Implementing Integrity Control in Active Databases

Claudia Bauzer Medeiros
Márcia Jacobina Andrade

Relatório Técnico DCC–06/92

Julho de 1992

Implementing Integrity Control in Active Databases

Claudia Bauzer Medeiros   Márcia Jacobina Andrade

Abstract

This paper presents an integrity maintenance system that has been developed for maintaining static constraints in databases, using the active database paradigm. This system has been added to the O_2 object oriented database system, and is fully functional. Constraints are specified by the user in a first order logic language, and transformed in production rules, which are stored in the database. The rules are then used to maintain the corresponding set of constraints, for all applications that use the database, and which no longer need to worry about integrity control. We extend previous work on constraint maintenance in two ways: our system can be used as a constraint maintenance layer on top of object-oriented, relational and nested relational databases; in the case of object-oriented systems, we provide constraint support not only in the case of object composition, but also consider inheritance and methods.

Keywords: Active databases, object-oriented databases, integrity constraint maintenance

1 Introduction

An *integrity constraint*, in a database environment, is a statement of a condition that must be met in order to maintain data consistency [JMSS90]. Constraints can be classified in several ways. One of these many classifications relies on distinguishing *static* constraints – i.e., those that define a valid database state and are usually stated using first order logic – and *dynamic* constraints – that specify valid database state transitions, and are specified using modal logic. The same kinds of constraints exist in the object-oriented framework, where one can still use first order logic to define static constraints, using classes, objects and object components as variable ranges (instead of relations, tuples and attributes). The new element is the appearance of constraints over methods, for which no study has yet been made.

The problem of automatic maintenance of integrity constraints in a database management system (DBMS) has been extensively considered in the literature. Since database systems have limited support for such a facility, application developers are forced to embed code in each application, in order to verify constraints. This has also the inconvenient of making all applications very sensitive to any modification in the constraint set, besides leaving to programmers the burden of having to know and check all relevant constraints at each step.

A solution to this problem presented by several researchers is to embed a trigger or a production rule system into the DBMS, to be activated upon update requests. If this solution is applied, constraint specification and maintenance can be kept independent from application development.

This paper analyzes the problem of maintaining static integrity constraints in databases, using production rules and the active database paradigm. The main results presented are the following:

- a general algorithm for automatically transforming constraints stated in first order logic into a set of production rules, using the active database paradigm. Unlike previous work, this algorithm provides support for constraint inheritance and composition, and for constraints over methods;

- the description of a constraint management system, which uses the algorithm, and has been added on top of a version of the O_2 object-oriented database system [Da90] that supports rule management ([MP91a]). The constraint management system is fully functional, and can be also used for constraint maintenance in relational or nested relational systems, which are shown to be special cases of the object-oriented data model adopted.

Our work uses the object-oriented data model of [Bee89] which considers a *class* to be a collection of objects. Objects are characterized by their structure (class type) and behavior (class methods). Classes are subject to inheritance and composition properties. An object-oriented *database schema* is a description of the properties of the database classes (described by composition and inheritance graphs) and object behavior (methods).

This paper is organized as follows. Section 2 discusses the role of active database systems on constraint maintenance. Section 3 presents the algorithm for transforming static constraints into production rules, for relational and object-oriented systems. Section 4 presents the algorithm's implementation and its use as a constraint transformation layer on top of the O_2 DBMS. Section 5 gives some examples of automated rule generation for object-oriented systems. Finally, section 6 presents conclusions and directions for future work.

2 Active Database Systems and Constraint Support

Active databases are systems that respond to events generated internally or externally to the system itself without user intervention. The term *active* appeared in [Mor83] to describe a database that supports automatic view modifications to reflect updates to the underlying database. Nowadays, the active dimension is usually supported by production rule mechanisms, provided by the DBMS. The term *rule* is associated with the notion of *production rules* which are usually defined as clauses **If X then Y**, where X is a predicate to be tested, and Y is an action to be performed if the predicate is true. In [DBM88] the execution of a production rule was described as being triggered by an *event* (e.g., transaction commit). This notion is now adopted by most researchers in active databases.

Rules have thus become triples

$$<E,C,A>$$

where E stands for the Event upon which the rule is triggered, C the Condition (predicate) to be tested, and A the Action to be performed if the condition is met. From now on, the term *rule* in this paper will denote this triple.

An active system should support not only the declarative specification of rules but also their manipulation (query, update, activation and deactivation of rules). The paradigm of active databases is useful for implementing or extending several database functions. Some examples are integrity control, handling of derived data, processing of view updates, and monitoring of events (e.g., [Mor83, Mor84, Ris89, RCBB89, MD89]).

Most of the published research on active databases discusses the support of static integrity constraints using <E,C,A> rules. In such a framework, rule components have the following meaning: the Event is an update request; the Condition is the constraint's predicate; and the Action, optional, is provided by the database designer, and can be corrective or preventative. Preventative actions block an update if it violates the constraint. Corrective actions correspond to sequence of operations to be executed that restore the database's consistency after the update is performed.

A typical example of the advantage of such systems over traditional systems is that of engineering database applications. As remarked in [Mor89], these applications require a large number of constraints, whose verification is usually coded within the application itself. Any modification in the set of constraints does therefore require intensive recoding. The active database approach, on the other hand, has the following advantages:

- it supports specification and maintenance of constraints by the DBMS itself, regardless of the application code, allowing independent evolution of both;

- it provides use of the same underlying database for all applications that have the same view of the world (i.e., supports constraint and data sharing);

- it allows the possibility of different applications having different views of the same database, by enabling and disabling different sets of rules (e.g., [CBB$^+$89]).

Examples of research on constraints and active relational databases are: [Mor84] (where some of the characteristics of active databases are discussed, and a semantic constraint language is proposed); [CW91, WF90] (where SQL is extended to incorporate constraint specification, and constraints generate production rules that check updates over tuples or sets of tuples); [SPAM91] (where it is shown how to extend a relational system with a layer that manages special – active – relations, which control events and allow trigger execution); [SZ91] (where the problem of optimizing the execution of large sets of rules by materializing specific relations is considered).

[SHP88, SJGP90] present a mechanism for managing rules in the extensible system POSTGRES, and how these rules can be used to support views, procedures, and integrity constraints.

Research on constraints using active object-oriented databases includes: [KDM88] (where an extended trigger mechanism is described for the DAMASCUS system); [DBM88, CBB$^+$89] (introducing <E,C,A> rules in the HIPAC system); [DHL90, DHL91] (where rules are studied in the context of long-running activities, where the action may be postponed); [NQZ90] (using rules to help semantic modelling on GemStone); [UD89, UD90, UKN92] (where rules are used to maintain constraints, and an algorithm is proposed to detect rule cycles); [MP91a, MP91b] (where a rule mechanism implemented on top of the O2 database is discussed, as well as its consequences on constraint maintenance); [DPG91] (where it is shown how to specify rules to maintain constraints in the ADAM system).

In all the above cases, the solutions proposed are particular to a given data model or subsystem. We now discuss how to generalize the solution.

3 Transforming Constraints into Rules

3.1 Introduction

Given that active databases are an adequate and flexible solution for maintaining constraints, there remains the problem of *automatically determining a set of rules* that, for a given constraint, will ensure its enforcement by an active database. Some solutions for this problem have already been proposed, but they are usually not integrated into any database system. Exceptions are [WF90], for an extended relational system, and [UKN92], for an object-oriented system. In both cases, the solutions proposed are specific to a system, and cannot be applied to active databases in general.

[WF90] show how to transform a constraint specified in first order logic into rules, which are triggered by updates in the extended relational system Starburst. [UKN92] describe automatic transformation of constraints into rules in an object-oriented database, but constraints are limited to schema (composition) integrity, and only some restricted types of data integrity. Inheritance, methods, or more general constraints over specific database states are not considered.

This section extends and generalizes previous proposals for automatically transforming general static constraints into $<$ E,C,A $>$ rules. Our transformation algorithms use ideas from both [Mor84] and [WF90]'s static analysis of constraints, and incorporate [MP91b]'s rule creation proposal. The constraints handled can be defined over schema (extending [UKN92]) or data (as in most of the references named in Section 2). Furthermore, in case of object-oriented databases, constraints can also apply to method execution (e.g., order). This extends previous work by considering not only the usual constraints over schema or state, but also over behavior.

3.2 Extracting C,A components from constraints

In non-active systems, programmers predict which updates violate which constraints, and add the proper verification procedures to the code. In active databases, the problem lies in establishing beforehand and independent from any application which updates (i.e., events) will be sources of constraint violation.

Let a constraint be specified as a first order logic predicate $\mathcal{P}$ over a database state, and $\mathcal{A}$ some user-defined action to be performed if $\mathcal{P}$ is not true (i.e., **if** $\neg\mathcal{P}$ **then** $\mathcal{A}$). The same predicate $\mathcal{P}$ has to be checked at several different events (updates) which may violate the corresponding constraint. An active database must thus provide a set of rules $<$\{E\},$\neg$ $\mathcal{P}$,$\mathcal{A} >$ to maintain this constraint. Whereas the $\neg$ $\mathcal{P}$ and $\mathcal{A}$ components can be derived straight from the constraint specification, event determination depends on additional information. Thus, the most complex part of automatic rule derivation is that of *determining events* (i.e., updates) which may violate a constraint. The objective of event derivation is thus to avoid unnecessarily checking constraints at every update, and thus save processing time.

In relational systems, updates (events) that may violate a given constraint can be usually determined by static analysis of the constraint, and have a limited range of action (i.e., constraints are generally restricted to at most two relations). In an object-oriented system, however, updates are performed by methods to obey encapsulation. This complicates enormously the definition of all events which may be associated with integrity violation, mainly for the following motives:

- overloading and polymorphism associated with message sending, which does not allow static determination of a message receiver, nor supports unique method implementation;

- multiplicity of functions and methods associated with an application;

- inheritance mechanisms, whereby constraints which refer to a given class may be valid for its subclasses;

- composition mechanisms, whereby constraints stated for one class may have to be checked when other (apparently unrelated) classes are updated.

In relational systems, events consist of applying the *insert, delete, update* operations over a tuple or relation. In object oriented systems, however, a given object may react to several methods, whose names are defined by the user, and whose implementation depends on the method's receiver and cannot be fathomed unless additional context-sensitive information is provided. Thus, algorithms for transforming constraints into rules such as proposed by [Mor84, WF90] do not apply to object-oriented systems. Furthermore, analysis of the constraint statement alone (e.g., [Mor84]) is not sufficient, since additional (method) semantic information must be provided.

3.3 Event detection algorithm

We now describe the algorithm that determines all events that can violate a given constraint. The set of events determined is a superset of the actual set of events, since specifying the exact set of events is computationally unfeasible. Constraints over method execution are assumed to be pre or post-conditions to methods.

Events produced by the algorithm are expressed as $< \mathcal{C}$lass,$\mathcal{M}$ethod$>$ pairs (or $< \mathcal{O}$bject,$\mathcal{M}$ethod$>$ for constraints that refer to a specific object)[1]). This notation means that $<$ method $\mathcal{M}$ is being executed on some object of class $\mathcal{C}$, or on object $\mathcal{O} >$. Analogously, for relational and nested relational systems, events are stated as $<$ relation_name, update_operation $>$, where the update operation may be *insert, delete* or *update*.

Consider, for instance, the schema on Figure 1, where Money, Address and Date are classes declared elsewhere. Client and Employee are subclasses of Person, and Client-Employee is a subclass of both. Let CN1 be the constraint below, which states that a manager's salary is greater than 10,000,000.00

$$\text{CN1} = \forall \text{ x} \in \text{Department, x.manager.salary} > 1{,}000{,}000.00$$

CN1 has to be checked when a manager's *salary* is updated. Since field *manager* is of class Employee, this means checking CN1 when the method

[1] Certain systems, like O2, allow specifying names for chosen objects

update-salary is activated on any object of class Employee. The algorithm would in this case indicate that <Employee,update-salary> is one of the events that may violate CN1. In other words, if *update-salary* is executed on an Employee, the rule has to check at run time if this Employee is a manager, since in this schema one cannot determine statically if an employee is a manager).

In order to understand the algorithm, the following terms are defined:

- *Method attributes* are all fields that can be updated by a method, and which are not necessarily restricted to the parameters specified in the signature (e.g., method *hire* updates several Employee fields, but has no parameters). These fields are provided by the designer and may vary with the receiver's class.

- *Class components* are the fields that are the components of a class type (e.g., *name, birth, address* in class Person);

- *Prefix fields* are the (compound) prefixes of a field in a constraint. For instance, the expression "x.manager.address.city" denotes: the *city* field of the *address* field of the *manager* field of (an object) x. Its prefix fields are: x.manager and x.manager.address.

3.3.1 Central ideas

The algorithm uses three sources of input: the integrity constraint itself; data from the schema; and method execution information. *Schema information* consists of data about class composition (composition graphs), inheritance (inheritance graphs) and methods defined for a class. *Method execution information* consists of all method attributes for each method in a database (i.e., all fields updated by a given method, as defined previously), and must therefore be provided by the user. The output of the event detection algorithm is the set of events that may violate the input constraint.

The main steps for event detection are the following:

1. Identify all classes and objects directly referenced in the constraint (e.g., in CN1, class Department is identified), all class components

Structural Information

Class *Person* **type**
tuple *(name: String, birth: Date, address: Address)*

Class *Employee* **inherits** *Person* **type**
tuple *(salary: Money, dep: Department, child:set(Person))*

Class *Client* **inherits** *Person* **type**
tuple *(credit: Money, status: String)*

Class *Department* **type**
tuple *(name: String, personnel: set(Employee), manager: Employee)*

Class *Client-Employee* **inherits** *(Client, Employee)*

Class *Vehicle* **type**
tuple *(chassis: Integer, price: Money)*

Class *Car* **inherits** *Vehicle* **type**
tuple *(model: String, owner: Employee)*

Class *Boat* **inherits** *Vehicle* **type**
tuple *(tonnage: Integer)*

Behavioral Information

method *update-birth(newbirth: Date)* **in class** Person
method *hire ()* **in class** *Employee*
method *add_child (child:Person)* **in class** *Employee*
method *update-salary(newsalary:Money)* **in class** *Employee*
method *update-dep (olddep, newdep: Department)* **in class** *Employee*
method *update-manager (newmanager: Employee)* **in class** *Department*
method *update-price (newprice: Money)* **in class** *Vehicle*
method *update-chassis (newchassis: Integer)* **in class** *Car*
method *assign-owner (chassis: Integer, newowner: Employee)* **in class** *Car*
method *insure (tonnage: Integer, newprice: Money)* **in class** *Boat*

Figure 1: Database Schema

referenced, and their respective class types, by analysis of prefix fields (e.g., in CN1, *manager* is a component of class Department and belongs to class Employee, and *salary* is a component of class Employee). This identification is done by querying the schema composition graph.

2. (Constraint propagation across the composition graphs) Once the classes and class components referenced are identified, determine all methods that can violate the constraint. This means verifying which methods update some of the components identified in (1). For CN1, methods *hire* and *update-salary* update component *salary* which was identified in (1), and may therefore violate CN1. All such methods and the classes of the components to which they apply constitute events, which are stored in an event list.

3. (Constraint propagation along the inheritance graphs) The database inheritance graphs are traversed, to determine other events that may violate the constraint by inheritance of components or of methods. This is equivalent to performing inheritance of constraints. These new events are added to the event list, which constitutes the output. Inheritance of constraints is of two types: the component and the method are inherited; or the component is inherited, but the method is local. For instance, suppose class Employee had a subclass Bad_Emp with component *salary* affected by a local method *decrease-salary.* Other events would be < Bad_Emp,update-salary >, < Bad_Emp, hire > (both by inheritance of component and of method), and <Bad_Emp, decrease-salary > (inheritance of component only).

This algorithm also applies to nested and relational models, where only the two first steps are necessary. In these cases, methods are replaced by relational database update operations, and there is no inheritance graph. For relational systems, the composition graph is simplified to relation names and attribute atomic domains. For nested relational systems, the composition graph allows relations as attribute domains, and is a special case of object-oriented composition graphs when the only constructors allowed are *relation* and *tuple*. Thus, instead of component information we simplify all structures to contain only relation schemas. Given these structures, the algorithm subsumes those presented in [WF90] for relational databases and in [UKN92] for object-oriented systems.

3.3.2 Algorithm

Input: Static constraint specified declaratively using a first order language, with components stated in prefix form
Output: Set of events

1. Step 1: Parse the constraint, and identify the classes and components involved. Initialize an empty list of events, L_event.

2. Step 2: Determine the class of each component and component prefix identified in **Step 1**.

   Store these classes in table Tclass (contains classes where updates may violate the constraint), and the attributes in table Tclass_attribute (which indicates class components that may be sources of constraint violation). Initialize an auxiliary empty class table, T_old.

3. Step 3: While (Tclass - T_old) not empty do

   - 3.1 For each class **c** in (Tclass - T_old) do
     - 3.1.1 Identify the methods **m** that can affect **c** (defined locally or inherited)
     - 3.1.2 For each method **m** identified in 3.1.1, check if at least one of its attributes appears in table Tclass_attribute (i.e., if the method updates a component mentioned in the constraint). If positive, then add event <**c**,**m**> to L_event.
     - 3.1.3 Add **c** to T_old (i.e., the class has been processed)

- 3.2 For each class **c** in T_old, identify its immediate descendants in the inheritance graph, and add them to Tclass

This algorithm is simplified in the case of constraints over methods. These constraints are declared as pre or post-conditions to method execution (i.e., to a set of events). In such a case, the events are determined directly at parse time, and subsequent steps are not executed. One example is

CN2 = $\forall$ e $\in$ Employee: precond(e,update_salary) :: e.salary > 0.0

(i.e., a precondition to event < Employee, update-salary > is that the employee's salary must be positive)

4 Implementation of the Constraint Transformation Layer

The constraint subsystem was implemented in C on top of the active version of the O_2 object-oriented DBMS, and runs in an Unix SPARC workstation environment.

The rule generation algorithm needs different types of information from the database schema, as well as semantic information about methods. If many constraints have to be processed at once, this requires repeatedly executing the same set of queries over the database schema, and constant interaction with the database designer. We solved this problem by dividing the implemented system into two modules: a *schema extraction* module and a *rule generation module.* The first module extracts all schema and method information needed for rule generation, and loads it into temporary structures in main memory. The second module is the rule generator, and uses these structures directly, instead of querying the database.

This decomposition presents the further advantage of allowing the rule generation module to be database independent, and even model independent. The rule generation module can be attached as a layer on top of any database system, for different (system-tailored) schema extraction modules. We developed the modules separately and added them to the

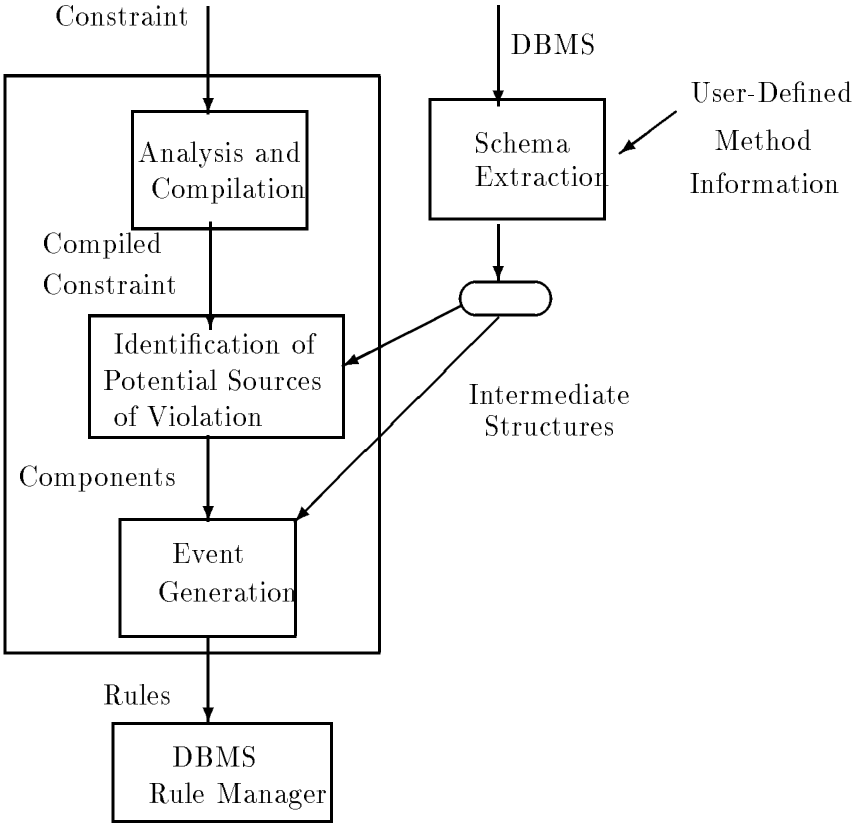

Figure 2: Constraint Transformation System

version of the O_2 object-oriented DBMS described in [MP91a]. Given these interface modules, our implementation will work for any object-oriented system that uses a class-based model as described in [ABD$^+$89], and for relational and nested relational databases.

In order to completely implement a constraint handling layer for a database system, we also developed a declarative constraint specification language. This language is based on first order logic, and allows defining constraints over schema components (or relations), objects or classes. It also allows defining relationship constraints involving instances of different classes, and constraints that check pre or post-conditions for method execution. More details can be found in [And92].

Figure 2 shows the general structure of the constraint system, where the Schema Extraction module is system-dependent, and the remainder is system and model independent. The Schema Extraction module collects information from the schema and from the user and stores it in the intermediate data structures, especially designed for fast access. These structures are passed on to the Rule Generation subsystem: module Analysis and Compilation parses the constraint using a recursive descendent analysis and builds the C and A fields of the $<$ E,C,A $>$ rule; it passes the result on to the module that performs Identification of Potential Sources of constraint Violation (which are classes, components and prefixes); finally, Event Generation determines the set of events. The rules are then handed to the database rule manager.

5 Examples

Let us consider some examples of rule derivation based on the database schema of Figure 1. Since we obtain the same results as other authors in the case of relational systems, we only show cases of constraints in object-oriented systems. All examples, except for the first one, correspond to situations not considered in the literature. User method information has been ommitted for brevity's sake and will be mentioned when needed.

The constraint in figure 3 is an example of a unique key specification (no two Cars can have the same *chassis* number). The only class named in the constraint is Car, and the only component is *chassis*, of type Integer. Notice that this component is inherited from Car's superclass, Vehicle. This information is stored in tables Tclass and Tclass_attribute. Method information given by the user shows that two methods (*update-chassis, assign-owner*) may affect the *chassis* component. Both methods are defined for class Car, though they modify an inherited component. These two methods are used in setting up the list of events L_event. Class Car has no subclasses in the inheritance graph, so the algorithm terminates. Two rules are generated (one for each event) of the form < E, C, *FORBID*> , where

the events are <Car, update-chassis> and <Car, assign-owner>, taken from L_event;

C is the violation of the constraint's predicate, i.e., C $\equiv \exists c1, c2 \in$ Car : c1.chassis=c2.chassis;

keyword *FORBID* in the action field is the default when no action is provided.[2]

This is an example of a constraint that is checked locally over the class for which it is declared, and is similar to examples of rule generation for key maintenance in [UKN92] or [WF90]. Condition and action are derived in a similar way in the examples that follow, and will thus not be shown again.

Figure 4 shows a simple example of constraint inheritance. The constraint is specified over class Employee, and affects component *salary*, and therefore potentially class Money. Both classes are stored in Tclass, and *salary* is stored in Tclass_attribute. The methods that affect this component are *hire* and *update-salary*, and the corresponding events are placed in L_event. The information about *hire* cannot be derived from the method's signature, and has to be provided by the user. Finally, class Client-Employee is a subclass of Employee, and thus the constraint

[2] In the active version of O2, the rule manager will transform this input into two rule objects, where the condition becomes an O2 query, and FORBID becomes a rule manager predefined action. In this system, it stores the rule as having to be triggered as post-condition to method execution. When the event is signalled, the query is performed, and the update is not committed if the query result is not empty

Tclass

Car

Tclass_attribute

chassis: Integer

Levent

Class	Method
Car	update-chassis
Car	assign-owner

$\forall$ c1,c2 $\in$ Car c1.chassis <> c2.chassis

Figure 3: Unique key constraint - no propagation

is propagated to it, the final set of events being composed by the methods defined for class Employee, and inherited by class Client-Employee. Notice that it is not every method in Employee or Client-Employee that triggers constraint verification, but only the methods that affect the field *salary* (which is in Tclass-attribute). Inheritance of constraints has not been considered by other authors.

Figure 5 extends the previous example showing propagation of constraints using information about composition (taken from prefix fields of the constraint). The constraint specifies a single class - Car - but refers to other classes (since a Car's *owner* is an Employee, and *price, salary* are of class Money). Therefore, events that may violate the constraint are not restricted to the class named in the constraint, but propagate to the component's classes. As in the previous example, the constraint is inherited by Client-Employee.

The list of events is a superset of the actual event list. For instance, method *hire* is included, even though in this case it does not violate the constraint (though it affects Employee's *salaries*). In order to eliminate superfluous events, additional information would have to be known (e.g., that execution of *hire* precedes any owner assignation, and thus does not

Tclass

Employee Money Client-Employee

Tclass-attribute

salary: Money

Levent

Class	Method
Employee	update-salary
Employee	hire
Client-Employee	update-salary
Client-Employee	hire

$\forall$ e $\in$ Employee : e.salary $>$ 10,000

Figure 4: Constraint propagation by inheritance of methods

affect Car objects). However, the information needed will vary from constraint to constraint and cannot be determined beforehand for all cases. (Our implementation lets the user examine the rules generated, and exclude or disable the ones which are not of interest).

Figure 6 is another example of constraint inheritance, but of another nature. This is a domain constraint, for the *price* of a Vehicle. The classes affected are Vehicle (named in the constraint), and Boat and Car (subclasses). The only method that affects a Vehicle's *price* is *update-price*, which is inherited by Boat and Car, thus updating L_event (similar to the previous examples). However, Boat has a local method (*insure*) that also affects the *price* and <Boat, insure> is thus added to L_event. This example therefore shows a case where a constraint may be violated by a (local) method modifying an inherited attribute, and where again constraint analysis is not sufficient to derive all rules.

Tclass

Car
Money
Employee
Client-Employee

Levent

Class	Method
Car	update-price
Car	assign-owner
Employee	update-salary
Employee	hire
Client-Employee	update-salary
Client-Employee	hire

Tclass-attribute

price: Money
owner: Employee
salary: Money

$\forall$ c $\in$ Car, c.owner.salary > c.price

Figure 5: Constraint propagation by inheritance of methods and composition

Tclass

Vehicle
Car
Boat

Levent

Class	Method
Vehicle	update-price
Car	update-price
Boat	update-price
Boat	insure

Tclass_attribute

price: Money

$\forall$ v $\in$ Vehicle: v.price > 0

Figure 6: Influence of local methods on constraint inheritance

6 Conclusions

This paper analyzed the problem of maintaining static integrity constraints in database systems and presented an algorithm for automatically transforming an integrity constraint into a set of <E,C,A> rules. This algorithm generalizes previous work in the area of constraint maintenance in active databases. Among the main contributions presented we can cite:

- the algorithm was implemented into a constraint management system that is fully functional, and takes advantage of the rule facilities implemented for an object-oriented DBMS (O_2);
- the system supports constraints over schema, over data and over method execution order, and automatically performs constraint propagation along inheritance and composition graphs;
- the system can be used as a layer over different active database systems that have rule support systems. It was designed to be model independent, and to be employed for constraint maintenance in active relational, nested relational or object-oriented DBMS.

Several extensions are under consideration. The first concerns taking schema evolution into account. If the schema is modified, previously defined constraints may cease to hold, and the corresponding rules have to be deleted, or new rules may have to be added (e.g., when a class is added in a hierarchy whose root is subject to an already existing constraint). In the present version, every schema update requires reprocessing all constraints in order to determine a new set of rules, which is very costly. One possible improvement we are studying is that of maintaining additional classes (or relations) in the database, to record association of constraints with the corresponding rules, and relationships of rules and the classes (relations) whose integrity they check. Thus, when the schema is modified, only a subset of constraints and rules need be checked. This however brings the inconvenient of burdening the database with additional information (relating rules and constraints) which may never be used.

The system described in this paper assumes that all constraints that refer to a given class are inherited by its subclasses. This may not be always the case, since in certain cases the user may want to specify exceptions to constraints, as in [Bor85]. We are also analyzing this problem, and its implications in the rule determination process.

Another extension concerns restricting the set of events, given additional semantic information about a database. For instance, when a constraint refers to an aggregate field (e.g., average) then not necessarily all methods that affect the field's components may violate the constraint. A constraint that imposes that a given average may never decrease will not be affected by a method that will increase the fields that compose this average. We are analyzing how to increase the input semantic information to the constraint system, in order to reduce the set of rules produced in those and other special cases.

Finally, no consideration was made in the direction of detecting cycles in rule execution. In our case, this would imply analysis of the rules' actions, to determine rule activation graphs, and from these graphs deduct the existence of rule loops (since actions may themselves require execution of methods, which will in turn trigger other rules). One possible solution is to adapt the study that appears in [AWH92] for rules in an extensible relational DBMS, extending it to include information about inheritance and composition.

Acknowledgements The authors thank Frederico Sidney Cox Jr. for his insightful comments and careful reading of this paper.

References

[ABD$^+$89] A. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-oriented Database System Manifesto. In *Proc. First Conference on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[And92] M. J. Andrade. Manutenção de Restrições de Integridade em Bancos de Dados Orientados a Objetos. Master's thesis, Dept. Computer Science, UNICAMP, March 1992.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence and Observable Determinism. Technical Report RJ8562, IBM Almaden, 1992.

[Bee89] C. Beeri. Formal Models for Object-oriented Databases. In *Proc. 1st International Conference on Deductive and Object-oriented Databases*, pages 370–395, 1989.

[Bor85] A. Borgida. Language Features For Flexible Handling of Exceptions in Information Systems. *ACM TODS*, 10(4):565–603, 1985.

[CBB$^+$89] S. Chakravarthy, B. Blaustein, A. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. MacCarthy, R. McKee, and A. Rosenthal. HiPAC: a Research Project in Active, Time-constrained Database Management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, 1989. Final Technical Report.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. 17th VLDB*, pages 577–589, 1991.

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

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object oriented database system. In *Lecture Notes in Computer Science*, volume 334, pages 129–143. Springer Verlag, 1988. 2nd Workshop in OODBS.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. ACM SIGMOD*, pages 36–58, 1990.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proceedings 17th VLDB*, pages 113–122, 1991.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Uniform Approach. In *Proceedings 17th VLDB*, pages 317–326, 1991.

[JMSS90] M. Jarke, S. Mazumdar, E. Simon, and D. Stemple. Assuring Database Integrity. *J. Database Admin.*, 1(1):391–400, 1990.

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

[MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD*, pages 215–224, 1989.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. 9th VLDB*, pages 34–42, 1983.

[Mor84] M. Morgenstern. Constraint equations: Declarative expression of constraints with automatic enforcement. In *Proc. 10th VLDB*, pages 291–300, 1984.

[Mor89] M. Morgenstern. Constraint-Based Systems: Knowledge About Data. In *Proceedings of 2nd International Conference on Expert Database Systems*, pages 23–43, 1989.

[MP91a] C. Medeiros and P. Pfeffer. A Mechanism for Managing Rules in an Object-oriented Database. Technical report, GIP-ALTAIR, 1991.

[MP91b] C. Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proceedings European Conference on Object-Oriented Programming*, pages 219–230, 1991.

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

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Proc. 15th VLDB*, 1989.

[Ris89] T. Risch. Monitoring database objects. In *Proc. 15th VLDB*, pages 445–453, 1989.

[SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7), 1988.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, pages 281–290, 1990.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings 17th VLDB*, pages 469–478, 1991.

[SZ91] A. Segev and J. Zhao. Data Management for Large Rule Systems. In *Proceedings 17th VLDB*, pages 297–307, 1991.

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

[UD90] S. Urban and M. Desiderio. Translating Constraints to Rules in CONTEXT: a CONstrainT EXplanation Tool. In *Proc.*

IFIP Conference Object Oriented Database Systems - analysis, Design and Construction, 1990.

[UKN92] S. Urban, A. Karadimce, and R. Nannapaneni. The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database. In *Proc. IEEE Data Engineering Conference*, pages 565–572, 1992.

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

Relatórios Técnicos

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`