such rule, as well as enable, disable or delete them individually.

6 Example of constraint maintenance

There follows an example of constraint maintenance in the system. We use the classical "Employee" - "Department" example, with a few minor additions, which will allow the reader to easily grasp the details of constraint management, without having to understand a particular application. Consider the O_2 schema in Figure 1, with classes Person (subclasses Employee, Client) and Department. Address, Money, Date and Merchandise are classes defined elsewhere.

This schema is submitted to the following Static and Dynamic constraints, and respective corrective actions:

- S_IC1: an Employee who is a Manager must earn at least $10,000 ⇒ Force Manager salary to be 10,000.
- S_IC2: (exception to S_IC1) constraint S_IC1 does not apply to Employee "Smith".
- D_IC3: a Person's age may never decrease ⇒ Forbid violation.
- S_IC4: the salary of a Manager is greater than the salary of all Employees of the Department ⇒ Non-Manager's salaries must always be kept to salary of Manager minus 1.
- D_IC5: a Client's status must be "good" before Client is allowed to buy Merchandise ⇒ Warn user and forbid operation.

The following sections show rule generation steps, pointing out relevant details. The use of O_2 queries and CO_2 methods is straightforward. We assume classes have extensions with the same name. For ease of understanding, a rule's components (E,Q,A) are numbered according the constraint they maintain - e.g., E1, Q1 and A1 refer to maintaining constraint number 1.

6.1 Processing S_IC1

- Determining **Q** and **A** components
  $\mathcal{P}$ is ($\forall$ d in Department, d.Manager.Salary $\geq$ 10,000)

  Q1: Q_result = select set(d.Manager)
  from d in Department
  where (d.Manager.Salary < 10,000)
  A1: for emp in Q_result { [emp change_sal(10,000)];}

  Notice the query returns an O_2 complex value, Q_result, whose type is defined in the select clause as being "set of Managers". This set contains all managers that violate the constraint. Also notice A1 is performed on the objects which are returned by the query.

- Determining **Events**
  Possible sources of violation are extracted from Q1's predicate "(d.Manager.Salary < 10,000)", i.e., Manager or a Manager's Salary. Manager is of type Employee and is a component of class Department, so this analysis just tells us that updating Employees or Departments may violate the constraint. This is obviously not a fine enough control to determine events. An examination of the schema (and method semantics) shows that the only events that must be checked for are
  E1.1: [Employee change_sal(Newsal)]
  E1.2: [Department change_manager(Newman)]

  This corresponds to the creation of two rules, R1.1 and R1.2, one for each event, which need only be checked after execution of the corresponding methods.

- **Q** and **A** simplification
  At event E1.1, the rule needs only be applied if method "change_sal" applies to an Employee who is also a Manager. Furthermore, Q1 is too general, since it always checks all Managers, whereas only one Manager is being affected at each event. Query and action are thus modified to:

  R1.1 (self is of type Employee), and for event E1.1:
  Q1.1: Q_result = (self $\rightarrow$ Dept.Manager == self
  AND
  self $\rightarrow$ Salary < 10,000)
  A1.1: if (Q_result) then [self change_sal(10,000)]
  R1.2 (self is of type Department), and for event E1.2:
  Q1.2: Q_result = (self $\rightarrow$ Manager.Salary < 10,000)
  A1.2: if (Q_result) then [self $\rightarrow$ Manager change_sal(10,000)]

- Detection of cycles
  The rule management system detects some types of cycles and warns the database designer. Here, one can immediately detect a cycle since R1.1 is both triggered by and executes the method "change_sal" and thus R1.1 activates itself. This is solved by using disable and enable rule operations in A1.1 as follows

  A1.1: if (Q_result) then { disable R1.1;
  [self change_sal(10,000)];
  enable R1.1;}

6.2 Processing S_IC2

This exception to S_IC1 can be handled in two ways. The first is to define pre (and post) method execution rules, where if Employee is "Smith" then both R1.1 and R1.2 are disabled (and enabled). The second (simpler) solution is to *compose* the two constraints, by modifying Q1.1 and Q1.2, adding to each the clause "AND self→Name != "Smith". The first choice maintains the independence between S_IC1 and S_IC2, but is less efficient in terms of processing time.

6.3 Processing D_IC3

This constraint predicate is stated as "*always* Birth > Newbirth", and is an example of a case where two-state constraints are expressed in terms of old (input) and new (output) values. **E**vent, **Q**uery and **A**ction after simplification are respectively:

E3: [Person change_birth(Newbirth)]
Q3: Q_result = (self → Birth ≤ Newbirth)
A3: if (Q_result) then *Break*

where *Break* is a special system method that results in not allowing the execution of "change_birth". This rule is to be checked before execution of "change_birth".

Here a new phenomenon can be observed: *constraint inheritance.* The user defines rule R3 for class *Person.* Two other rules are automatically created by the system, to account for the fact that this class has two subclasses (*Employee, Client*) to which "change_birth" can also be applied. This finally results in three rules, one stated by the system designer and the other two generated by the system, with the components < Q3, A3 > and for events [Employee change_birth] and [Client change_birth].

6.4 Processing S_IC4

Query, Action and Events are

```
Q4: Q_result = select set(emp)
               from emp in Employee
               where (emp.Salary > emp.Dept.Manager.Salary)
A4: for e in Q_result
        { [e change_sal(e →Dept.Manager.Salary - 1)];}
```

Events are those that affect Salary, Managers and Employees

E4.1 [Employee hire()]
E4.2 [Employee change_sal(Newsal)]
E4.3 [Employee change_dept(Oldep, Newdep)]
E4.4 [Department change_manager(Newman)]

Notice the query cannot be simplified to its predicate component in all four cases, because for certain events (e.g., change of Department's Manager) more than one Employee may be affected. If we had taken the approach of other authors, constraint violation would be signalled by having the constraint's predicate return the value of "false". Then, we would have to navigate through the database to perform corrective actions. Our approach is more efficient in that the query result itself already shows where to perform such actions. Query simplification to a boolean predicate can be made for events E4.1 and E4.3, only.

6.5 Processing D_IC5

This constraint can be stated as "*sometime* Client.status = "good" *before* Client buy Merchandise". This is another case of rule processing before method execution. Rule components after simplification are:

Q5: Q_result = (self → Status != "good")
A5: if (Q_result) { Warn user; Break; }
E5: [Merchandise buy]

6.6 General comments

In previous research about maintaining constraints through rules, each constraint gives origin to one rule (since updates and their effects are localized, and are not performed across classes). Here, one constraint may give origin to several rules, applied to distinct classes.

Each of the previous constraint transformations is an instance of the different issues covered. Processing of S_IC1 shows simplification of query and action, as well as the fact that part of an action may include the temporary disabling of the constraint itself. Enable and disable operations allow controlling constraint scope. Thus, constraints may be enabled only within an application or a transaction. One may want, for instance, to disable constraint S_IC1 for a transaction that will change all salaries of a company, and perform a global verification of Manager's salaries at the end.

Processing of S_IC2 shows exception handling options and that constraint composition is transformed into query and action composition. Processing of D_IC3 shows processing of an "always" dynamic constraint, the special system action *Break*, and the feature of automatic constraint inheritance. S_IC4 is an example of the need for a query that returns a complex value and not just a boolean. Again in this our system differs from the proposals of most authors that are limited to checking boolean expressions. Finally, D_IC5 shows handling of a "sometime/before" dynamic constraint.

7 Conclusions and future work

This paper presented a solution for maintaining integrity constraints in an object-oriented system, which was implemented using the O_2 production rule mechanism. The approach described here is original in that constraints are transformed into objects and therefore managed as database components by the database management system itself. Constraints are considered as part of a schema and are treated as first-class citizens. This permits supporting object-oriented characteristics such as constraint inheritance and independence, and which are ignored by most researchers.

Unlike previous work on constraint maintenance in object-oriented databases, this solution considers not only static but also some two-state dynamic constraints, as well as constraints on behavior. Also unlike previous work, we consider both global and local constraints, as well as inter-class constraints. Other aspects that distinguish our approach from others' are the ability to treat exception handling, considering one constraint as enforced by sets of rules (and not just one) and support to system evolution, where modification of the set of database constraints is transparent to the applications. Finally, constraint enforcement can be disabled and enabled for different transactions.

Future work will consider extending the set of allowable constraints, as well as partially automating the determination of the set of events which correspond to a constraint statement.

Acknowledgements

The authors thank Guy Bernard and Christophe Lécluse for their careful reading of previous versions of this paper, and their insightful comments.

References

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A query-language for an object-oriented database system. In *Proceedings of the Second Workshop on DataBase Programming Languages*, Salishan, Oregon, USA, June 1989. Morgan Kaufman.

[BMP91] C. Bauzer-Medeiros and P. Pfeffer. A Mechanism for Managing Rules in an Object-Oriented Database. Technical Report 65-90, GIP Altaïr, Rocquencourt, France, 7 janvier 1991.

[CCF82] J. Castilho, M. Casanova, and A. Furtado. A Temporal Framework for Database Specifications. In *Proceedings of VLDB*, pages 280–291, 1982.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proceedings of the 16th VLDB*, pages 566–577, 1990.

[Da90] O. Deux and al. The Story of O_2. *Special Issue of IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarty, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jaurhy. The HiPAC Project: Combining Active Databases And Timing Constraints. *SIGMOD RECORD*, 17(1), March 1988.

[fADF90] The Committee for Advanced DBMS Function. Third Generation Data Base System Manifesto. In *Proceedings of SIGMOD'90*, Atlantic City, May 1990.

[HK89] S. Hudson and R. King. Cactis: a Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM TODS*, 14(3):291–321, 1989.

[HS90] K. Hulsmann and G. Saake. Representation of the Historical Information Necessary for Temporal Integrity Monitoring. In *Proceedings of the 2nd EDBT*, pages 378–392, 1990.

[JMSS89] M. Jarke, S. Mazumdar, E. Simon, and D. Stemple. Assuring Database Integrity. Submitted for publication, 1989.

[KDM88] A. Kotz, K. Dittrich, and J. Mulle. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In *Proceedings of the 1st EDBT*, pages 76–91, 1988.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109 –124, March 1990.

[Kun85] C. Kung. On verification of database temporal constraints. In *Proceedings of the ACM SIGMOD*, pages 169–179, 1985.

[Lip88] U. Lipeck. Transformation of Dynamic Integrity Constraints into Transaction Specifications. In *Proceedings of ICDT*, pages 323–337, 1988.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Database. In *Proceedings of PODS*, 1989.

[LR90] Christophe Lécluse and Philippe Richard. Data Base Schemas and Types Systems for DBPL. Rapport Technique 55-90, GIP Altaïr, Rocquencourt, France, 29 août 1990.

[MBW80] J. Mylopoulos, P. Bernstein, and H. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM TODS*, 5(3):185–207, 1980.

[Mor84] M. Morgenstern. Constraint Equations: Declarative Expression of Constraints with Automatic Enforcement. In *Proceedings of the 10th VLDB*, pages 291–300, 1984.

[NQZ90] R. Nassif, Y. Qiu, and J. Zhu. Extending the Object-Oriented Paradigm to Support Relationships and Constraints. In *Proceedings of the IFIP Conference Object Oriented Database Systems - Analysis, Design and Construction*, 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Database Systems. In *Procedings of the ACM SIGMOD*, pages 281–290, 1990.

[SK84] A. Shepherd and L. Kerschberg. PRISM: a Knowledge Based System for Semantic Integrity Specification and Enforcement in Database Systems. In *Proceedings of ACM SIGMOD*, pages 307–315, 1984.

[SLR89] T. Sellis, C. Lin, and L. Raschid. Implementing Large Productions Systems in a DBMS Environment: Concepts and Algorithms. In *Proceedings of ACM SIGMOD*, pages 404–412, 1989.

[UD89] S. Urban and L. Delcambre. Constraint Analysis for Specifying Perspectives of Class Objects. In *Proceedings of the 5th IEEE Conference on Data Engineering*, pages 10–17, 1989.

[Via88] V. Vianu. Database Survivability Under Dynamic Constraints. *Acta Informatica*, 25:55–84, 1988.

[WF90] J. Widom and S. Finkelstein. Set Oriented Production Rules in Relational Database Systems. In *Proceedings of the ACM SIGMOD*, pages 259–270, 1990.