Mathematical Problems of Computer Science 23, 2004, 80–99.

Consistency Management In Database Systems: Review

Armen J. Asatryan

Institute for Informatics and Automation Problems of NAS of RA
UNICAD CJSC
e-mail armen.asatryan@unicad.am

Abstract

By its definition, a database must serve as a faithful and incorruptible repository of data. Applications that consult the database expect a "warranty" that the database is supplying the correct values. This survey briefly presents the approaches of integrity constraint management in database systems. It reflects the various research activities in this field. We focus on central approaches, concepts, methods, and systems in this area.

1 Introduction

In the field of databases, the term *integrity* normally refers to the correctness or validity of the stored data, as defined explicitly by means of integrity rules or integrity constraints. Integrity is a very important property of database systems. The lack of integrity has usually negative consequences. It is possible to build some mechanisms to guarantee some level of integrity in a database. This is achieved by means of conditions defined over the facts of the database called integrity/consistency constraints (or, for short, constraints). If these conditions are satisfied, we may have some level of confidence on the database integrity. Therefore the integrity of a database is partially guaranteed by means of constraints, i.e. conditions that define properties to be satisfied by the database. The purpose of constraints is to restrict the database states to those that are considered as legal. Therefore, constraints have to be expressed in a formal way, i.e., must be made known to the database system. In general, consistency constraints can be classified according to where they are specified into *internal* and *external* consistency constraints. Internal constraints are those that are known to and can be enforced by the database system, while external constraints have to be expressed, checked, and enforced within application programs. Internal constraints can be subdivided into the following three types:

- *inherent* consistency constraints: they are fixed for a given data model and therefore do not have to be specified (e.g., absence of cycles in inheritance relationships),
- *implicit* consistency constraints: they express restrictions due to the semantics of the data model, i.e. they are implicit in the database schema, e.g., the identification of unique properties,

- *explicit* consistency constraints: they are arbitrary properties to be satisfied by the database that cannot be captured by schema restrictions. They are formulated in a separate sublanguage of the data definition language (DDL), the constraint definition language, either as predicates in a declarative language.

Another classification of constraints is based on whether only single database states or even database state transitions can be constrained:

- *static* constraints consider only single states of a database, they express state–independent properties that must hold at any state of the database and depend only on the current state, independently of any previous states of the database,

- *dynamic* constraints allow to constrain state transitions, i.e., they specify which transformations of database states into new ones are allowed. They allow expressing conditions over sequence of two or more database states. There is a particular case of dynamic constraints called transition constraints. A transition constraint imposes restrictions on pairs of states, the before and after state of a transaction.

Finally, the third classification of consistency constraints determines which properties of entities can be subject to constraints:

- *state* constraints specify conditions on the values (states) of entities,

- *behavioral* constraints further allow to constrain the behavior of entities, e.g., specify correct semantics of methods.

Since the database is expected to be consistent with respect to these properties, one important issue is the enforcement of integrity constraints upon updates. Integrity enforcement deals with the prevention of semantic errors made by users due to their carelessness or lack of knowledge.
Integrity checking is the process of verifying that a given update satisfies the constraints. If a constraint is violated, then the update is rejected. Otherwise the update is accepted.
Integrity maintenance is a process that also starts with a given update and the constraints but now, if some constraint is violated, an attempt is made to find a repair, that is, an additional set of insertions and/or deletions of facts to be added to the update, such that the resulting update satisfies all integrity constraints.
Based on the time in which the method can be applied to enforce the constraints, there are two approaches. The first is a *run–time* approach that must be carried out in the time the update takes place. It takes into account the information provided by the update request and the contents of the database. The second is a *compile–time* approach that is based on meeting a solution to enforce constraints at compile–time, i.e. before the time of update execution. In this case, production rules, ECA (Event–Condition–Action) rules or transaction programs are generated from a database schema.
The simplest solution of static constraint enforcement would be to evaluate all constraints whenever the database is updated. However this naive approach is impractical, because it is too costly and highly redundant. To avoid such situation all practical methods are based on the assumption that constraints are known to be satisfied prior to the update. Only a subset of the constraints needs to be verified after the update, namely those that are affected by it. Then, given a particular update, methods derive simplified conditions of the constraints such that, if the database satisfies the simplified conditions, it is guaranteed that the database

will be consistent after the update.
Many works [8, 9, 5, 6, 21, 15, 14, 27, 28, 29, 30] have proposed that the technology of active databases provides a natural framework for implementing integrity enforcement through repairing actions. The predominant paradigm for active databases is that of ECA rules which are triggered by an event and their action is executed only if a condition is met.
There are some features that are important for constraint enforcement approaches:

- *Kind of constraint*: this feature indicates if the method deals with dynamic or static constraints.
- *Kind of the database considered*: the kind of the database used as basis to develop the method. In this paper, we present methods related with the relational, object–oriented and active databases.
- *Kind of constraint enforcement*: which indicates if a method is an integrity checking method or an integrity maintenance method.
- *Time of constraint enforcement*: this feature refers to the instant in which the method is applied. Enforcement can be performed completely at compile–time, completely at run–time, or partially at compile–time and partially at run–time. However, compile–time methods have the advantage over runtime methods because performance is less critical at compile–time.
- *Constraint language used*: the language to specify the constraints. In the case of transition constraints, it is possible to use a first–order logic formula, the same commonly used for static constraints, extended by operators such as *old* or *new*.
- *Implementation*: which indicates if the proposed approach is implemented or not.

In the next sections we present the approaches with respect to above features.

2 Approaches

2.1 The approach of Ceri and Widom

Ceri and Widom [8] propose an approach for static constraint enforcement in relational databases. Authors describe an SQL–based language for defining integrity constraints and a framework for translating these constraints into constraint–maintaining production rules. Some parts of the translation are automatic while other parts require user intervention. Based on the semantics of represented set–oriented production rules language, authors prove that at the end of each transaction the rules are guaranteed to produce a state satisfying all defined constraints. Constraints are expressed through a declarative language, whose syntax is similar to SQL. Once constraints have been declared, they can be translated into a set of event–condition–action (ECA) rules that enforce them. Each rule has

- an event part, containing a so–called *transition predicate*, that lists all the data manipulations potentially violating the constraint,
- a precondition, an arbitrary predicate which is true whenever the constraint is violated,

- an action, an arbitrary list of SQL statements whose execution may correct the constraint's violation.

Rule execution semantics is the same as in active databases [25, 12, 11, 18, 19]. **Figure 1** shows the general structure of proposed constraint enforcement system, which automatically, at compile–time, produces rule templates–portions of the production rules necessary for constraint enforcement. The system includes:

- *Producing rule templates from constraints.* Rule templates enumerate all operations that may cause constraint violation (these form the triggering components of the constraint–enforcing rules) and include rule conditions. Rule actions are provided by the user.

- *Detecting potential cycles in rule activation.* As the number of rules increases and rules become more complex, there is increasing possibility of infinite triggering behavior. This component detects the potential for such behavior and provides warnings to the user.

- *Rule optimization.* The system automatically optimizes rules derived from constraints, preserving their constraint–maintaining semantics. It is possible to evaluate a rule's condition only over the changes that have occurred since the constraint was last considered. For example, suppose a constraint may be violated when tuples are inserted into a table. It may be sufficient for the condition part of the rule enforcing the constraint to check only those tuples that have been inserted, rather than checking the entire table. This component transforms rule conditions to incorporate this kind of optimization.

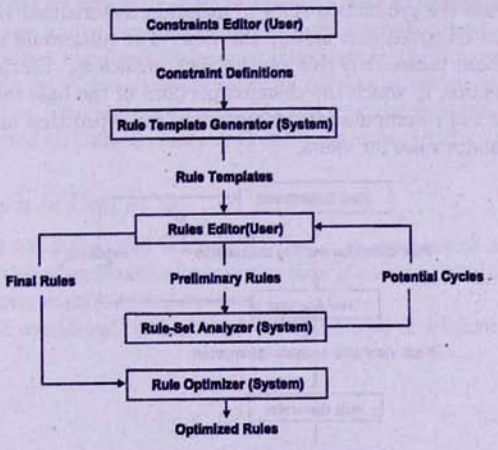

Figure 1. **Interactive System for Rule Derivation**

Rule generation and analysis are only partially automated. Only the event and precondition part of rules are generated, the action part of rules has to be added manually by the designer. An analysis of rules determines potential sequences of rules, which can trigger each other indefinitely. This information is given to the designer, who is then responsible

for taking appropriate actions. This work considers a compile–time environment. When transactions are determined to have potential for constraint violation, feedback is provided to the user. Finally, a theorem is proven stating that the final set of production rules is guaranteed to maintain all defined constraints. That is, at the end of every transaction, rule execution terminates in a consistent state.

2.2 The approach of Ceri and Widom

In [9] Ceri and Widom propose a facility whereby a user defines a view as an SQL select expression, from which the system automatically derives setoriented production rules that maintain a materialization of that view. The maintenance rules are triggered by operations on the view's base tables. Generally, the rules perform incremental maintenance: the materialized view is modified according to the sets of changes made to the base tables, which are accessible through logical tables provided by the rule language. However, for some operations substantial recomputation may be required.
In relational database systems, a view is a logical table derived from one or more physical (base) tables [10]. Views are useful for presenting different levels of abstraction or different portions of a database to different users. Typically, a view is specified as an SQL select expression. A retrieval query over a view is written as if the view were a physical table, the query's answer is logically equivalent to evaluating the view's select expression, then performing the query using the result. There are two well–known approaches to implementing views. In the first approach, views are virtual: queries over views are modified into queries over base tables. In the second approach, views are materialized: they are computed from the base tables and stored in the database.
Authors suggest to use the production rules to maintain materialized views: when base tables change, rules are triggered that modify the view. The rules could simply rematerialize the view from the base tables, but this can be very inefficient. Efficiency is achieved by incremental maintenance, in which the changed portions of the base tables are propagated to the view, without full recomputation. A system is developed that automatically derives incremental maintenance rules for views.

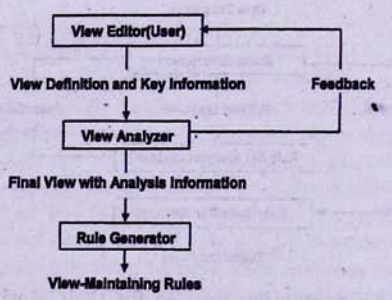

Figure 2. Rule derivation System

Figure 2 shows the structure of proposed system, which is invoked at compile-time when a view is created. Initially, the user enters the view as an SQL select expression, along with information about keys for the view's base tables. The system then performs syntactic

analysis on the view definition, this analysis determines two things: whether the view may contain duplicates and for each base table referenced in the view, whether efficient view maintenance rules are possible for operations on that table. The user is provided with the results of this analysis. The results may indicate that, in order to improve the efficiency of view maintenance, further interaction with the system is necessary prior to rule generation. Once the user is satisfied with the view definition and its properties, the system generates the set of viewmaintaining rules. Rules are produced for insert, delete, and update operations on each base table referenced in the view. The rules are defined by rule language of Starburst database system [34, 32, 33].The syntax for defining rule is:

create rule name **when** transition predicate [**if** condition] **then** action

A rule is triggered by a given transition when its transition predicate holds with respect to that transition. Transition predicates specify operations on particular tables and columns. Once rule is triggered, it may be chosen for evaluation. At this point, the rule's condition is checked rule conditions are arbitrary predicates on the database state. If the condition is true, the action is executed. An action may specify a list of SQL data manipulation operations to be executed or it may request a rollback of the current transaction.
The used rule language is set-oriented, meaning that rules are triggered after arbitrary sets of changes to the database. For those operations, which the system has determined that efficiency is possible, the maintenance rules modify the view incrementally according to the changes made to the base tables. For those operations for which efficiency is not possible, rematerialization is performed. This method is applicable for simultaneous maintenance of multiple views.
This work is related to that in [8], where a method for deriving production rules is given, that maintains integrity constraints. These solutions to the two problems differ considerably, but the approaches are similar: in both cases a general compile-time facility is described in which the user provides a high-level declarative specification, then the system uses syntactic analysis to produce a set of lowerlevel production rules with certain properties relative to the user's specification. Theorems are proven stating that the final set of generated rules is guaranteed to maintain all defined constraints and keep views irredundant.

2.3 The approach of Ceri et al.

[6] originates from work of Ceri and Widom [8]. Ceri et al. propose an approach of integrity maintenance, consisting of automatically compile–time generating production rules for static and transition integrity constraint enforcement.
The general problem considered by Ceri et al. is represented in **Figure 3.**

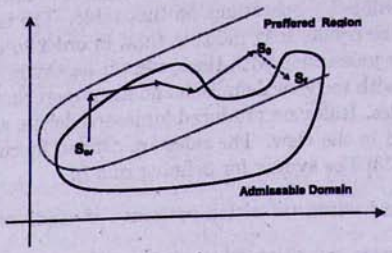

Figure 3. **A pictorial view of the integrity enforcement problem**

Database state is represented as a point in an n–dimensional space, integrity constraints partition the space of possible states into two distinct regions, an "admissible" region (in which all constraints hold) and a "forbidden" region (in which one or more constraints are violated). All transactions apply to an original legal state S_{or}. Incorrect transactions lead to a forbidden state ($S_{or} \rightarrow S_0$). The traditional response of a DBMS to an incorrect transaction is to roll back the transaction. In this approach the database system reacts autonomously to inconsistencies, by triggering a set of repair actions capable of elimination of constraint violations until a consistent state is reached (this corresponds to the database update $S_0 \rightarrow S_f$). In some cases, repair is impossible, and an abort is forced by some rules, yielding state S_{or}.
An additional requirement is that the final state be chosen within a subspace of states as "close" as possible to the original intention of the transaction supplier. The state resulting from the composition of the transaction and the constraint repair actions should, therefore, belong to the intersection of the admissible region and the region describing the preferences of the transaction supplier.
In **Figure 4** architecture of a proposed system – Integrity Maintenance System is presented (similar to interactive system in [8]).

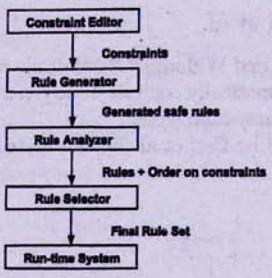

Figure 4. **Architecture of the integrity maintenance system**

The system includes:

- *Constraint Editor*, by means of which a user can introduce all constraints.

- *The Rule Generator* is applied to all constraints and produces a large set of active rules, called the maximal rule set that can be used to enforce them. This set may be modified by the user, to add specific domain knowledge.

- *The Rule Analyzer* is a compile–time component that selects a rule set included in the maximal rule set and determines a partial order on the constraint set. Rules produced after this step are guaranteed to terminate in a final state so that all violated constraints are corrected, with the possible exception of few elements, called *critical constraints.*

- *The Rule Selector* is a postoptimization step, possibly user assisted, which generates a new rule set, by excluding some redundant rules for noncritical constraints and including at least one rule for each critical constraint. However, the new critical rules introduced for repairing critical constraints are treated in a special way at run–time.

- *The Run–Time System* is responsible for execution control after a user–supplied transaction. All critical rules are executed in a experimental way, if their execution leads to a violation of a higher–level constraint, thereby introducing the potential for a nonterminating execution, then the transaction is immediately aborted, noncritical rules are instead executed in a conventional way.

Basic concepts from Relational Calculus and Relational Algebra are used in definitions of constraint specification and enforcement languages. Domain Relational Calculus is adopted as the underlying query language in the expression of constraints and repair actions.
Rule analysis selects a rule set and determines a partial order on constraints. The rules selected after this step are guaranteed to terminate in a final state such that all violated constraints are corrected, with the exception of the bottom elements of the partial order, such elements are called *critical constraints.* The proposed approach to guarantee system termination is that of imposing a particular priority order on constraints and applying only the repair actions that, while enforcing a constraint, do not violate higher–priority constraints. In this way, cycles among repair actions are prevented. The goal of rule analysis is to find the *best* constraint order, i.e., the order that guarantees that most constraints will be repaired by active rules and that the selected rules reflect the user's wishes. Selecting rules is an NP–hard optimization problem. That's why authors develop the heuristic method for rule analysis via *Triggering Hypergraph.*
A Triggering Hypergraph (THG) is defined as a pair $H : (V, A)$, where V is a finite set of *vertices* representing constraints, and A is a finite set of *hyperarcs*, representing rules, of the form (v, V') where v is a vertex and V' a possibly empty subset of V. v is called the *tail* of the hyperarc, $V' : v_1 ... v_n$ the *head set.* A hyperarc $(\vartheta, \{\vartheta_1, \ldots, \vartheta_n\})$, uniquely labeled as R, connects constraint ϑ to a set of constraints $\{\vartheta_1, \ldots, \vartheta_n\}$ if and only if R is a rule that enforces ϑ and can introduce violations of all the constraints listed in $\{\vartheta_1, \ldots, \vartheta_n\}$. The case of a hyperarc with an empty head set represents a rule that cannot violate any constraint. Rules that have the abort operation in the action side are represented by hyperarcs of this kind.
The rules lying on a cyclic path of THG can trigger each other indefinitely and cause the nontermination. The presented approach to termination consists of selecting a set of repair actions, so that the THG is reduced to a directed acyclic hypergraph (DAHG). If only the rules in the DAHG are executed at run–time, termination is guaranteed. The reduction of the THG to a DAHG is performed by selecting the less suitable rules to be monitored at run–time. The semantic knowledge of adequacy of a rule is encoded into a numerical value,

called the repair action "weight", the greater the weight, the more preferable the repair action. When rules used for repair are included in the DAHG, termination is ensured, when, instead, rules not included in the DAHG are used, then additional run-time monitoring is required.
This paper contains the core technical material, describing the architecture of proposed rule system and the full detailed descriptions of rule generation and analysis algorithms.

2.4 The approach of Urban et al.

Urban et al. in [27, 28, 29, 30] works propose the approach similar to that in the [6], in the context of deductive object–oriented databases. Constraints are expressed as clausal formulas of a first–order logical language. Repair actions are encoded as ECA rules with a particular format: several condition–action pairs may be associated with a single triggering event. Triggering events correspond to the elementary operations on objects: object creation and deletion, single–valued/multivalued property modification. The same operations can be used in the action side of a rule, which may also contain composite operations and I/O operations. The core issue is the transformation of a set of declaratively expressed constraints into a set of production rules, called *Integrity Maintenance Production Rules* (IMPRs) that are executed by an active DBMS.
The generation of IMPRs from constraints is supported by a tool called *CONTEXT* [29]. The phase of *Constraint Analysis* derives, from the analysis of related constraints, semantic information that drives the phase of IMPR generation. This strategy proposes repair actions to the designer who keeps the responsibility to decide which IMPRs to generate. Multiple repair actions for a single constraint are allowed.
A variant of the Triggering Graph introduced in [6], is used to evaluate a set of IMPRs to see if they may cause contradictory updates and infinite triggering. In addition to nodes representing rules, the graph contains a node for every possible elementary operation. Two kinds of arcs are defined: *propagation* arcs connect an operation or a rule to another rule, iff the former can generate the triggering event of the latter, *conflict* arcs link a rule to another rule or to an operation, iff the action of the former contradicts the latter. Direct cycles contains only propagation arcs denote potential infinite triggering, while cycles containing at least one conflict arc show the possibility of conflicting updates. Cycles are *essential* if they always generate anomalous IMPR behavior, *inessential* if it can be demonstrated, based on IMPRs semantics, that infinite triggering or conflicting updates do not arise at run–time, *instance based* if the occurrence of anomalies depends on the actual transaction and database instance.
The goal of IMPR analysis is that of identifying inessential cycles and to provide information to the run–time executor of IMPRs, to reduce the probability of repair failure. However, no problem-solving tool is provided to support IMPR analysis.

2.5 The approach of Schewe et al.

Schewe et al. [26] address the integrity enforcement problem with respect to static and transition constraints. Their analysis is based on object–oriented databases with a clear distinction between values and objects. They use formulae in first–order logic to express both static and transition constraints. The approach adopted is that of modifying a user's transaction, by replacing the (basic) methods that update objects with rewritten ones, which

satisfy the established constraints. Such rewritten methods constitute the *Greatest Consistent Specialization* (GCS) of database–updating operations. A GCS is defined as a derived operation that extends the original one to make it compliant to a single constraint. A GCS can add to the original operation extra updates, similar to the repair actions of [8] or [6], provided that they do not contradict the original operation, and the "distance" between the original and the extended operation must be minimal, this means that any other consistent specialization of the original operation must be *more specific* than the GCS. There is significant difference between repair rule and GCSs: the rule is triggered only if the constraint is violated, while the GCS represents a conditional update that, in any state, guarantees consistency.
The notion of repair failure is also present: when it is impossible to derive a consistent specialization of the original operation that respects the constraints, the GCS is conventionally set to a *fail* operation. GCSs are built manually. A negative result shows that GCS generation does not scale up trivially: it is not possible to build GCSs of arbitrary methods or transactions by simply "assembling" GCSs of elementary operations. Also, building GCSs for multiple constraints is a problem, since GCSs for a single constraint are usually not elementary operations.

2.6 The approach of Jagadish and Qian

In [20] authors present an approach for constraint enforcement in object–oriented databases through integrating inter-object constraint maintenance into an object-oriented database system. They develop a constraint compilation scheme that accepts declarative global specification of constraints, including *relational integrity, referential integrity*, and *uniqueness* requirements, and generates an efficient representation that permits localized processing. The feasibility of this approach is demonstrated by designing a constraint preprocessor for O++ [2], the programming language interface to the Ode [?, 17] object-oriented database.
The recommended approach in an object-oriented database is to associate constraints with classes, and upon the update of an object to check each constraint associated with its class and none others. The constraint compilation approach that's presented here generates efficient representations and localized consistency maintenance, by appropriately transforming a specified declarative constraint and associating it with exactly the relevant set of class definitions, where each of a small number of relevant constructs can efficiently be checked.
Here a language CIAO++ is introduced, which is a small extension of O++, suitable for declaratively expressing integrity constraints. The constraint facility provided in O++ is *intra–object* in that when an object is updated only the constraints associated with it, through its class definition, are checked. The problem is to implement each inter-object constraint as an equivalent set of (intra-object) constraints to be associated with the appropriate class definitions, that need be checked only when an object of that class is updated, created, or deleted.
A transformation technique is developed that correctly associates an inter-object constraint with the appropriate classes. It transforms an inter-object constraint into a logically equivalent one such that all objects referenced explicitly in the constraint expression are "brought to attention" in the quantification. In case of implicit object references the constraint is associated with the class of each of them. This step captures all implicit object references, provided it is applied recursively through all function definitions encountered. Any separately compiled functions must declare what classes they refer to, and the constraint is

associated with each of these classes.
By performing of this transformation all kind of inter–object constraints are maintained. To provide more efficient representations of the constraints the optimizations are presented that may be applied to the canonical representation of a constraint after it has been instantiated in a class definition.
In the proposed approach only integrity checking is considered no repairing actions are performed in presence of violations and enforcement is performed completely at run–time.

2.7 The approach of Oakasha et al.

In [23] Oakasha et al. present concepts and ideas of another approach for consistency management in object-oriented databases. Authors believe that a consistency management subsystem (CMS) for OODBMSs should have the following features:

- *Object-orientation.* CMS should work with basic principles of object-orientation such as inheritance and encapsulation.
- *Specification.* Constraints should be specified declaratively and structured as user accessible objects.
- *Interrelated Objects.* CMS should be able to maintain consistency of a large number of interrelated objects with complex structures.
- *Transactions.* CMS should work with advanced types of transactions such as interactive and long duration transactions.
- *Efficiency.* The constraint checking should rely on optimization techniques for improving constraint evaluation.
- *Integrity Independence.* CMS should be capable to change constraint specifications without changing application programs and update transactions.
- *Inconsistency.* CMS should control inconsistency of objects.
- *Persistence Style.* CMS should maintain consistency of all objects, that is persistent and nonpersistent ones, and regardless of the persistence style of OODBMSs.
- *Disabling and Enabling of Constraints.* CMS should provide the tools for enabling and disabling constraints at various levels of abstraction like the whole database, or a class or a specific object.
- *Update Granularity.* CMS should work with different levels of update granularity of objects such as updating a simple attribute or a complex attribute or the whole state of an object.

In this work authors propose an approach that support the features listed above.
Main aspects of this approach are the *constraints catalog* and a novel technique for constraints structuring. The constraints catalog is a meta-database acting as repository of constraints specifications. The main purpose of the constraints catalog is to separate the constraints specification from transactions and application programs and hence to provide the feature of integrity independence to this approach. The constraints are defined as first-order logic

formulas. The information stored in the constraints catalog is divided into two categories. The first category is about specification of constraints. This kind of information is stored in a class named **IC**. The second category of information describes definition of constraints. This type of information is stored in a class named **Shell**. To avoid the drawbacks concerning redundant checkings and specific objects handling shells are associated with another type of objects called *constraint kernels*. This type of information is stored in a class named **Kernel**. Kernels store information that is local to objects and which is common among interrelated objects.

For consistency maintenance every class in the database schema is augmented with a set of *integrity control* methods. These methods are used for object manipulation and consistency maintenance. Their semantics includes all tasks of integrity control, that is:

- linking each newly created object with shells and kernels,
- unifying kernels among shells of interrelated objects,
- monitoring updating to objects states and checking immediate constraints,
- checking violated constraints locally w.r.t. an object and globally w.r.t. all objects that are updated by a transaction,
- disabling or enabling constraints.

For consistency maintenance only constraint checking is considered and constraint enforcement is performed completely at run-time.

This approach of consistency enforcement can be extended by the method of *integrity independence* proposed by Oakasha and Saake [24]. Applying the feature of integrity independence means that constraints can be modified without recompiling updating transactions and applications. The key aspect of this approach is *constraint handler* (**Figure 5**). Connection between the class C and constraints $W_1 \dots W_n$ is established via constraint handler H_C. It becomes responsible for controlling all aspects of constraints $W_1 \dots W_n$.

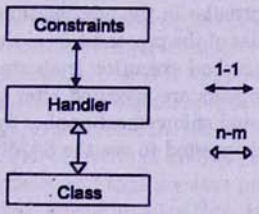

Figure 5. **Relationships among Classes, Handlers and Constraints**

2.8 The approach of Geppert and Dittrich

Geppert and Dittirch [15] propose an approach to the specification and implementation of consistency constraints in object–oriented database systems, adopting the programming–by–contract (PbC) paradigm developed for object–oriented programming. This approach supports implicit, dynamic (two–state transition) and behavioral consistency constraints. PbC is an approach to achieve correct and robust object-oriented, modular programs. In

addition to the usual parts of class definitions, preconditions, postconditions, and class invariants are specified. Together, preconditions and postconditions define a *contract*:

- the precondition defines the conditions the sender of a message has to obey,
- the postcondition defines what the receiver is obliged to produce.

Constraints that are not specific for a method but restrict the permissible states of any object of a class can be defined through *invariants*. Consistency constraints in PbC–DB are expressed through invariants, pre– and postconsditions.
This approach is defined on the object model as provided by most OODBSs [1] (defining objects, classes, methods, extensions etc.). Class definitions and method signatures are augmented by special clauses that define invariants, pre– and postconditions:

invariant name formula [**repair** action]{, name formula [**repair** action]};
require name formula [**repair** action]{, name formula [**repair** action]};
ensure name formula [**repair** action] {, name formula [**repair** action]};

Formulas are written as conjunctive forms. Each conjunct is a disjunction of predicates. Predicates can also contain range–restricted, quantified predicates. The only free variables permitted in formulas are the instance variables of the class. Methods are also permitted in formulas, as long as they do not perform modifications on the state of objects. *Inter–object* constraints are expressed as formulas, that refer to methods and instance variables of other objects of other classes (encapsulation violation in constraint definitions is permitted). The **repair** clause specifies an action to be performed whenever the invariant is violated. Actions are code fragments in the object model's data manipulation language. The user is responsible for consistent/inconsistent object state resulting from repair action.
Consistency is maintained on three levels: for objects, extensions, and the entire database. At each level the role of constraints is different, but the enforcement semantics is the same and is defined as follows. Whenever a method m of the object o is called, the precondition of the m is checked. If none of the formulas in the precondition evaluates to false, the method body is executed. If a set of formulas of the precondition evaluates to false and if any of them has no repair action defined, the method execution is aborted. If all formulas evaluating to false have repairs defined, these repairs are executed after the condition check. The same procedure for constraint checking and enforcement applies to postconditions and invariants. For implementation purposes it's suggested to use the SAMOS [16] ADBMS.

2.9 The approach of Ceri et al.

Ceri et al. [5, 7] present a proposal for constraint enforcement in **Chimera**. **Chimera** is a novel database language jointly designed by Ceri et al. at Politecnico di Milano. **Chimera** integrates an object–oriented data model, a declarative query language and an active rule language for reactive processing. In these papers, the proposal for constraint management in **Chimera** is presented, which relies on the declarative specification of passive constraints as rules and on their transformation into different types of active rules.
To face violations of a constraint by an update transaction, two approaches are traditionally available: either the transaction checks the integrity of data before committing and undertakes repair actions in case of violations, or the DBMS independently detects the violations

(at least for a limited variety of constraints) and rolls back the incorrect transaction. Authors claim that the former approach is unsatisfactory since it scatters the constraint enforcement criteria among the applications, thus jeopardizing data integrity and compromising change management, whereas the latter does not present this problem but is not suited to model complex semantic constraints. In **Chimera** a more flexible approach is presented where constraints can be handled at different levels of sophistication:

- At the simplest level, constraints are expressed declaratively through deductive rules, whose head defines a constraint predicate that can be used by the transaction supplier to query constraint violations, integrity is then enforced manually by the transaction supplier.
- At the next level, constraints are encoded as triggers, activated by any event likely to violate the constraint. Their precondition tests for the actual occurrence of violations, if a violation is detected, then the rule issues a rollback command. They are called *abort rules*. Abort rules can be syntactically generated from declarative constraints.
- At the most sophisticated level, constraints are encoded as triggers having the same precondition and triggering events as abort rules, their action part, however, contains database manipulations for repairing constraint violations. **Chimera** provides adequate tools to support above defined *maintenance rules* generation from declarative constraints and their analysis for termination and correctness (ability to achieve a consistent state for any input transaction as introduced in [8, 6]).

2.10 The approach of Medeiros and Andrade

In [21] Medeiros and Andrade present a method for static constraint enforcement that automatically transform integrity constraints into a set of ECA–rules.
The considered problem of derivation of production rules from constraints can be defined as follows:
Let a constraint be specified as a first order logic predicate P over a database state, and A some userdefined action to be performed if P is not true (i.e. if $\neg P$ then A). The same predicate 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 $< \{E\}, \neg P, A >$ to maintain this constraint. Whereas the P and A components can be derived straight from the constraint specification, event determination depends on additional information. Thus, the most complex part that of automatic rule derivation is that of determining events set, which may violate a constraint. The objective of event derivation is thus avoid unnecessarily checking constraints at every update, and thus are save processing time.
Unlike relational database systems, where updates that may violate a given constraint can be usually determined by static analysis of the constraint, in an object-oriented system, updates are performed by methods to obey encapsulation. This complicates the definition of all events, which may be associated with integrity violation. In relational systems, events consist of applying the *insert*, *delete* and *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 is encapsulated into class and cannot be handled unless additional context sensitive information is provided. Thus, algorithms for transforming constraints into rules such as proposed by [35, 8] do not apply to object-oriented systems. For that purpose authors propose an approach, similar to that in the

[6], where the transformation from integrity constraints into production rules is defined on notions of active object-oriented database. For constraint definition declarative constraint specification language is developed. This language is based on first order logic, and allows defining constraints over schema components, objects or classes.

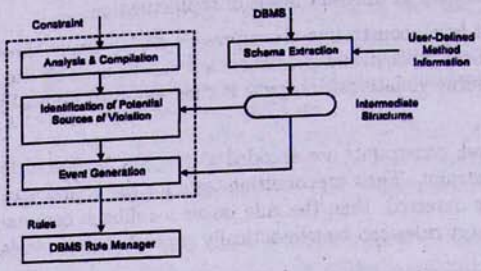

Figure 6. **Constraint Transformation System**

The constraint subsystem was implemented on top of the active version of the O_2 object-oriented DBMS [22]. The rule generation algorithm needs different types of information from the database schema, as well as semantic information about methods. The implemented system consists of 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 schema extraction modules. **Figure 6** 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 & Compilation* parses the constraint and builds the condition and action fields of the production rule. It passes the result on to the module that performs *Identification of Potential Sources of Constraint Violation*. Finally, *Event Generation* determines the set of events.

2.11 The approach of Benzaken and Schaefer

The basic assumption that Benzaken and Schaefer made in [4] is that the run-time checking of constraints is too costly to be undertaken systematically. Therefore, methods that are always safe problems with respect to integrity constraints should be proven at compile-time. The run-time checks should only concern the remaining methods. To that purpose, they propose a new approach of static management of integrity constraints, based on abstract interpretation, to prove the invariance of integrity constraints under complex methods.
This method undertakes a very detailed analysis of methods and provides some precise information on the impact they have upon constraints. Partial but concepts, reliable information

concerning methods is obtained by means of a *predicate transformer*. A predicate transformer is a function that, given a method m and a constraint C satisfied by the input data of m, returns a formula $m'(C)$ that is satisfied by the output data of m. In other words, provided C is satisfied before an execution of m, $m'(C)$ is satisfied after an execution of m. A method m is then safe with the respect to constraint C if $m'(C) \Rightarrow C$ (C is a consequence of $m'(C)$). To prove the implication, a classical theorem proving technique based on the tableaux method is used. The tableaux method is a denial method. In order to prove that $F \Rightarrow G$, that method proves in fact that $F \wedge \neg G$ is unsatisfiable. In order to do that, it develops a tree, called a *tableau*, which has several branches. A tableau is of the form:

$$[branch_1, branch_2, \ldots, branch_n]$$

Each branch is of the form:

$$[formula_1, formula_2, \ldots, formula_m]$$

Each branch is a conjunction of formulas, and a tableau is a disjuncntion of branches. The tree use is expanded only up to a certain "depth". Then, a "test for closure" is done on that tree, which amounts to an instantiation of free variables that reduces the tableau to a contradiction. Automatic theorem proving techniques in details can be found in [13].
The simple extension of O_2 [22] programming language is used to define integrity constraints, which are well-formed formulas on a specific, first-order logic.

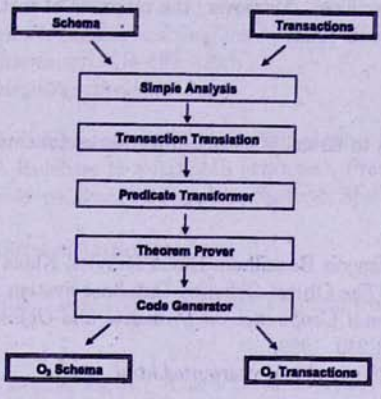

Figure 7. **Structure of the Integrity Constraint Manager**

The proposed approach is implemented as a pre-processor for O_2. It provides O_2 with an integrity constraint management facility. **Figure 7** shows the general structure of the integrity constraint manager. First, a simple syntactical technique is used to show that some methods are safe with respect to some constraints. That component is a preliminary filter that only considers type information from the constraints and the methods. Roughly, it compares the types of the objects that are updated in each method with the types of the objects that are constrained by each constraint.
After that, for methods that have not been proven safe, the more sophisticated method of predicate transformers is applied in different steps:

- Methods and constraints are decomposed. In particular, methods are the translated into the intermediate form to make the task of abstract interpretation easier.
- For each method and each constraint, the predicate transformer is applied. Which method has to be checked with which constraint is determined by the first step mentioned previously.
- An automatic first-order theorem prover based on the tableaux method is applied to the result provided by the predicate transformer.
- The code generator finally takes into account the results of the analysis at to generate the O_2 schema (if needed) and the O_2 methods.

3 Conclusion

In this paper we present a review of approaches to integrity constraint enforcement in database systems. The illustrated methods have classified and compared with respect to some characterizing features. We concluded that that the technology of active databases provides a natural framework for implementing integrity enforcement through repairing actions. Regardless of this fact, most approaches propose checking instead of maintenance of the database integrity, i.e. they abort or rollback the transaction in case of violations and repair actions are not undertaken. Moreover, the majority of methods perform constraint enforcement completely at run–time.

Acknowledgments
The author is very grateful to Hrant Marandjian for his useful comments and suggestions.

References

[1] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier and Stanley Zdonik, "The Object-Oriented Database System Manifesto", *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, pp. 223-240, 1989,
citeseer.nj.nec.com/atkinson89objectoriented.html.

[2] Rakesh Agrawal, Shaul Dar and Narain H. Gehani, "The O++ Database Programming Language: Implementation and Experience", *ICDE*, pp. 61-70, 1993,
citeseer.nj.nec.com/dar93database.html.

[3] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): the language and the data model", pp. 36-45, 1989,
citeseer.nj.nec.com/agrawal89ode.html.

[4] Véronique Benzaken and Xavier Schaefer, "Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Transformers", *Lecture Notes in Computer Science*, vol. 1241, pp. 60-??, 1977,
citeseer.nj.nec.com/article/benzaken97static.html.

[5] Stefano Ceri, Piero Fraternali and Stefano Paraboschi, "Constraint Management in Chimera", *IEEE Data Eng. Bull.*, vol. 17, no. 2, pp. 4-8, 1994.

[6] Stefano Ceri, Piero Fraternali, Stefano Paraboschi and Letizia Tanca, "Automatic generation of production rules for integrity maintenance", *ACM Trans. Database Syst.*, ACM Press, vol 19, no. 3, pp. 367-422, 1994.

[7] Stefano Ceri and Rainer Manthey, "Chimera: A Model and Language for Active DOOD Systems", *East/West Database Workshop* ,pp. 3-16, 1994, *citeseer.nj.nec.com/ceri94chimera.html.*

[8] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance", *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, D. McLeod and R. Sacks-Davis and H. Schek, pp. 566-577, 1990, *citeseer.nj.nec.com/ceri90deriving.html.*

[9] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance",*Proceedings of the 17th Conference on Very Large Databases,* (Los Altos CA), Barcelona, Morgan Kaufman, 1991, *citeseer.nj.nec.com/ceri91deriving.html.*

[10] C. J. Date, "Database Systems", Addison-Wesley, 2000.

[11] Umeshwar Dayal, Eric N. Hanson and Jennifer Widom, "Active Database Systems", *Modern Database Systems*, pp. 434-456, 1995, *citeseer.nj.nec.com/dayal94active.html.*

[12] K. R. Dittrich and S. Gatziu and A. Geppert, "The Active Database Management System Manifesto: A Rulebase of a ADBMS Features", *Proceedings of the 2nd International Workshop on Rules in Database Systems*, Springer, vol. 985, pp. 3-20, 1995, *citeseer.nj.nec.com/dittrich95active.html.*

[13] M. Fitting, "First-Order Logic and Automated Theorem Proving ", SpringerVerlag, 1990.

[14] Piero Fraternali and Letizia Tanca, "A structured approach for the definition of the semantics of active databases ", *ACM Trans. Database Syst.*, ACM Press, vol. 20, no. 4, pp. 414-471, 1995, *http://doi.acm.org/10.1145/219035.219042.*

[15] Andreas Geppert and Klaus R. Dittrich," Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract" , *Datenbanksysteme in Buro, Technik und Wissenschaft*, pp. 322-337, 1995, *citeseer.nj.nec.com/geppert95specification.html.*

[16] Stella Gatziu and Andreas Geppert and Klaus R. Dittrich, "The SAMOS active DBMS prototype", pp. 480-480, 1995, *citeseer.nj.nec.com/gatziu94samo.html.*

[17] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proceedings of the 17th Conference on Very Large Databases*, (Los Altos CA), Barcelona, Morgan Kaufman, 1991,
citeseer.nj.nec.com/gehani91ode.html.

[18] Theodore Hong, "A Survey of Active Database Systems", 1997,
citeseer.nj.nec.com/hong97survey.html.

[19] Ulrike Jaeger and Johann Christoph Freytag, "An Annotated Bibliography on Active Databases", *SIGMOD Record*, vol. 24, no. 1, pp. 58-69, 1995,
citeseer.nj.nec.com/article/jaeger95annotated.html.

[20] H. V. Jagadish and X. Qian, "Integrity Maintenance in Object-Oriented Databases", *Proceedings of the 18th Conference on Very Large Databases*, Los Altos CA), Vancouver, Morgan Kaufman, 1992,
citeseer.nj.nec.com/jagadish92integrity.html.

[21] C. Medeiros and M. Andrade, "Implementing Integrity Control in Active Databases", *Implementing Integrity Control in Active Databases . The Journal of Systems and Software*, december, pp. 171-181, 1994,
citeseer.nj.nec.com/medeiros94implementing.html.

[22] C. Medeiros and P. Pfeffer, "Object Integrity Using Rules", *n Proceedings European Conference on Object-Oriented Programming*, pp. 219-230, 1991.

[23] H. Oakasha and S. Conrad and G. Saake, "Consistency management in object-oriented databases", *Concurrency and Computation: Practice and Experience*, vol. 13, no. 11, pp. 955-985, 2001,
citeseer.nj.nec.com/296653.html.

[24] Oakasha, H. and Saake, G., " Integrity Independence in Object-Oriented Database Systems ", *Kurzfassungen — 10. Workshop "Grundlagen von Datenbanken", Konstanz (02.06.–05.06.98)*, Universität Konstanz, Fachbereich Informatik, M. H. Scholl and H. Riedel and T. Grust and D. Gluche, no. 63, pp. 94-98, 1998,
citeseer.nj.nec.com/oakasha98integrity.html.

[25] N.W. Paton and O. Diaz, " Active Database Systems ", *ACM Computing Surveys*, vol. 1, no. 3, pp. 63-103, 1999,
citeseer.nj.nec.com/paton99active.html.

[26] K.-D. Schewe, and B. Thalheim, and J.W. Schmidt, and I. Wetzel, "Integrity Enforcement in Object-Oriented Databases", *Proc. 4th Int. Workshop on Foundations of Models and Languages for Data and Objects* , Volkse, Germany, October, pp. 19-22, 1992.

[27] Susan D. Urban and Lois M. L. Delcambre, Constraint Analysis: A Design Process for Specifying Operations on Objects", *IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 4, pp. 391-400, 1990.

[28] Susan D. Urban, Anton P. Karadimce and Ravi B. Nannapaneni, "he Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database", *Proceedings of the Eighth International Conference on Data Engineering* ,Tempe, Arizona, IEEE Computer Society, Forouzan Golshani, pp. 565-572, 1992.

[29] Susan D. Urban and Mario Desiderio, "CONTEXT: a CONstrainT EXplanation Tool", *Data Knowl. Eng.*, Elsevier Science Publishers B. V., vol. 8, no. 2, pp. 153-183, 1992,
http://dx.doi.org/10.1016/0169-023X(92)90035-A .

[30] Susan D. Urban and Billy B. L. Lim, "An intelligent framework for active support of database semantics", *nt. J. Expert Syst.*, JAI Press, Inc., vol. 6, no. 1, pp. 1-37, 1993.

[31] Ullman J. D., Widom J., Garcia-Molina H., "Database Systems: The Complete Book", Prentice Hall, 2001.

[32] J. Widom, " The Starburst Rule System: Language Design, Implementation, and Applications ", *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, vol. 15, no. 1-4, pp. 15-18, 1992,
citeseer.nj.nec.com/widom92starburst.html.

[33] Jennifer Widom, "The Starburst Active Database Rule System", *Knowledge and Data Engineering*, vol. 8, no. 4, pp. 583-595, 1996,
citeseer.nj.nec.com/widom96starburst.html.

[34] J. Widom and R. Cochrane and B. Lindsay, "Implementing Set-Oriented Production Rules as an Extension to Starburst", *Proceedings of the 17th Conference on Very Large Databases*, (Los Altos CA), Barcelona, Morgan Kaufman, 1991,
citeseer.ist.psu.edu/widom91implementing.html.

[35] J. Widom and S. J. Finkelstein, "Set-oriented production rules in relational database systems", pp. 259-270, 1990,
citeseer.nj.nec.com/widom90setoriented.html.

Ամբողջականության ապահովումը տվյալների հենքերում. ակնարկ

Ա. Ջ. Ասատրյան

Ամփոփում

Ըստ իր սահմանման տվյալների հենքը պետք է ծառայի որպես տվյալների «վստահելի» եվ «անխոցելի» շտեմարան եվ պետք է ապահովի տվյալների կոռեկտությունը՝ հենքի հետ աշխատող կիրառական ծրագրերի համար: Սույն ակնարկում ներկայացվում են տվյալների հենքերում ամբողջականության սահմանափակումների մշակման մի շարք մոտեցումներ: Այն արտացոլում է այս բնագավառում ներկայումս կատարվող հետազոտական ուղղությունները: