Bulletin of the Technical Committee on

Data Engineering

June, 1994 Vol. 17 No. 2 **IEEE Computer Society**

Letters

Special Issue on Database Constraint Management

Conference and Journal Notices

Distribution

IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1903
(202) 371-1012

Letter from the Editor-in-Chief

The current issue of the Bulletin on database constraint management returns the bulletin to a topic that is of very high interest to the database research community. Thus far, there has not been substantial commercial interest. There is no doubt that constraint management is a hard problem. But there is also no doubt in my mind that there can be significant practical payoff to users of commercial systems. Already commercial systems support referential integrity and uniqueness constraints. In addition, they usually provide triggers, a mechanism of great power but enormous complexity. Declarative approaches to triggers, which seek to impose order on the currently unruly situation, have much in common with constraints.

Jennifer Widom, the editor for this issue, is exceptionally well qualified for the task of handling the topic of database constraint management, being herself an active researcher in this area. The issue contains contributions drawn from the first rank of researchers in database constraint managment. I wish to thank Jennifer for a fine editorial job.

New in this issue is the expansion of our "Notices" section to include both conference notices and journal notices. In that section you will see both the PDIS'94 conference call and a call from the *Transactions on Knowledge and Data Engineering* for surveys and correspondence. Further broadening of the bulletin's non-paper sections will receive further thought over the next few months. One idea is to include a calendar section devoted exclusively to database related events. I encourage you to send me your suggestions. Within our tight budget constraints, we would like to be as responsive as possible to the needs of our community.

David Lomet
DEC Cambridge Research Lab
lomet@crl.dec.com

Letter from the Special Issue Editor

The topic of this special issue is constraint management in database systems. Although rarely thought of as an especially hot or fashionable topic in the database community, constraint management has persisted throughout the years as an active ongoing area of both theoretical and practical research. I attribute this continued interest in large part to the importance and prevalence of various forms of constraints in many, if not most, practical database scenarios.

Most relational database applications rely on key constraints, non-null constraints, and referential integrity constraints. More general constraints are supported by the `assertion` feature in emerging SQL standards. Still more complex constraints frequently can be found embedded in database application code—one important goal of research in constraint management is to move constraint monitoring and enforcement out of applications and into the database system. Constraint management research also has focused on *temporal* or *dynamic* integrity constraints, on constraints in object-oriented database systems, on constraints across heterogeneous databases, on exploiting constraints to improve the power and performance of query processing in centralized and distributed databases, on efficiency issues in constraint evaluation, along with a smattering of other areas.

In putting together this issue I chose to solicit a broad collection of "synopsis" papers, rather than a small collection of in-depth papers. My goal is to give the reader a flavor of the many important aspects of database constraint management. Hence, each paper is relatively brief, but citations are provided for readers interested in delving more deeply into a particular topic.

The papers are divided into four topical groups; papers within each group are ordered alphabetically by first author's last name.

Constraint languages and implementation

The first paper, by Ceri, Fraternali, and Paraboschi, describes constraint management in the *Chimera* project at Politecnico di Milano. Constraints in Chimera are available in three styles: (1) as declarative rules that can be checked by transactions; (2) as triggers that detect and rollback constraint violations; (3) as triggers that detect and repair constraint violations.

The second paper, by Grefen, de By, and Apers, describes two approaches to constraint management taken at the University of Twente: (1) an integrity control subsystem embedded in the *Prisma* parallel main-memory relational database system; here the key feature is an approach based on *transaction modification*; (2) constraint specification and enforcement in *TM*—a functional, object-oriented database model.

The last paper in this group, by Ramakrishnan and Srivastava, outlines some results in the efficient evaluation of *constraint queries*. In addition, the paper describes an approach whereby constraints can be used as data values to represent and manipulate partially specified information in object-oriented databases.

Constraints in distributed and heterogeneous environments

The first paper in this group, by Alonso and El Abbadi, considers constraints in distributed database systems. A framework is developed for the localization of global constraints, for dynamic reconfiguration of local constraints, and for integrating constraints with distributed concurrency control using *set serializability*.

The second paper, by Chawathe, Garcia-Molina, and Widom, considers constraints in loosely coupled, heterogeneous database systems. A framework is developed for handling constraint management in autonomous distributed environments where traditional constraint management techniques are impossible. A toolkit architecture is presented that implements the approach.

The last paper in this group, by Mazumdar and Stemple, considers the problem of transaction design in the presence of integrity constraints. A theorem-proving based method is described for helping designers create transactions that respect constraints. The method is extended for distributed databases, where the goal is to find local sufficient conditions for global constraints.

Temporal/dynamic constraints

This section includes two papers, one by Chomicki, and one by Lipeck, Gertz, and Saake. Both papers are concerned with the efficient monitoring of *temporal* or *dynamic* integrity constraints—constraints specified over a sequence of database states, rather than over a single state. Chomicki describes theoretical results for constraints expressed in temporal logic, specifies a method for *history-less* checking of such constraints, and describes an implementation of the method using an active database system. Lipeck et al. propose transforming constraints expressed in temporal logic into *transition graphs*. Transition graphs can be used to evaluate the constraints based on database transitions, to generate transaction pre- and post-conditions, or to generate triggers for constraint monitoring.

Applications of constraints

Last, but hardly least, are two papers describing practical applications of database constraints. The first paper, by Griffin and Trickey, describes the redesign of the integrity maintenance component in a database-oriented telecommunications switch. Previously, large numbers of constraints on the underlying database were evaluated in a brute-force manner; the paper describes a new approach in which transactions are analyzed and modified in advance to ensure they will not violate constraints. The approach is being deployed within AT&T.

The second paper, by Gupta and Tiwari, describes constraint management in a distributed engineering design scenario. Constraints are used to specify and check interdependencies between the components of a distributed design. A system architecture is described in which constraint management features are layered on existing databases. The architecture incorporates constraint specification, compilation, checking, and notification of constraint violations.

I hope you enjoy this Special Issue.

Jennifer Widom
Stanford University
widom@cs.stanford.edu

Constraint Management in Chimera

Stefano Ceri, Piero Fraternali, Stefano Paraboschi
Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.zza Leonardo da Vinci 32, I-20133 Milano - Italy
e-mail: {ceri,fraterna,parabosc}@elet.polimi.it

Abstract

Chimera *is a novel database language jointly designed by researchers at Politecnico di Milano, Bonn University, and ECRC of Munich in the context of the Esprit Project P6333, IDEA.* **Chimera** *integrates an object-oriented data model, a declarative query language based on deductive rules, and an active rule language for reactive processing. In this paper, we present the authors' proposal for constraint management in* **Chimera***, which relies on the declarative specification of* passive *constraints as deductive rules and on their transformation into different types of* active *rules.*

1 Introduction

Constraints are declarative specifications of properties that must be satisfied in any database instance. In OODBs constraints are typically used to restrict the admissible extension of a class, to impose restrictions on the state of individual objects, or to specify general laws that must be obeyed by sets of objects possibly belonging to different classes.

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. 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** we advocate a more flexible approach where constraints can be tackled at different levels of sophistication:

1. 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.

2. 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; we call them *abort rules*. Abort rules can be syntactically generated from declarative constraints.

3. 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. We call them *maintenance rules* and provide **Chimera** users with adequate tools to support their generation from declarative constraints and their analysis for termination and correctness (ability

to achieve a consistent state for any input transaction). Such encoding of constraints into triggers was originally introduced in [CW90] for relational databases.

In the rest of this paper, each of the three levels is considered in a separate section.

2 Declarative Specification of Constraints

Chimera [CM93] is a novel database language which integrates an object-oriented data model, a declarative query language based on deductive rules, and an active rule language for reactive processing [1].

Chimera has an object-oriented model; objects are abstractions of real-world entities (such as persons, companies, or machines), distinguished by means of unique object identifiers (OIDs). The main construct for structuring schemas of **Chimera** are *classes*, defined as collections of homogeneous objects; each class has a signature and an implementation, and is described by means of several properties (its attributes, operations, constraints, and triggers).

Constraints are essentially set-oriented deductive rules integrated with object management. Differently from other approaches [GJ91], constraints are not treated as first-class objects; they may be defined in the context of a single class of **Chimera** - in which case they are called *targeted constraints* - or instead be defined in the context of multiple classes - in which case they are called *untargeted constraints*. This distinction is relevant for schema design and modularization, but there is little syntactic difference and no semantic difference between targeted and untargeted constraints.

Targeted constraints are further distinguished into object-level and class-level constraints. The former restrict attribute values of objects, whereas the latter apply to the whole extension of a class.

Each constraint is defined by a name, a list of typed output parameters and a body. The body is a condition that should *not* hold in any database instance (thus **Chimera** constraints have a negative semantics, as e.g., in [CW90]). The output parameters are a subset of the free variables that appear in the constraint body. If the evaluation of the body yields a non-empty set of bindings for these variables, these bindings are passed to the output parameters.

Example 1: An employee must not earn more than his manager

```
define constraint TooMuchPay(Emp: Employee)
  TooMuchPay(Emp) <- Employee(Emp),
                     Emp.Salary > Emp.Manager.Salary
end
```

Targeted constraints are considered part of the signature as well as of the implementation of a class. In particular, the signature of a class includes the name of all constraints targeted to that class and the name and type of their output parameters, whereas the implementation contains the passive rules that implement the constraints.

Each constraint implicitly introduces a predicate with the same name as the constraint and whose arguments correspond to the output parameters. A user's transaction can check for the presence of constraint violations by issuing a query involving the constraint predicate.

Example 2: Show the employees that earn more than their manager

```
display(Emp.Name where TooMuchPay(Emp));
```

[1] A *Chimera* is a monster of Greek mythology with a lion's head, a goat's body, and a serpent's tail; each of them represents one of the three components of the language.

3 Integrity Checking

Traditionally constraints are implemented by a mechanism that detects violations and then discards the modifications that caused the violations. If transactions are supported, the transaction generating the violation is aborted.

Such a strategy can be easily implemented in **Chimera**, by translating passive constraints into active rules (called triggers in **Chimera**). Triggers are constituted by three parts: event, condition, and action. If one of the events listed has occurred, the condition is evaluated. If the condition is satisfied by any object in the database, then the action part is executed.

Chimera offers various advanced characteristics that are relevant to the creation of triggers for constraint checking. *Event-formulas* permit to identify the objects that have been modified in the course of the transaction execution. The *old* meta-predicate refers to old database states: when a term is preceded by *old*, it is associated to the value that the term had either at the start of the transaction or immediately after the last execution of the rule in the same transaction; this alternative is fixed by assigning a suitable mode of evaluation to triggers.

Another feature of **Chimera** that has a particular relevance in the context of constraint management is the definition of two categories of triggers: *immediate* and *deferred*. Immediate triggers are executed at the end of the transaction command that activated them, while deferred triggers are executed when the transaction that activated them commits or issues a *savepoint* command.

In order to provide integrity checking, it is sufficient to automatically generate one abort rule for each integrity constraint, with the following syntax-directed translation.

- The event part is given by the set of database updates that can introduce violations of the constraint, which can be exhaustively identified at compile-time (see [CFPT92b] and [CW90] in the context of relational databases; the extension of such methods to **Chimera** is trivial);
- The condition part is identical to the body of the declaratively defined constraint;
- The action part consists of a single *rollback* command.

The following example shows the translation of the previous example into an abort rule.

Example 3: Rollback the transaction when the constraint is violated

```
define deferred trigger TooMuchPay
event       create(Employee), modify(Employee.Salary), modify(Employee.Manager)
condition   Employee(E), E.Salary > E.Manager.Salary
action      rollback
end
```

The advanced features of **Chimera** can be used in the following way in the context of integrity checking.

- Event-formulas can be used to restrict the set of objects on which the condition has to be tested: this generates a relevant improvement in efficiency. In the above example, the event-formula `occurred` may be used in the condition in order to restrict the evaluation to the employees that have been created or whose salary or manager has changed:

```
condition occurred
          ((create(Employee),modify(Employee.Salary),modify(Employee.Manager)),E),
          E.Salary > E.Manager.Salary
```

- The distinction between immediate and deferred triggers corresponds to a distinction between constraints. Immediate triggers are used to implement constraints representing properties of the database that should

never be violated, hence their immediate checking saves unnecessary computation. Deferred triggers represent instead constraints that can be violated in the course of a transaction and need to be satisfied only when the database is in a stable state. The constraint of Example 3 is a typical deferred one, as it needs to be checked only at transaction commit.

- The *old* meta-predicate enables the inspection of previous states in the transaction evolution; therefore it can be used to implement *transition* constraints (often called *dynamic* constraints), which forbid the occurrence of particular state sequences; we omit to describe in this paper how triggers are generated for transition constraints.

4 Integrity Enforcement

The implementation of constraints by triggers offers the opportunity to react to constraint violations in a flexible way, permitting an integrity repairing strategy, where, instead of aborting transactions, changes are applied to the incorrect database state in order to reach a consistent situation. Such a strategy is appealing because aborting transactions can be very costly. Often strategies to react to constraint violations are available and are implemented in the applications that access the database: the goal is to introduce this part of the behavior directly into the database. This extends the applicability of constraints, which can be used in contexts where the all-or-nothing behavior of traditional implementations is not adequate.

Chimera permits to pass bindings between the condition and action part of the trigger: the objects that have been identified violating a constraint can be managed in a particular way by the action part. This feature is useful when writing consistency repairing triggers. The example below shows a possible repairing trigger for the previously introduced constraint.

Example 4: If an employee earns more than his manager, make his salary equal to the manager's one

```
define deferred trigger TooMuchPay
event       create(Employee), modify(Employee.Salary), modify(Employee.Manager)
condition   occurred
            ((create(Employee),modify(Employee.Salary),modify(Employee.Manager)),E),
            E.Salary > E.Manager.Salary
action      modify(E, Emp.Salary, E.Manager.Salary)
end
```

In the above example, the `modify` primitive applies to those objects of class emp which are bound to E and assigns to the `Salary` attribute of each of them the result of the expression in the third argument, yielding the manager's salary. Note that the variable `E` is used both in the condition and in the action; indeed, the objects that satisfy the condition are bound to `E` by the condition's evaluation, and these are the objects that are updated by the *repair action*.

An approach where triggers are automatically derived from a declarative constraint specification is illustrated in [CFPT92a]. The approach considers a possible interaction with the constraint designer to guide the decision process. In that approach we distinguish two phases: first repairing rules are generated from declaratively defined constraints, then a subset of these rules is selected.

The selection process aims at maximizing the "adequacy" of constraint repair and minimizing the possibility of mutual activation between repairing rules, where one repairing rule eliminates violations of a constraint and introduces violations of a second one, and another rule eliminates violations of the second constraint and introduces violations of the first one. This situation causes cycles that may produce a nonterminating rule-system. A conservative static analysis identifies rules of this kind that are not selected.

In [CFPT92a] the constraint language is more restricted than the one of **Chimera** and the data model is simply relational. To deal with **Chimera** constraints, the generation phase needs small extensions, which are currently being formally defined.

5 Conclusions

If we reason on the evolution of database technology, we observe a progressive attempt of subtracting semantics from applications and of adding it to the generic, system-controlled components of the database. First, applications were decoupled from the physical data representation on files (physical independence); next, applications were decoupled from the actual schema structure used in the data base (logical independence). Constraint management by means of a database system can be considered as the next step in this evolution, allowing us to decouple applications from consistency maintenance. This step can be regarded as the achievement of *knowledge independence*, can be considered as part of the decoupling of application from generic, application-independent, shared knowledge; such a step will require substantial evolution of databases and of design techniques.

One of the goals of the IDEA project at the Politecnico di Milano is to design a methodology for trigger design and an environment where tools and techniques will be developed in order to help the design and analysis of sets of triggers derived from declarative specifications; constraint management represent a favorable area of application.

References

[CBB+89] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsh, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, July 1989.

[CFPT92a] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. Technical Report 92-054, Politecnico di Milano - Dipartimento di Elettronica e Informazione, 1992. To appear on *ACM Transactions on Database Systems*.

[CFPT92b] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Constraint enforcement through production rules: Putting active databases to work. *IEEE Data Engineering*, 15(1-4):10–14, December 1992.

[CM93] Stefano Ceri and Rainer Manthey. Consolidated specification of Chimera, the conceptual interface of Idea. Technical Report IDEA.DD.2P.004, ESPRIT Project n. 6333 Idea, June 1993.

[CW90] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proc. Sixteenth Int'l Conf. on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[GJ91] Narain Gehani and H. V. Jagadish. ODE as an active database: Constraints and triggers. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 327–336, Barcelona, Spain, September 1991.

[WCL91] Jennifer Widom, Roberta J. Cochrane, and Bruce G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

Integrity Control in Advanced Database Systems

Paul W.P.J. Grefen Rolf A. de By Peter M.G. Apers
Design Methodology & Database Groups
Computer Science Department, University of Twente
{grefen,deby,apers}@cs.utwente.nl

1 Introduction

Integrity control is a key feature of systems that provide access to large and shared data resources. Especially when such resources are updatable, integrity constraints have, as their prime use, the role of defining the system's criteria for update validity. For complete integrity control we identify the following tasks:

- **Integrity specification and integration** by which we mean stating declaratively what constitutes a legal, i.e. semantically allowed, database state and also what should be done if constraints are invalidated.
- **Transaction verification** which aims at proving that a database transaction cannot possibly invalidate a given constraint.
- **Integrity enforcement** which deals with the way the system checks whether constraints have been invalidated by a transaction and "repairs" integrity violations.

The first two tasks can be considered "specification-time" of an application, whereas the last task is a run-time notion of the system. Fully functional integrity management should entail all three activities, but most current systems – be they CASE-tools or DBMS's – provide little to support them.

In this paper, we give an overview of our research on integrity control in advanced database systems. We have been following two major tracks in this respect. The first is embedded in the PRISMA project on parallel, main-memory, relational databases [1]. Integrity enforcement has been an active area of research in this project, and we discuss its results in Section 2. The second track has been dealing with integrity integration and transaction verification in the context of a functional object-oriented database model, called TM [2]. This work is described in Section 3. In the Conclusions we will argue that our two tracks have been independent but orthogonal approaches to the integrity management problem that we hope to combine in the future.

2 Integrity control in a parallel DBMS

In the PRISMA project, a parallel main-memory relational database system with complete functionality has been developed [1]. One of the research tracks in the project has investigated the design and integration of an integrity control subsystem that fits the requirements of a high-performance database system. In this section, we discuss the approach taken and the results obtained in this research track.

When integrating integrity control mechanisms into a full-blown database system, a number of aspects are important. The integrity control algorithms should have clear semantics at the right level of abstraction. In particular, they should take transaction semantics into account. Also, the integrity control mechanisms should provide complete functionality, i.e. they should deal with a broad range of integrity constraint types. Further, the mechanisms should be implementable in a modular fashion with well-defined interfaces, such that integration into the DBMS architecture is relatively easy. And last but not least, the integrity control subsystem should offer good

performance, as the cost of integrity control is generally considered one of the main obstacles for its general use in practice.

2.1 Approach

To obtain both clear semantics and complete expressiveness, constraints should be specified in a high-level declarative language like first-order logic. The constraint that every employee should be employed in an existing department can thus be specified as follows, for example:

$$(\forall x)(x \in Employee \Rightarrow (\exists y)(y \in Department \wedge x.department = y.name))$$

Constraints must be translated to a high-level language with operational semantics, fit for use in constraint enforcement by the DBMS. In the PRISMA approach, an extension of the relational algebra has been chosen as constraint enforcement language. This language, called XRA, has a formal definition and clear semantics [13]. The above constraint can be expressed as follows in XRA[1]:

$$alarm(\pi_{department} Employee \Leftrightarrow \pi_{name} Department)$$

Constraints translated to their extended relational algebra form can be used to modify a transaction such that the resulting transaction is safe, i.e. cannot violate any constraints. This approach to integrity control, called *transaction modification*, is used in the PRISMA database system. Its clear semantics are supported by a declarative specification of the algorithms [11, 12], which can be easily mapped to a modular system architecture [9]. The modification of a transaction is performed by analyzing the transaction on a syntactical basis and extending it with XRA statements for integrity control purposes. This implies that the actual effect of the transaction on the database does not have to be taken into account. This gives the approach a performance and complexity advantage over other approaches, but also implies a risk of indefinite triggering behaviour in the integrity control subsystem [11, 12].

Modified transactions can be executed in the same parallel fashion as "unmodified" transactions. In the PRISMA system, this means that several forms of parallelism are available [8, 11]. Multiple transactions can be executed in parallel, each having their own processes for transaction and integrity control, and relational operation execution. Multiple layers of the DBMS can work in parallel on the same transaction, passing XRA statements through a "preprocessing pipeline". This means for example that transaction modification and transaction execution can proceed in parallel. Further, multiple XRA statements in one transaction and multiple XRA operations in one statement can be executed in parallel. And finally, as PRISMA uses fragmented relations, intra-operator parallelism can be used to spread the work of one relational operator over multiple processors.

2.2 Results

A prototype transaction modification subsystem has been integrated into the PRISMA database system, resulting in a full-fledged system with integrity control mechanisms. Experiments with this system have been performed on a POOMA shared-nothing multi-processor hardware platform. Performance measurements on this machine have resulted in two important observations that we discuss below.

Parallelism has shown to be an effective way to deal with the high processing requirements of integrity control on large databases, and transaction modification has shown to be an adequate technique to support parallelism. Detailed performance measurements are presented in [10, 11]; here we only present a typical example to give the reader an idea of the obtained results. Given is a benchmark database with a key relation of 1000 tuples and a foreign key relation of 10000 tuples, both of the well-known Wisconsin type. Enforcing a referential integrity constraint after insertion of 1000 tuples into the foreign key relation takes less than 2 seconds on an 8-node POOMA system. Enforcing a domain constraint in the same situation only takes a few tenths of a second.

[1] The *alarm* operator aborts the transaction in which it is embedded if its operand is non-empty.

The second observation is that the costs of integrity constraint enforcement in terms of execution times are quite reasonable compared to the costs of executing "bare" transactions [10]. Depending on the type of the constraint and the data fragmentation, the overhead of constraint enforcement ranges from a few percent to about the execution time of the bare transaction. This implies that integrity control is indeed usable in real-world systems with large amounts of data.

3 Integrity control in an OODB

3.1 Data model context

The TM data model is a functional, object-oriented database specification language [2] based on Cardelli-Wegner type theory. For database purposes it has been extended with logic and set expressions [4], and is currently being revised to suit the definition of technical information systems in the ESPRIT-project IMPRESS (EP 6355). It serves as the vehicle for our study of OODB-related issues, which comprises specification, optimization, and implementation. We believe that our data model is a clean, non-procedural language in the style of, for instance, the O_2 data model.

The TM data model allows the specification of classes and a special database object. For each of these, one may declare typed *attributes*, first-order logic *constraints* and *methods*. Constraints as defined in classes come in two flavours: *object* and *set*. An object constraint is a restriction on the possible values of the attributes of each object of the class in isolation[2]. A set constraint is a rule over the set of all objects of the class. This distinction is not intrinsic and serves ease of use, or in other words, each object constraint can be described as a set constraint. Constraints can also be defined on the database object, and all constraints together define a set of allowed database states, the so-called *database universe* [3].

The object-set distinction is also made for methods defined in a class: an object method takes a single object of the class as argument, a set method takes a set of such objects. This set need not be the full population of the class but can be a subset of it. A database method, furthermore, is an operation on the database object. An orthogonal distinction between methods (of any kind) is that of *retrieval* and *update*. A retrieval method simply extracts information from its arguments, whereas an update method can be used to (destructively) change its first argument. In this sense, a database retrieval method is a *query* and a database update method is a *transaction*. Needless to say, methods may invoke other methods as long as some natural laws of encapsulation are respected.

The result of a TM specification is the definition of (1) a single *database object*, (2) its allowed *values*, and (3) the allowed *operations* on it, adhering to rules of encapsulation. This means that if we want to provide an operation on a single object in the database, we have to do so via the database object level and such provision can be done automatically.

3.2 Constraints and methods

The use of the TM data model assumes a simple ACID transaction model of integrity enforcement. In other words, inconsistent transactions can be specified and will be dealt with by run-time rollbacks. But in practice, this is an undesirable situation as, at least in principle, every constraint needs to be checked. This either results in poor performance or the complete abandoning of constraint enforcement. This problem has obviously been identified in database research a long time ago, and our approach to it is both methodological and through proof checking techniques. We discuss both issues.

[2] Constraints on just a single object are also possible, but they would be phrased as object constraints that refer to some unique identification of the object.

3.3 Transaction correctness methodology

As the levels of granularity – i.e. object, set, database – of methods on the one hand and constraints on the other coincide, this is taken advantage of in the implementation of TM on real DBMS's. This is work currently being carried out in the ESPRIT project IMPRESS. The TM language is implemented on top of an object-oriented programming language that has been enriched with persistency and an interface to an object server with complex object algebra. The programming environment and object server run in a client–server architecture.

In this implementation, for each object update method an adorned version is created that checks all object constraints of the class at hand. For each set update method an adorned version is created that checks both the object and set constraints of the class. Each database update method will take into account all constraints, addressing the issue of relevance obviously: some object, set and database constraints need not be verified due to the method code. For instance, a transaction that updates objects of a single class only, needs not check object and set constraints of other classes. Note that the method adornment technique has some similarities with the transaction modification approach described in Section 2. As a tiny example, consider the object constraint '$\mathbf{self} \cdot \mathtt{age} \leq 65$' of the class `Employee` and an object method `inc` that increments the age of an employee object. The adorned version, basically saying not to change the object if the constraint would be violated afterwards, will look like:

object update method
`inc_adorned =` **if** `inc`[**self**] $\cdot$ `age` ≤ 65
then `inc`[**self**]
else self
endif

An adorned method is created such that if it calls an update method, it calls its adorned version. In this way, constraints are treated in a structural and efficient way. As much as possible, constraint checks are translated to preconditions on the actual data manipulation to increase run-time efficiency.

Checking any constraint in a method results by default in correctness or abort, but this is an over-simplified approach. Our method allows designers to define corrective actions upon expected constraint invalidation. This can be done on a per-method and per-constraint basis and may result in several adorned method versions, depending on application requirements [7].

3.4 Transaction verification

To further reduce the number of constraints to be checked by a transaction one can try to prove that the transaction can never invalidate a constraint and thus that checking it is needless.

We are currently working on proof checking techniques for compile-time transaction verification in the style of [14]. Our work is a proper extension of their work as it sets out to deal with proof checking issues on a theory in which subtyping and parametric types form an integral part. A simple first version has been implemented in the Cambridge HOL system [5], and we are currently revising that system based on lessons learnt [6].

4 Conclusions

To make integrity control techniques usable for database practice, attention should be paid both to issues of functionality and semantics, and to issues of feasibility and performance. As described above, the research at the University of Twente addresses both aspects. The work in the IMPRESS project focuses on functionality and semantics in a database system with an advanced data model. The work in the PRISMA project focuses on feasibility and performance in a database system with an advanced architecture.

In integrity control, compile-time transaction verification and run-time constraint enforcement are two complementary techniques. Combination of both techniques will provide the best basis for complete functionality

and high performance. Run-time constraint enforcement can deal with high performance requirements by the use of parallelism, such that integrity control on large databases becomes easily feasible.

References

[1] P.M.G. Apers et al.; *PRISMA/DB: A Parallel, Main-Memory Relational DBMS*; IEEE Trans. on Knowledge and Data Engineering; Vol.4, No.6, 1992.

[2] H. Balsters, R.A. de By, C.C. de Vreeze; *The TM Manual*; Technical Report INF92–81, University of Twente, 1992.

[3] H. Balsters, R.A. de By, R. Zicari; *Typed sets as a basis for object-oriented database schemas*; Proc. 7th European Conf. on Object-Oriented Programming; Kaiserslautern, Germany, 1993.

[4] H. Balsters, C.C. de Vreeze; *A semantics of object-oriented sets*; Proc. 3rd Int. Workshop on Database Programming Languages; Nafplion, Greece, 1991.

[5] R.J. Blok; *A Proof System for FM*; M.Sc. Thesis, University of Twente; October 1993.

[6] R.J. Blok, R.A. de By; *Representing FM in Isabelle*; Working document, University of Twente; March 1994.

[7] R.A. de By; *The Integration of Specification Aspects in Database Design*; Ph.D. Thesis; University of Twente, 1991.

[8] P.W.P.J. Grefen, P.M.G. Apers; *Parallel Handling of Integrity Constraints on Fragmented Relations*; Proc. Int. Symp. on Databases in Parallel and Distributed Systems; Dublin, Ireland, 1990.

[9] P.W.P.J. Grefen, P.M.G. Apers; *Integrity Constraint Enforcement through Transaction Modification*; Proc. 2nd Int. Conf. on Database and Expert Systems Applications; Berlin, Germany, 1991.

[10] P.W.P.J. Grefen, J. Flokstra, P.M.G. Apers; *Performance Evaluation of Constraint Enforcement in a Parallel Main-Memory Database System*; Proc. 3rd Int. Conf. on Database and Expert Systems Applications; Valencia, Spain, 1992.

[11] P.W.P.J. Grefen; *Integrity Control in Parallel Database Systems*; Ph.D. Thesis; University of Twente, 1992.

[12] P.W.P.J. Grefen; *Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem*; Proc. 19th Int. Conf. on Very Large Data Bases; Dublin, Ireland, 1993.

[13] P.W.P.J. Grefen, R.A. de By; *A Multi-Set Extended Relational Algebra—A Formal Approach to a Practical Issue*; Proc. 10th Int. Conf. on Data Engineering; Houston, Texas, USA, 1994.

[14] T. Sheard, D. Stemple; *Automatic Verification of Database Transaction Safety*; ACM Trans. on Database Systems, Vol.14, No.3, 1989.

Semantics and Optimization of Constraint Queries in Databases

Raghu Ramakrishnan[‡]
University of Wisconsin, Madison
raghu@cs.wisc.edu

Divesh Srivastava
AT&T Bell Laboratories
divesh@research.att.com

1 Introduction

Our research focuses on the use of constraints to represent and query information in a database. We have worked on two main issues, semantics and query optimization.

2 Optimization of Constraint Queries

Recent work (e.g., [1]) seeks to increase the expressive power of database query languages by integrating constraint paradigms with logic-based database query languages; such languages are referred to as *constraint query languages* (CQLs). In [4], we consider the problem of optimizing a CQL program-query pair $\langle P, Q\rangle$ by propagating constraints occurring in P and Q. More precisely, the problem is to find a set of constraints for each predicate in the program such that:

- Adding the corresponding set of constraints to the body of each rule defining a predicate yields a program P' such that $\langle P, Q\rangle$ is query equivalent to $\langle P', Q\rangle$ (on all input EDBs), and
- Only facts that are *constraint-relevant* to $\langle P, Q\rangle$ are computed in a bottom-up fixpoint-evaluation of $\langle P', Q\rangle$ on an input EDB.

Constraint sets that satisfy the first condition are called query-relevant predicate (QRP) constraints; those that satisfy both conditions are called minimum QRP-constraints. The notion of *constraint-relevance* is introduced to capture the information in the constraints present in P and Q. (We note that *every* fact for a predicate that appears in P or Q is constraint-relevant if neither P nor Q contains constraints.) Identifying and propagating QRP-constraints is useful in two distinct situations:

- Often, CQL queries can be evaluated without actually generating constraint facts (e.g., facts of the form $p(X, Y; X \leq Y)$). The constraints in the program are used to prune derivations, and only ground facts are generated.
- Even when constraint facts are generated, we may ensure termination in evaluating queries on CQL programs that would not have terminated if these constraints had not been propagated.

One of our main results is a procedure that computes minimum QRP-constraints (if it terminates), based on the *definition* and *uses* of program predicates. (Levy and Sagiv [2] examined this problem independently, and obtained similar results.) By propagating minimum QRP-constraints to the original program, we obtain a program that fully utilizes the constraint information present in the original program.[1] We also show that determining

[‡]The research of Raghu Ramakrishnan was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, IBM, Tandem and Xerox, and NSF grant IRI-9011563.

[1] We note that simply using Magic Templates could generate constraint facts, even when the evaluation of the original program generates only ground facts. Our algorithm avoids this.

whether (any representation for) the minimum QRP-constraint for a predicate is a finite constraint set is undecidable. We describe a class of programs for which this problem is decidable, and for which our procedure for computing minimum QRP-constraints always terminates. Further, we show that it is always better, in considering a combination of Magic Templates (for propagating constant bindings) and our techniques for computing QRP-constraints, to defer the application of Magic Templates.

3 Semantics of Constraints in Database Objects

In the previous section, when we discussed query optimization, we used the interpretation of constraint facts that is most widely studied thus far, namely the notion of a constraint fact as a finite presentation of all its ground instances. However, constraints can also be used to represent partially specified values. This is desirable when the knowledge that we would like to represent in the database is incomplete. Further, combining constraints with the notion of objects with unique identifiers offers a powerful mechanism for constraint specification and refinement. We are currently exploring these ideas ([5]), which we motivate and briefly outline below. Our technical contributions in [5] are as follows:

- We describe how an object-based data model can be enhanced with (existential) constraints to naturally represent partially specified information. We refer to this as the Constraint Object Data Model (CODM).
- We present a declarative, rule-based language that can be used to manipulate information represented in the CODM. We refer to this as the Constraint Object Query Language (COQL). COQL has a model-theoretic and an equivalent fixpoint semantics, based on the notions of constraint entailment and "proofs in all possible worlds". One of the novel features of COQL is the notion of *monotonic refinement* of partial information in object-based databases.

Both the constraint object data model and the constraint object query language are easily extended to compactly represent sets of fully specified values using universal constraints, and manipulate such values using a declarative, rule-based language, following the approach of [1, 3]. Indeed, it appears that both existential and universal constraints are instances of a more general paradigm—in essence, an underlying set of values is defined intensionally, and a number of different interpretations arise by considering various operations on this set. For reasons of space, we do not pursue this further in the paper.

3.1 Constraint Object Data Model (CODM)

In CODM, we allow facts (tuples) as well as objects, which are essentially facts with unique identifiers. The novel feature is that certain attribute values can be "don't know" nulls whose possible values are specified using constraints. We refer to attributes that can take on such values as *E-attributes*. Relations and classes are collections of facts and objects, respectively.[2]

Example 1: There are two classes of objects in the database: $playwrights$ and $plays$. Partial information is represented about the year of composition of the plays, the writers of the plays, and the year of birth of the playwrights.

playwrights			
Oid	Name	Year_of_birth	Constraints
oid1	Shakespeare	$Y1$	$Y1 \leq 1570 \ \wedge \ Y1 \geq 1560$
oid2	Fletcher	$Y2$	

[2] We note that CODM only allows *first-order* constraints, i.e., the names and types of the attributes are fixed for each fact and object, and cannot be partially specified using constraints; only the values of these attributes can be partially specified using constraints.

Note that there is no information on Fletcher's year of birth, which is equivalent to stating that Fletcher could have been born in any year.

plays				
Oid	Name	Writers	Year_of_composition	Constraints
oid11	Macbeth	{ oid1 }	$Y11$	$Y11 \leq 1608 \ \wedge \ Y11 \geq 1604$
oid12	Henry VIII	S1	$Y12$	$Y12 \leq 1613 \ \wedge \ Y12 \geq 1608 \ \wedge$ oid2 $\in S1 \ \wedge \ S1 \subseteq$ { oid1, oid2 }

The form of the constraints allowed depends on the types of the E-attributes. The Year_of_birth and the Year_of_composition E-attributes are of type integer, and hence they are constrained using arithmetic constraints over integers. Similarly, the Writers E-attribute of $plays$ is of type set of playwrights and it is constrained using set constraints $\supseteq, \subseteq, \in$. For example, the constraint on the Writers attribute of oid12 indicates that either Fletcher is the sole writer of Henry VIII, or Fletcher and Shakespeare are joint writers of that play. □

3.2 Constraint Object Query Language (COQL)

A COQL program is a collection of rules similar to Horn rules, where each rule has a body and a head. The body of a rule is a conjunction of literals and constraints, and the head of the rule can be either a positive literal or a constraint.

3.2.1 Inferring New Relationships

A COQL program can be used to infer new relationships, as facts, between existing objects and facts. For simplicity, we assume that COQL rules do not create new objects. Our results are orthogonal to proposals that permit the creation of new objects using rules, and can be combined with them in a clean fashion. We now present an example query to motivate the inference of new relationships using COQL rules.

Example 2: Consider the database of plays and playwrights from Example 1. Suppose we want to know the names of all playwrights born before the year 1700. The following rule expresses this intuition:

q1 (P.Name) : $\Leftrightarrow$ playwrights (P), P.Year_of_birth < 1700.

We use a "proofs in all possible worlds" semantics[3], wherein a playwright "satisfies" the query if: (1) *every* assignment of an integer constant to the Year_of_birth attribute of the playwright, consistent with the object constraints, satisfies the query, and (2) the derivation trees corresponding to each of the possible assignments are "similar". Only Shakespeare would be retrieved as an answer to the query under this semantics. To compute this answer set to the query, we need to check that the constraints present in the objects *entail* (i.e., imply) the (instantiated) query constraints. □

3.2.2 Monotonically Refining Objects

COQL programs can also be used to *monotonically refine* objects, in response to additional information available about the objects. For example, suppose research determined that Shakespeare could have been born no later than 1565, then the object Shakespeare can be refined by conjoining the constraint Shakespeare.Year_of_birth ≤ 1565.

[3] See [5] for a discussion of the closely related "truth in all possible worlds" semantics, and an alternative "truth in at least one possible world" semantics.

The notion of *declarative monotonic refinement* of partially specified objects is one of our main contributions in [5]. Object refinement can be formalized in terms of a lattice structure describing the possible states of an object, with a given information theoretic ordering. The value $\perp$ corresponds to having no information about the attribute values of the object, and $\top$ corresponds to having inconsistent information about the object. (There are many different complete and consistent values for the object attributes; each of these is just below $\top$ in this information lattice.) Object refinement can now be thought of as moving up this information lattice. We give an example of declarative, rule-based, object refinement next.

Example 3: The following refinement rule expresses the intuition that a playwright cannot write a play before birth:

$$\text{W.Year\_of\_birth} \leq \text{P.Year\_of\_composition} : \Leftrightarrow \text{playwrights (W), plays (P), W} \in \text{P.Writers.}$$

The right hand side (body) of the rule is the condition, and the left hand side (head) is the action of the rule. If the body is satisfied, then the instantiated head constraint is conjoined to the (global) constraints on the E-attributes.

In the presence of partial information, we give a meaning to refinement rules based on the "proofs in all possible worlds" semantics. In this case, we would conjoin the constraint $\mathsf{Fletcher.Year\_of\_birth} \leq \mathsf{Henry\ VIII.Year\_of\_composition}$ to the global collection of constraints. Conflicting refinements could, of course, result in an inconsistent constraint set. □

Rules that refine objects can be combined cleanly with rules that infer relationships between existing objects in COQL programs. For example, the rule in Example 3 can be combined with the rule in Example 2. In the resulting program, Fletcher also would be an answer to the query $q1$.

References

[1] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, Apr. 1990.

[2] A. Y. Levy and Y. Sagiv. Constraints and redundancy in Datalog. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, San Diego, CA, June 1992.

[3] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3):189–216, 1991.

[4] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. *Journal of Logic Programming*, 16(3–4):361–414, 1993.

[5] D. Srivastava, R. Ramakrishnan, and P. Z. Revesz. Constraint objects. In *Second Workshop on the Principles and Practice of Constraint Programming*, Orcas Island, WA, May 1994.

Integrating Constraint Management and Concurrency Control in Distributed Databases [§]

Gustavo Alonso Amr El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106
E-mail: {gustavo,amr}@cs.ucsb.edu

1 Introduction

Database systems often impose constraints on the values data items can take. These constraints are determined by the application semantics. The traditional approach is to assume that user transactions respect such constraints. In particular, most transaction managers are based on the assumption that each transaction, if executed by itself on a database that meets the constraints, will transfer it to another state that also meets those constraints. Serializability theory was developed to ensure that the interleaved execution of a set of concurrent transactions is correct. Note that in this traditional model the system is not aware of the database constraints. The constraints are maintained by the transactions, not by the concurrency control protocol. Recently, proposals have been made to generalize these protocols to shift the burden of constraint management from the transactions to the database system [6]. However, when distributed constraints are used, the cost of checking them across the network becomes a crucial factor in the performance and throughput of the system. The less communication involved, the less overhead the constraints impose on the system, thus, several authors [8, 7, 9, 2] have proposed the use of local constraints instead of global constraints. The local constraints ensure that when they are met, the global constraints are also met. Hence transactions can execute locally without the need for any communication.

Distributed constraints were first addressed by Carvalho and Roucairol [4]. Using their ideas, a number of protocols have been proposed to maintain distributed assertions [2, 9, 5]. However, these protocols are not integrated with the underlying concurrency control protocol. In this paper we describe a formal framework for these ideas and provide a mechanism to incorporate the semantic constraint management into traditional serializability theory. In particular, we show how constraints can be reconfigured efficiently in a distributed database system and how this reconfigurations can be combined with the concurrency control of the database. This approach not only captures the power of distributed constraints for executing transactions locally, but also its ability to support the cooperation among transactions to commit in a correct manner, i.e., without violating the distributed constraints.

The paper is organized as follows. The next section motivates the problem with an example. Section 3 presents the formal framework. Section 4 discusses how dynamic reconfiguration can take place and the consequences of these dynamic reconfigurations. In Section 5 a correctness criteria is proposed using standard serializability theory tools. Section 6 concludes the paper.

[§]This research was partially supported by the NSF under grant number IRI-9117904.

2 Motivation

When local constraints are used instead of global constraints, the local constraints are often quite restrictive since a transaction may not be executable according to a local constraint while the same transaction can execute from the global constraint point of view. This problem can be solved by dynamically adjusting the local constraints. Consider the following example. Assume a bank customer has two accounts, (A and B respectively) at two different locations. The bank requires the customer to maintain a minimum balance, N, such that $A + B \geq N$. In a conventional database, when the customer attempts to withdraw money from one account, say A, the following operation takes place:

if $A + B \Leftrightarrow \Delta \geq N$ then $A = A \Leftrightarrow \Delta$
else Abort Transaction

This requires a message exchange between both sites to learn the value of B. This, plus the locking operations involved, may cause significant delays in the execution of the transaction. Instead of using this approach, Barbara and Garcia Molina [2], for instance, propose that each site should keep a local limit, A_l and B_l, such that $A \geq A_l$, and $B \geq B_l$, and $A_l + B_l \geq N$. When a transaction attempts to withdraw Δ from A, the check that takes place is $A \Leftrightarrow \Delta \geq A_l$, which does not require communication between the sites. If, however, $A \Leftrightarrow \Delta < A_l$, it may still be possible to adjust the local limits, A'_l and B'_l, so that $A \Leftrightarrow \Delta \geq A'_l$ and $B \geq B'_l$. In what follows we generalize this approach and develop a general framework for describing how these local constraints can be maintained and dynamically modified. Furthermore, we develop a theory that permits the integration of two types of data: data where explicit constraints are used to ensure its integrity and data that assumes transactions to be correct in the sense of maintaining implicit constraints and that relies on traditional concurrency control to ensure correct execution.

3 Distributed Constraints

Consider a distributed system with three sites (the results generalize to any number of sites, we choose three for simplicity). Each one of the sites contains a data item. Let x, y, and z be those items. We will often refer to both the site and the data item residing at the site by the same name. For instance, assume a rental car company that operates at three different locations. Each location has associated with it a data item that represents the number of cars available at that location. Let $\mathcal{F}(x, y, z)$ be a distributed constraint over x, y, and z. For example, the rental car company has decided that to be able to attend unexpected requests from an important client, it will always have N cars available. The constraint is then $x + y + z > N$. Instead of enforcing $\mathcal{F}$, we define for each data item a *range*. Given the actual values of x, y and z, their *range*, $\mathcal{R}_x$, $\mathcal{R}_y$ and $\mathcal{R}_z$, are defined as the sets of values such that $x \in \mathcal{R}_x \wedge y \in \mathcal{R}_y \wedge z \in \mathcal{R}_z \Rightarrow \mathcal{F}(x, y, z)$. In the car rental example, assume $N = 10$ and that at a certain point $x = 5$, $y = 5$ and $z = 15$. Possible ranges are $\{x|x \geq 0\}, \{y|y \geq 0\}, and \{z|z \geq 10\}$. The advantage of using ranges is that, as long as each data item is within its range (which can be checked locally), the global constraint is guaranteed to hold. A *configuration* is a set of ranges. A *valid configuration* is a set of ranges that meets the distributed constraint. More formally, $\mathcal{C}$ a valid configuration with respect to a predicate $\mathcal{F}$ is as follows: $\mathcal{C} = \{\mathcal{R}_x, \mathcal{R}_y, \mathcal{R}_z \mid \forall x \in \mathcal{R}_x \wedge \forall y \in \mathcal{R}_y \wedge \forall z \in \mathcal{R}_z,\ \mathcal{F}(x, y, z)\ is\ true\}$. There may be several valid configurations for the same values of x, y and z. The set of valid configurations is called an *equivalent set*, $\mathcal{E}_{x,y,z} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3 ... \mid given\ a\ particular\ value\ for\ x,\ y,\ and\ z\ \forall \mathcal{C}_i,\ \mathcal{F}(x, y, z)\ is\ true\}$. Two configurations, with respect to a particular set of values for x, y and z, are *equivalent* if they belong to the same equivalent set. In the car rental example, with the values $x = 5$, $y = 5$ and $z = 15$, the following configurations are equivalent: $\{\{x|x \geq 5\}, \{y|y \geq 0\}, \{z|z \geq 5\}\}$ or $\{\{x|x \geq 3\}, \{y|y \geq 3\}, \{z|z \geq 2\}\}$.

An operation, $g(x)$, with result x', executed in a valid configuration $\mathcal{C} = \{\mathcal{R}_x, \mathcal{R}_y, \mathcal{R}_z\}$, is *safe* if $x' \in \mathcal{R}_x$ [2]. An operation $g(x)$ is *unsafe but solvable* if $x' \notin \mathcal{R}_x$ but $\exists \mathcal{C}'$ that is equivalent to $\mathcal{C}$ in which $g(x)$ is safe. An

operation is *unsolvable* if it is neither safe nor unsafe but solvable. Going back to the example, assume the current configuration is $\{\{x|x \geq 3\}, \{y|y \geq 3\}, \{z|z \geq 2\}\}$ and the actual values are $x = 5$, $y = 5$ and $z = 15$. If at site x clients want to rent 2 cars, they can be rented with the guarantee that the assertion still holds. In this same site, 3 cars can not be rented without violating the current configuration. However, it is possible to call any of the two other sites and ask them to modify their ranges so that x can modify its own and satisfy the current demand. Both sites y and z can safely increase their range by one (either $\{y|y \geq 4\}$ or $\{z|z \geq 3\}$) so x's range can be decreased by one ($\{x|x \geq 2\}$), which makes the operation safe. Hence to reserve 3 cars at site x is an unsafe but solvable operation. To execute an operation $g(x)$, that is unsafe but solvable in the current configuration $\mathcal{C}$, the current configuration $\mathcal{C}$ must be transformed to a new equivalent configuration $\mathcal{C}'$ where $g(x)$ is safe. There can be several such configurations. We will denote such a transformation as a *reconfiguration*, $\mathcal{R}(\mathcal{C}) = \mathcal{C}'$. Any configuration in an equivalent set can be transformed to any other configuration in that set.

4 Constraint Reconfiguration

There are several protocols that maintain semantic constraints such as those described above: the *Demarcation Protocol* [2], the *Data Value Partitioning* protocol [9], or the *Reconfiguration Protocol* [1]. The Reconfiguration Protocol generalizes the other approaches, and can be briefly summarized as follows: when a user transaction accesses one of the data items, it checks whether the operation to perform is safe or not. If it is, then it can be executed locally without communication overhead. Otherwise the protocol contacts other sites to try to find a new configuration that makes the operation safe. This is done by coordinated changes to the values of the different data items.

This reconfiguration process can be seen as a series of *reconfiguration transactions*. A reconfiguration transaction, t_{R0}, is executed at the local site when the user transaction is executed. This transaction first checks whether the operation is safe or not by comparing the final result with the corresponding range. If it is, it terminates. If it is not, it sends messages to other sites. These messages trigger remote reconfiguration transactions, t_{R1} and t_{R2}, that try to modify the ranges at those sites so the operation becomes safe. When an acknowledge message arrives at the local site, t_{R0} resumes execution modifying the local range accordingly and terminating, signaling that the operation is safe. If no acknowledge is received (after a timeout, for instance), then t_{R0} terminates signaling that the operation is unsafe and must be aborted. The design of these reconfigurations ensures that after each step the database is in a consistent state. This particular feature is important since failures may occur during the execution of the entire reconfiguration process. According to the above, the changes introduced by t_{R0} can be committed only if t_{R1} or t_{R2} commit (since t_{R0} only does those changes upon receiving a message that indicates that t_{R1} or t_{R2} have performed the appropriate changes at the remote sites). This can be formalized as follows: t_i *conditionally-commits* t_j, or $t_i << t_j$, if the abortion of t_i implies the abortion of t_j. This definition includes any dependency among transactions that may lead to cascading aborts, e.g. *reads-from* relations [3]. In the Reconfiguration Protocol, t_{R1} or t_{R2} conditionally-commit t_{R0}.

To illustrate the effects of the reconfiguration process, consider, for instance, a small shares exchange application in which shares from different companies are sold, bought and traded. For simplicity, assume there are only two types of shares, X and Y, and two trade centers, sites A and B. Each site has a certain amount of each share to trade with. X_a will represent the amount of resource X residing at site A, and so forth. Assume that the initial values are $X_a = 5$, $X_b = 5$, $Y_a = 5$, $Y_b = 5$. The constraints are $X_a + X_b \geq 0$ and $Y_a + Y_b \geq 0$ (i.e. only shares that actually exist can be sold). The initial ranges at each site are $\{X_a \geq 0 \wedge Y_a \geq 0\}$ and $\{X_b \geq 0 \wedge Y_b \geq 0\}$. There are two investors, P_a at site A and P_b at site B. When P_a and P_b perform a financial operation, the result is calculated in terms of what they had at the beginning of the operation and what they have at the end. Hence, each operation has to be performed in its entirety or not at all, i.e., each operation is executed as a transaction.

Assume P_a owns a number of shares of type X while P_b owns a number of shares of type Y. Now suppose

that P_a wants to exchange a number of X shares for Y shares and P_b wants to exchange a number of Y shares for X shares. Given the initial distribution, if both investors want to exchange 5 shares they locally execute transactions t_a and t_b. However, if they want to exchange 10 shares, the local operations are unsafe which triggers the execution of reconfiguration transactions to reset the ranges to $\{ X_a \geq 5 \wedge Y_a \geq \Leftrightarrow 5\}$ and $\{X_b \geq \Leftrightarrow 5 \wedge Y_b \geq 5\}$. t^a_{R0}, t^a_{R1} and t^b_{R0}, t^b_{R1} represent the reconfiguration transactions triggered by t_a and t_b respectively:

$$Site_a \ = \ w_a[X_a + 10] \ t^a_{R0} \ t^b_{R1} \ w_a[Y_a \Leftrightarrow 10]$$

$$Site_b \ = \ w_b[Y_b + 10] \ t^b_{R0} \ t^a_{R1} \ w_b[X_b \Leftrightarrow 10]$$

Note that at the end of this execution both transactions want to commit, since they have successfully performed the operations specified. The only difficulty is to handle this commitment so that the dependency created is captured. This dependency, or *cooperation*, between these two transaction can be formally captured by the same notion of *conditional-commitment*. In the schedule above we have the following chain of relations: $t^a_{R1} << t^a_{R0}$ and $t^a_{R0} << t_a$. The same can be said of t_b and t^b_{R0} and t^b_{R1}. Furthermore, since t^a_{R1} reads uncommitted data from t_b (the value of Y_b after t_b's increase by 10), $t_b << t^a_{R1}$. By transitivity $t_b << t_a$. Using the same type of reasoning it can be stated that $t_a << t_b$. Hence, the two transactions must now be treated as a single unit, i.e., either both commit or both abort. Traditional database theory does not support this type of cooperation. In the next section we extend database theory so that such executions can be accepted. In [1] we develop protocols for the cooperative execution of transactions.

5 Set Serializability and Constraint Management

In [1] we have developed a theory, referred to as *set serializability*, to capture the correctness of transaction execution in a system that supports implicit constraints using serializability theory and explicit constraints using reconfiguration transactions. Our approach extends the standard serializability theory by incorporating reconfiguration transactions into the theory. In particular, an execution includes both user transactions, t_i, as well as reconfiguration transactions, t_R. A reconfiguration transaction is composed of a set of local reconfiguration transactions, t_{Rj}, where each t_{Rj} is executed at a different site s_j. Each t_{Rj} accesses a range $\mathcal{R}_x$ and may also update it too. As discussed earlier, a conditional commit relationship may occur between the various local reconfiguration transactions. In particular, if the reconfiguration transaction t_{Rj} is triggered at site s_j to update the local range $\mathcal{R}_x$ as a result of a request from t^j_{R0}, then t^j_{Rj} conditionally commits t^j_{R0}.

An expanded history incorporates both the execution order specified by the user as well as the conditional commit dependencies resulting from the reconfiguration transactions. As in the traditional theory, a serialization graph can be defined on such expanded history to incorporate dependencies among transactions. However, unlike the traditional graph, not all cycles imply incorrect executions. In particular, cycles that arise due to a set of committed cooperating transactions are valid. We therefore define the notion of a *transaction set*, which corresponds to a set of cooperating transactions. A transaction set consisting of transactions with only conditional commit dependencies among them is correct, even if these dependencies form a cycle (as illustrated by the example of the previous section). We define a new type of serializability graph that distinguishes between edges created by conflicts and edges due to commit dependencies. As a result, a necessary and sufficient condition for correctness can be characterized in terms of the different cycles in the newly defined graph. If a cycle contains conflicts between transactions, then the execution is incorrect. However, transactions involved in cyclic commit dependencies form a transaction set that can be committed or aborted as a whole. Set-serializability serializes sets of transactions instead of individual transactions.

6 Conclusions

Existing work in the area of managing distributed constraints lacks a general formal framework. In this paper we describe such a framework by formalizing the main concepts involved in the maintenance of distributed constraints. Using the formalisms developed, we describe how constraint management can be integrated with concurrency control. Using serializability theory arguments, we can prove that distributed assertion protocols are correct and do so using traditional tools such as serialization graphs. This is especially interesting since most previous protocols ignore the issue of integrating distributed constraints with normal data and do not argue the correctness of the executions.

References

[1] G. Alonso and A. El Abbadi, *Partitioned data objects in distributed databases*. To appear in the Journal of Distributed and Parallel Databases.

[2] D. Barbara and H. Garcia Molina, *The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems*, 3rd International Conference on Extending Database Technology, March, 1992. pp. 373-387

[3] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[4] O.S.F. Carvalho and G. Roucairol, *On the distribution of an assertion*, ACM SIGOPS Symposium on Principles of Distributed Computing, 1982. pp. 121-131

[5] L. Golubchik and A. Thomasian, *Token allocation in distributed systems*. IEEE 12th International Conference on Distributed Computing Systems, June, 1992. pp. 64-71

[6] A. Gupta and J. Widom, *Local verification of global integrity constraints in distributed databases*. ACM SIGMOD International Conference on Management of Data, May, 1993. pp 49-58

[7] A. Kumar and M. Stonebraker, *Semantics based transaction management techniques for replicated data* ACM SIGMOD International Conference on Management of Data, May, 1988. pp. 117-125

[8] P. O'Neil, *The Escrow Transactional Method* ACM Transactions on Database Systems, 11(4), December, 1986. pp. 405-430

[9] N. Soparkar and A. Silberschatz, *Data-Value Partitioning and Virtual Messages*, Proceedings of the Conference on Principles of Database Systems (PODS), May 1990. pp. 357-367

Flexible Constraint Management for Autonomous Distributed Databases*

Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom
Computer Science Department
Stanford University
Stanford, California 94305–2140
E-mail: {chaw,hector,widom}@cs.stanford.edu

1 Introduction

When multiple databases interoperate, integrity constraints arise naturally. For example, consider a flight reservation application that accesses multiple airline databases. Airline A reserves a block of X seats from airline B. If A sells many seats from this block, it tries to increase X. For correctness, the value of X recorded in A's database must be the same as that recorded in B's database; this is a simple distributed copy constraint. However, the databases in the above example are owned by independent airlines and are therefore autonomous. Typically, the database of one airline will not participate in distributed transactions with other airlines, nor will it allow other airlines to lock its data. This renders traditional constraint management techniques unusable in this scenario. Our work addresses constraint management in such autonomous and heterogeneous environments.

In an autonomous environment that does not support locking and transactional primitives, it is not possible to make "strong" guarantees of constraint satisfaction, such as a guarantee that a constraint is always true or that transactions always read consistent data. We therefore investigate and formalize weaker notions of constraint enforcement. Using our framework it will be possible, for example, to guarantee that a constraint is satisfied provided there have been no "recent" updates to pertinent data, or to guarantee that a constraint holds from 8am to 5pm every day. Such weaker notions of constraint satisfaction require modeling time, consequently time is explicit in our framework.

Most previous work in database constraint management has focused on centralized databases (e.g., [7]), tightly-coupled homogeneous distributed databases (e.g., [6, 10]), or loosely-coupled heterogeneous databases with special constraint enforcement capabilities (e.g., [3, 9]). The multidatabase transaction approach weakens the traditional notion of correctness of schedules (e.g., [2, 5]), but this approach cannot handle a situation in which different databases support different capabilities. In its modeling of time, our work has some similarity to work in temporal databases [11] and temporal logic programming [1], although our approach is closer to the event-based specification language in RAPIDE [8].

In this paper, we give a brief overview of our formal framework for constraint management in autonomous databases and describe the constraint management toolkit we are building. Details of the underlying execution

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

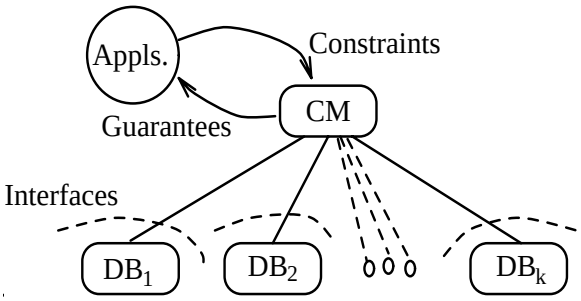


Figure 1: Constraint Management Architecture

model, the semantics of events, and the syntax and semantics of our rule-like specification language may be found in [4].

2 Formal Framework

In this section, we present an outline of our formal framework for constraint management. Our framework assumes the basic system architecture shown in Figure 1. Note that we are assuming a centralized Constraint Manager (hereafter CM) only to simplify the presentation; the CM actually is distributed. Each database chooses the *interface* it offers to the CM for each of its data items involved in an interdatabase constraint. The interface specifies how each data item may be read, written, and/or monitored by the CM. Applications inform the CM of constraints that need to be monitored or enforced. Based on the constraint and the interfaces available for the data items involved in the constraint, the CM decides on the constraint management *strategy* it executes. This strategy monitors or enforces the constraint as well as possible using the interfaces offered by the local databases. The degree to which each constraint is monitored or enforced is formally specified by the *guarantee*. We describe interfaces, strategies, and guarantees next.

2.1 Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, and/or monitored by the CM. Interfaces are specified using a notation based on *events* and *rules*. As an example, consider a simple write interface for a data item X. With this interface, the database promises to write a requested value to X within, say, 5 seconds. We express this as the rule:

$$WR(X, b)@t \rightarrow W_g(X, b)@[t, t + 5]$$

Here, $WR(X, b)@t$ represents a "write-request" event requesting operation $X := b$ at a time t. The rule says that whenever such an event occurs, a "write" event, $W_g(X, b)$[1], occurs at some time in the interval $[t, t+5]$. The interfaces for the data items involved in interdatabase constraints are specified by the database administrator of each database, based on the level of access to the database he or she is willing to offer the CM. Currently, we rely on the users of our framework to verify that the interfaces specified do faithfully represent the actual systems.

2.2 Strategies

The strategy for a constraint describes the algorithm used by the CM to monitor or enforce the constraint. Like interfaces, strategies are specified using a notation based on events and rules. In addition to performing operations

[1] $W_g()$ is a *generated* write event which occurs as the result of CM activity, to be distinguished from *spontaneous* write events, $W_s()$, which occur due to user or application activity in the underlying database.

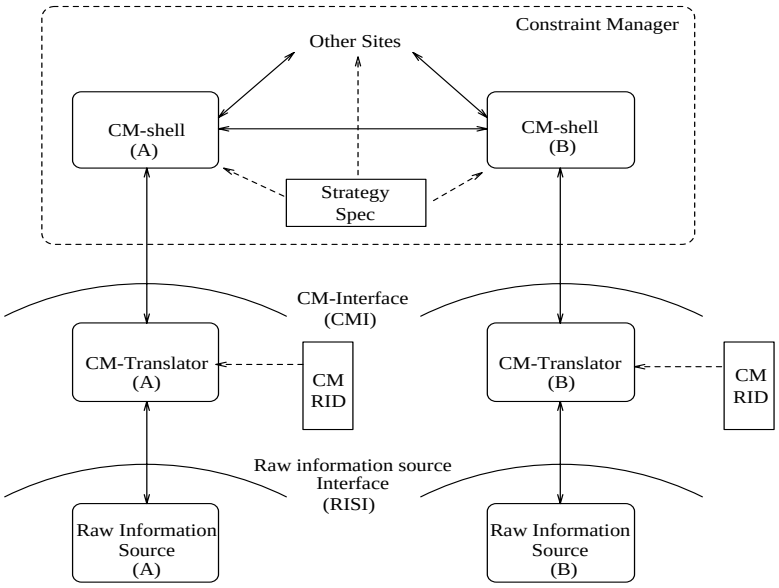


Figure 2: Constraint Management Toolkit Architecture

on data items involved in a constraint, strategies may evaluate predicates over the values of data items (obtained through database read operations) and over private data maintained by the CM. As a simple example, consider the following strategy description, which makes a write request to Y within 7 seconds whenever a "notify" event is received from X:[2]

$$N(X, b)@t \rightarrow WR(Y, b)@[t, t + 7].$$

Rule-based strategy specifications are implemented using the host language of the CM. This is typically a simple translation, and it may be achieved using a rule engine.

2.3 Guarantees

A guarantee for a constraint specifies the level of global consistency that can be ensured by the CM when a certain strategy for that constraint is implemented. Typically a guarantee is *conditional*, e.g., a guarantee might state that if no updates have recently been performed then the constraint holds, or if the value of a CM data item is true then the constraint holds. Guarantees are specified using predicates over values of data items and occurrences of certain events. For example, consider the following guarantee for a constraint $X = Y$:

$$(\text{Flag} = \text{true})@t \Rightarrow (X = Y)@@[t \Leftrightarrow \alpha, t \Leftrightarrow \beta].$$

This guarantee states that if the Boolean data item Flag (maintained by the CM) is true at time t, then $X = Y$ holds at all times during the interval $[t \Leftrightarrow \alpha, t \Leftrightarrow \beta]$. Note that this guarantee is weaker than a guarantee that $X = Y$ always holds, which is a very difficult guarantee to make in the heterogeneous, autonomous environments we are considering.

3 A Constraint Management Toolkit

We are building a toolkit that will permit constraint management across heterogeneous and autonomous information systems. This toolkit will allow us to enforce, for example, a copy constraint spanning data stored in a

[2] A notify event represents the database notifying the CM of a write to a data item. Thus, e.g., $N(X, 5)$ represents the notification that a write $X := 5$ occurred.

Sybase relational database and a file system, or an inequality constraint between a *whois*-like database and an object-oriented database. In this section we give a brief overview of our constraint management toolkit.

3.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit. The Raw Information Sources (RIS) are what exist already at each site (for example, a relational database, a file system, or a news feed). The RISI is the interface offered by each RIS to its users and applications. For example, for a Sybase database, the RISI is based on a particular dialect of SQL, and includes details on how to connect to the server.

The CM-Translator is a module that implements the CM-Interface (the interface discussed in Section 2.1) using the RISI. The CM-RID is a configuration file used to specify: (1) which CM-Interfaces (selected from a menu of predetermined interface types) are supported by the CM-Translator, and (2) how these interfaces are implemented using the underlying RISI.

The CM-Shell is the module that executes the constraint management strategies described in Section 2.2. Since we specify strategies using a rule-based language, the CM-Shells are distributed rule engines that are configured by a Strategy Specification.

3.2 Application

We now describe how database administrators would use our toolkit to set up constraint management across autonomous systems. The database administrators at each site first decide on the CM-Interfaces they are willing to offer, selected from menu of predetermined interfaces provided by the toolkit. For example, if the underlying RIS provides triggers, then a notify interface may be offered; if not, perhaps a read/write interface can be offered. The choice also depends on the actions the administrator wants to allow. For instance, even if the RIS allows database updates, the administrator may disallow a write interface that lets the CM make changes to the local data.

Each CM-RID file records the interfaces supported, as well as the specification of the RIS objects to which the interface applies. The CM-RID also stores any site-specific translation information. This includes, for example, the name of the Sybase data server that holds the data and its port number, how a read request from the CM is translated into an SQL query, how a request to set up a notify interface is translated to commands that define a trigger, and so on.

Next, the administrator uses a Strategy Design Tool (not shown in Figure 2) to develop the CM strategy. This tool takes as input the interdatabase constraints, and based on the available interfaces, suggests strategies from its available repertoire. For each suggested strategy, the design tool can give the guarantee that would be offered. The result of this process is the Strategy Specification file, that is then used by the CM-Shells at run time. Note that knowledgeable administrators might choose to write their Strategy Specifications directly, bypassing the Design Tool.

At run-time, the CM-Shells execute the specified strategy based on the event-rule formalism described above. The CM-Translators take care of translating events to site-specific operations and vice versa.

4 Conclusion

We are addressing the problem of constraint management across heterogeneous and autonomous databases and information sources. We believe that this is a problem of great practical importance, and that it is not readily solved by traditional constraint management techniques. Constraint management in autonomous environments requires weaker notions of consistency and fewer built-in assumptions than currently found in the literature. We have proposed a flexible event-based formal framework which allows us to express weaker notions of consistency, and in which time is explicitly modeled. We have also described a constraint management toolkit that we

are currently building which demonstrates the practical aspects of our work. In the future, we plan to expand our framework to handle more complex interface and constraint types, including those with quantification over data items, and we plan to incorporate probabilities into our framework for interfaces, strategies, and guarantees.

References

[1] Martin Abadi and Zohar Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295, 1989.

[2] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181, October 1992.

[3] Stefano Ceri and Jennifer Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, August 1993.

[4] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. Constraint management in loosely coupled distributed databases. Technical Report, Computer Science Department, Stanford University, 1994. Available through anonymous ftp from `db.stanford.edu: pub/chawathe/1994/cm.ps`.

[5] Ahmed Elmagarmid, editor. *Special Issue on Unconventional Transaction Management*, Data Engineering Bulletin 14(1), March 1991.

[6] Paul Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *Proceedings of the International Conference on Very Large Data Bases*, pages 581–591, Dublin, Ireland, August 1993.

[7] M. Hammer and D. McLeod. A framework for database semantic integrity. In *Proceedings of the Second International Conference on Software Engineering*, pages 498–504, San Francisco, California, October 1976.

[8] David C. Luckham et al. Specification and analysis of system architecture using RAPIDE. *IEEE Transactions on Software Engineering*, 1994.

[9] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12):46–51, December 1991.

[10] Eric Simon and Patrick Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

[11] Richard Snodgrass. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

Helping the Database Designer Maintain Integrity Constraints

Subhasish Mazumdar
Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
mazumdar@ringer.cs.utsa.edu

David Stemple
Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003
stemple@gaviao.cs.umass.edu

Abstract

The guarantee of database integrity can be extremely expensive to fulfill unless attention is paid to the problem during the design of the database. Such a guarantee can be enhanced through mechanical feedback to database designers; different kinds of feedback are obtained through analysis of database transactions in the presence of integrity constraints. In the case of distributed databases, including heterogeneous databases, the problems associated with constraint maintenance are even more acute. In our approach, the designer can reduce the non-locality of the constraints by deriving sufficient conditions from the original constraints as well as those derived after transaction analysis.

1 Introduction

Semantic integrity is a guarantee provided by a DBMS against nonmalicious errors. In practice, however, the database designer finds this to be a limited assurance because integrity checking after running each transaction can be costly. We mitigate this limitation by showing that, with the use of a theorem prover, the designer can be given rich feedback that can enhance the integrity of the resulting design.

Mechanical theorem proving is a method of symbolic manipulation: usually we want the input expression, the theorem, to be rewritten to the term true. Each rewriting step must use rules and techniques that respect logical inference; an element of heuristic-based search is usually involved at each step, and thus the time required to come to a conclusion cannot be predicted. Nevertheless, the approach is feasible at the time of the *compilation* of the database specification, when considerable computing resources can be invested at ease, with enormous potential savings during execution.

ADABTPL [9] is a high level Pascal-like language in which a database designer specifies the schema (defining the tuple types with constraints on the attributes, relation types with intra-relational constraints, and the database with inter-relational constraints). A transaction T is written as a procedure with the transaction inputs being the formal parameters. T is translated into a functional equivalent f_T. A *safety theorem* for transaction *T* is then attempted to be proven by the system [6]. This theorem
$I(d) \Rightarrow I(f_T(d))$, (*I* being the integrity constraints, and *d* any database state) states that the transaction T is *safe*, i.e., it will always preserve the integrity constraints provided the inputs meet their type restrictions [7].

It is not always easy to write specifications for safe transactions, especially if constraints involve transaction inputs. In this case, given that the safety theorem cannot be proven, the role of mechanical analysis should be to make use of the rich semantics of the integrity constraints and give the designer feedback that is useful in promoting integrity and correctness of the design [10, 11]. In the next section, we discuss forms of such feedback including run-time tests, which can be added as preconditions to the transaction, and missing updates, which can be viewed as possible corrections, The subsequent section shows that some of these techniques are applicable to

distributed databases, including heterogeneous ones. Here, feedback consists of local sufficient conditions for (both transaction-specific and transaction-independent) global constraints. The final section contains concluding remarks.

2 Transaction Analysis

Tests can be added to an unsafe transaction in order to transform it into a safe one. These tests are to be executed at run-time and their failure must cause the transaction to abort. Such tests can be generated as a side-effect of the attempt to prove the safety theorem. Let a transaction delete an element a from r_1, when the integrity constraint I is a conjunction of the following:

1. $contains(project(r_1, pid), project(r_2, pid))$, i.e., $\Pi_{pid} r_1 \supseteq \Pi_{pid} r_2$,
2. *for-all*$(r_2, t, lessp(pid(t), 1000))$, i.e., $(\forall t \in r_2) t.pid < 1000$
3. $key(r_1, pid)$, i.e., pid is the primary key of r_1.

We need to prove that the safety theorem, i.e., the conjunction of the following:

1. $I \Rightarrow contains(project(delete(a, r_1), pid), project(r_2, pid))$
2. $I \Rightarrow$ *for-all*$(r_2, t, lessp(pid(t), 1000)))$
3. $I \Rightarrow key(delete(a, r_1), pid)$

To prove an implication, the prover assumes the lefthand side (LHS) and attempts to rewrite the righthand side (RHS) to true. Formula 2 is trivial because the RHS is contained in the LHS. Formula 3 is proved directly with a rewrite rule (saying deletion does not disturb the *key*ness). The system simplifies the RHS of formula 1 using rewrite rules, but stops with the unsimplifiable expression:

$$not(member(pid(a), project(r_2, pid))), \text{ i.e., } a.pid \notin \Pi_{pid} r_2$$

which is also the necessary and sufficient test for the preservation of the integrity constraints.

Other cheaper tests may also be found — tests that are only sufficient (and not both necessary and sufficient) for the preservation of safety. Towards this end, the system attempts to *back-chain* from the unsimplifiable formula and output a sequence of sufficient conditions. If there is a lemma of the form $hypoth \Rightarrow (test = true)$, then *hypoth* can be sufficient for *test*. This process continues until there are no more such lemmas [3]. In the above example, we get the three tests

1) $project(r_2, pid) = emptyset$, 2) $r_2 = emptyset$, 3) $not(lessp(pid(a), 1000))$.

Such sufficient tests are useful in those cases where the condition tested, emptiness of relations, and a type check on the deleted data element are cheap and occur often enough to reduce the overall cost.

Adding tests may not always be the right thing to do: missing tests may simply be a symptom of another problem. The transaction writer may have forgotten updates: integrity constraints often require that updates be grouped in order to rationally update a database to model a real world event. Given an integrity constraint of the form $p(r_1, r_2)$, and a delete from r_2 (say), if we find that the expression $p(r_1, delete(a, r_2))$ is not provably true, we ask for what value of the function variable $!f$ is the following expression true?

$$p(!f(r_1), delete(a, r_2))$$

One common example of update propagation is a hierarchy in which one deletion must give rise to others. Consider the example in which $r_2 \subseteq r_1$ is part of a hierarchy. The transaction consisting solely of the deletion of an element a of r_1 gives rise to the test $not(member(a, r_2))$. We rewrite the test by replacing r_2 with a variable, and then attempt to find bindings for the variable so the test can be rewritten to true. This is a limited kind of functional unification: the undecidability of higher-order unification is not an issue here since only pattern matching is performed. We get the binding

$$r_2 \leftarrow delete(a, r_2).$$

This says that an additional update deleting a from the set r_2 would make the transaction safe.

Safety does not provide a total guarantee of correctness: transactions that are safe (or made safe) may still not be correct as per the intent of the designer. We do not believe that database designers should be called upon to make formal statements about the intent of their transactions in dynamic logic or its variants; it is impractical to expect such a high degree of mathematical maturity. Instead, we have shown that additional feedback can be provided to the designer in the form of *system-generated* postconditions [11]. These postconditions enumerate a few selected transformations of the database objects effected by the transaction. Such feedback helps the designer who finds it easier to correlate the analyzed purport of the specification against his/her knowledgeable anticipation.

Although these examples deal with very simple transactions, the method works for complex transactions and constraints [11].

3 Distributed Constraints

Distributed databases exacerbate the problem of constraint maintenance [5]. The naive method of checking constraints after each transaction is worse in the distributed case since that checking would typically require data transfer as well as computation. Further, the penalty for allowing a transaction to execute with the intention of aborting it (at commit time) in the event of violation of constraints is more severe, since rollback and recovery must occur at all the sites in which the transaction participated. In addition, owing to fragmentation and replication, there are a lot more constraints to maintain.

It was first suggested in [5] that global constraints should be *reformulated* into *local* ones and a very general theoretical framework was offered for such reformulation. The key idea here was that a sufficient condition for a global constraint may be a conjunct of local constraints.

The Demarcation Protocol [1] attacked global numeric inequality constraints, e.g., $A + B \geq 100$, where *site*(A) $\neq$ *site*(B), and broke up such constraints into local inequality constraints, e.g., $A \geq A_l$ and $B \geq B_l$, where A_l and B_l were appropriate local constant bounds. Recognizing the inequality constraints as local sufficient conditions, this can be understood as an instance of reformulation. More important, the Protocol offered algorithms for dynamic changes in these local bounds.

In our approach, we extend the mechanisms of the last section as follows. First, for a constraint C, we define a metric $\delta(C)$, which we call its *distribution* or *scatter*, to capture the amount of non-local access necessary to evaluate C. Second, we modify our back-chaining rewriting system that finds sufficient conditions so that its goal is the reduction in scatter. Given a global constraint C, it outputs $C_1, C_2, \ldots, C_m$ as sufficient, such that $\delta(C_i) < \delta(C)$ for $1 \leq i \leq m$. In the best case, $\delta(C_i) = 1$, i.e., each sufficient condition is local. Third, we take C to be a transaction-specific test that is necessary and sufficient for preservation of a constraint and derive local sufficient conditions from it. This is particularly useful in cases where the previous step does not yield purely local conditions for a particular constraint. The second and third steps are clearly modifications of one of the feedback mechanisms outlined in the previous section. Now we will briefly illustrate each of these steps.

First, let us discuss the metric. Let $\mathcal{R}$ be the set of all relation fragments. We assume that each relation fragment is located in exactly one of a set of sites $\mathcal{S}$; we define a function *site*:$\mathcal{R} \rightarrow \mathcal{S}$, where *site*($r$) denotes the location of a relation fragment r (this does not rule out replication of fragments). An integrity constraint can be a conjunction of simple constraints. A simple constraint C is given by:

$$(Q_1 x_1 \in r_1)(Q_2 x_2 \in r_2) \ldots (Q_n x_n \in r_n)\ p(x_1, x_2, \ldots, x_n),$$

where $Q_i \in \{\exists, \forall\}, 1 \leq i \leq n$, and p is a predicate. Let *site*(r_1)=s_1, *site*(r_2)=s_2, ..., *site*(r_n)=s_n, where $s_1, s_2, \ldots, s_n \in \mathcal{S}$. We define *site-set*($C$) = $\{s_1, s_2, \ldots, s_n\}$, and the distribution of C as its cardinality i.e., $\delta(C)$ = |*site-set*(C)|. The definition can be extended to accommodate free variables.

Example 1: Let $C = (\forall x_1 \in r_1)(\exists x_2 \in r_2)\ x_1.frnkey = x_2.key$, where *site*($r_1$) = s_1, *site*(r_2) = s_2, and $s_1 \neq s_2$. The constraint C has *site-set*(C) = $\{s_1, s_2\}$ and $\delta(C) = 2$. □

We define the distribution of a conjunction constraint $C = \bigwedge_{i=1}^{i=m} C_i$ as $\delta(C)$ = max($\delta(C_1)$, $\delta(C_2)$, ..., $\delta(C_m)$). If we arrive at a conjunctive constraint C with $\delta = 1$, then we can be sure that all its components C_i have unit distribution, i.e., are entirely local.

Second, let us illustrate the process of finding a reduced-scatter sufficient condition for a given transaction-independent global constraint on numeric fields of three relations:

$$(\forall x_1 \in r_1)(\exists x_2 \in r_2)(\forall x_3 \in r_3)\ x_2.fld_2 + x_3.fld_3 > x_1.fld_1$$

Let *site*(r_1) = s_1, *site*(r_2) = s_2, and *site*(r_3) = s_3, with $s_1 \neq s_2 \neq s_3$. The constraint has a scatter value of 3. Using a transitivity rewrite rule, it is possible to get the following conjunct as the sufficient condition:

$$(\exists x_2 \in r_2)(\forall x_3 \in r_3)\ x_2.fld_2 + x_3.fld_3 > k_1,$$
$$(\forall x_1 \in r_1)\ k_1 > x_1.fld_1$$

where k_1 is a constant. The net scatter is 2, but the second conjunct has unit scatter, so we can focus on the first. Rewriting it and applying the transitivity rule again, we get the following conjunct for it:

$$(\exists x_2 \in r_2)\ x_2.fld_2 \Leftrightarrow k_1 > k_2$$
$$(\forall x_3 \in r_3)\ k_2 > \Leftrightarrow x_3.fld_3,$$

where k_2 is a constant. Now we have unit scatter for the first conjunct as well as the second conjunct. Such local sufficient conditions can be found for constraints including partial orders and equivalence relations [4].

Third, we can apply this technique to a constraint C that is a test for the preservation of constraint I by a specific transaction T. In Example 1, assume that r_1 and r_2 are in different sites and a transaction inserts an element a into relation r_1. Here C_2, the test generated, is

$$a.\textit{frnkey} \in \Pi_{key} r_2,$$

but $\delta(C_2) = 2$ (a is local to s_1). The system can now find C_3 with $\delta(C_3) = 1$, a local sufficient condition:

$$a.\textit{frnkey} \in \Pi_{frnkey} r_1,$$

This approach is suitable for a distributed database because any reduction in scatter implies less communication for constraint checking. The special case in which all sufficient conditions are local is extremely important for *heterogeneous* distributed databases [2], where the requirements of site autonomy may make it impossible to abort local transactions owing to violation of a global constraint. The designer can use a tool based on our approach and ask that the sufficient local conditions be maintained by the autonomous databases, thereby maintaining the global constraint while respecting the demands of site autonomy. When local sufficient conditions are violated, the database must trigger a transaction that attempts to readjust the sufficient conditions so that global checks are once more avoided. We have recently found that the readjustment algorithms of the Demarcation Protocol can be adapted for constraint predicates such as partial orders, equivalence relations, and referential integrity (including variants with delayed checking) [8]. We are currently implementing these algorithms and finding a precise description of a class of constraints amenable to the localization strategy.

4 Conclusion

Integrity checking is a problem because of the excessive costs involved. We take the approach of providing mechanical feedback to designers in order to provide alternatives and ramifications of the existing design. In the distributed case, where the problem is harder, we use our metric on constraints to gauge non-locality, and thus provide mechanical feedback in the form of local sufficient conditions of (given and transaction-specific) global constraints.

References

[1] D. Barbará and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems. Proc. of the Third Intl. Conf. on Extending Database Technology. *EDBT 92*, pages 373–388. Vienna, Austria. March 1992.

[2] A. Gupta and J. Widom. Local Verification of Global Integrity Constraints in Distributed Databases. In *Proc. of the ACM-SIGMOD Intl. Conf. on Management of Data, Washington, D.C.*, pages 49–58, 1993.

[3] L. Henschen, W. McCune, and S. Naqvi. Compiling Constraint Checking Programs from First-Order Formulas. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Database Theory*, volume 2, pages 145–169. Plenum Press, New York, 1984.

[4] S. Mazumdar. Optimizing Distributed Integrity Constraints. In *Proc. of the Third Intl. Symp. on Database Systems for Advanced Applications (DASFAA-93). Taejon, Korea*, pages 327–334, 1993.

[5] X. Qian. Distribution Design of Integrity Constraints. In *Proc. of the Second Intl. Conf. on Expert Database Systems*, pages 205–226. 1989.

[6] T. Sheard and D. Stemple. Coping with Complexity in Automated Reasoning about Database Systems. In *Proc. of the Eleventh Intl. Conf. on Very Large Databases*, pages 426–435, 1985.

[7] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 12(3):322–368, September 1989.

[8] S. Sogani and S. Mazumdar. Extending the Scope of the Demarcation Protocol. Technical report, University of Texas at San Antonio, March 1994. UTSA-CS-94-115.

[9] D. Stemple and T. Sheard. Specification and Verification of Abstract Database Types. In *Proc. of the Third ACM Symp. on Principles of Database Systems*, pages 248–257, 1984.

[10] D. Stemple, E. Simon, S. Mazumdar, and M. Jarke. Assuring Database Integrity. *Journal of Database Administration*, 1(1):12–26, Summer 1990.

[11] D. Stemple, S. Mazumdar, and T. Sheard. On the Modes and Meaning of Feedback to Transaction Designers. In *Proc. of the ACM-SIGMOD Intl. Conf. on Management of Data, San Francisco, California*, pages 374–386, 1987.

Temporal Integrity Constraints in Relational Databases *

Jan Chomicki

Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
chomicki@cis.ksu.edu

1 Introduction

As historical databases become more widely used in practice [9], the need arises to address database integrity issues that are specific to such databases. In particular, it is necessary to generalize the standard notion of *static integrity* (involving single database states) to *temporal integrity* (involving sequences of database states). Temporal integrity constraints are used to support *time-related* policies, e.g., *"once a student drops out of the Ph.D. program, she should not be readmitted"*. In this paper, we survey our work on *temporal integrity constraints* in relational databases, done in the context of First-Order Temporal Logic (called *FOTL* hereafter). We have addressed both foundational and practical issues.

2 First-order temporal logic

Syntax. In addition to the constructs of first-order logic, *FOTL* has a number of temporal connectives: past ($\bullet$ and $\mathbf{since}$) and future ($\circ$ and $\mathbf{until}$). *Past* formulas contain only past temporal connectives, *future* formulas – only future ones. An *integrity constraint* is a closed *FOTL* formula.

Semantics. We assume that time is isomorphic to natural numbers, i.e., time instants form an infinite sequence $0, 1, 2, \ldots$ A corresponding infinite sequence $D = (D_0, D_1, D_2, \ldots)$ of *database states* with the same schema is called a *temporal database*. Every state contains a relation (set of tuples) for every relation symbol in the schema. Thus every state is just a relational database instance.

The truth value of a closed *FOTL* formula at a given time i in a temporal database D is defined in the following way:

1. an atomic formula A is true at time i if A is true in D_i.
2. if the main connective of the formula is non-temporal, then the semantics is the standard first-order semantics. For example, $\neg A$ is true at time i if A is not true at time i.
3. $\bullet A$ is true at time i iff $i > 0$ and A is true at time $i \Leftrightarrow 1$.
4. A $\mathbf{since}$ B is true at time i iff for some j, $0 \leq j < i$, B is true at time j, and for every k, $j < k \leq i$, A is true at time k.

*Research supported by NSF grant IRI-9110581.

5. $\bigcirc A$ is true at time i iff A is true at time $i+1$.

6. A **until** B is true at time i iff for some j, $j > i$, B is true at time j and for every k, $i \leq k < j$, A is true at time k.

A formula ϕ is satisfied in D (D is a *model* of ϕ) if ϕ is true in the *first* state of D. We also consider finite temporal databases, in which the satisfaction of past formulas is defined as above and future formulas are not considered. Although temporal integrity constraints are *FOTL* formulas, the above notion of formula *satisfaction* is distinct form the notion of integrity constraint *fulfillment* defined in Section 3. The temporal connectives $\blacklozenge$ (meaning *"sometime in the past"*) and $\blacksquare$ (*"always in the past"*) can be defined using **since**, while $\Diamond$ (*"sometime in the future"*) and $\Box$ (*"always in the future"*) can be defined using **until**. In temporal logic parlance we interpret constant symbols as *rigid* (having the same meaning in different states) and relation symbols as *flexible* (corresponding to different relations in different states). This interpretation seems well-suited to database applications.

We give now several examples of how *FOTL* can be used for the specification of temporal integrity constraints.

Example 1: Consider a temporal database $(D_0, \ldots, D_k)$. Each state D_{i+1} in this database results from an update applied to the state D_i. Assume that the database stores information about graduate students. A state consists of *new* information about the status of a number of students, i.e., whether they are admitted, graduate, drop out, or are reinstated. Old information is available in the previous states. The constraint *"once a student drops out of the Ph.D. program, she should not be readmitted"* can be written as:

$$\Box \neg(\exists x)(Dropout(x) \wedge \Diamond Admitted(x))$$

or as the formula C_0:

$$C_0: \ \Box \neg(\exists x)(Admitted(x) \wedge \blacklozenge Dropout(x)).$$

A looser constraint, *"if a student drops out and is not subsequently reinstated, she should not be readmitted"*, can be formulated (this time the operator **since** is essential) as the formula C_1:

$$C_1: \ \Box \neg(\exists x)(Admitted(x) \wedge (\neg Reinstated(x) \ \mathbf{since} \ Dropout(x))).$$

Finally, the constraint *"a student can not be reinstated more than twice"* is written as the formula C_2:

$$C_2: \ \Box \neg(\exists x)(Reinstated(x) \wedge \blacklozenge (Reinstated(x) \wedge \blacklozenge Reinstated(x))).$$

The last example demonstrates the convenience of using a declarative, logical language to express constraints. The additional restriction on the number of reinstatements is added in an incremental way, without any change to previously existing constraints.

Note that temporal integrity constraints relate events arbitrarily far apart in time. None of the above constraints can be directly expressed as a *transition constraint* that relates only the state after the update and the state before the update.

3 Temporal integrity

For an integrity constraint C, the *set $Pref(C)$ of prefixes of models of C* is defined as follows:

$\eta \in Pref(C)$ if there is a model $D = (D_0, \ldots, D_i, \ldots)$ of C such that $\eta = (D_0, \ldots, D_i)$.

A constraint C is *fulfilled* at time i if the current history $(D_0, \ldots, D_i) \in Pref(C)$. In other words, a constraint is fulfilled after an update if the history ending in the state resulting from the update has an (infinite) extension to a model of the constraint [7].

Unfortunately, even for restricted classes of *FOTL* formulas constraint fulfillment is in general not a computationally feasible notion. Thus, we have also proposed a weaker notion of *eventual* constraint fulfillment [3].

A set P of finite temporal databases is *a set of superprefixes* of C if it satisfies the following conditions:

1. *every constraint violation is eventually detected:* for all D, η, and i such that $D = (D_0, \ldots, D_i, \ldots)$, $\eta = (D_0, \ldots, D_i)$, $\eta \notin Pref(C)$ implies $\exists k$, $(D_0, \ldots, D_k) \notin P$ (k can be less, equal to, or greater than i).

2. *no false alarms are raised:* $Pref(C) \subseteq P$.

C is *eventually fulfilled* under a set of superprefixes P at time i if the current history $(D_0, \ldots, D_i) \in P$.

For the purposes of efficient implementation and computational complexity analysis several restricted classes of *FOTL* formulas have been identified. Reference [7] proposed the class of *biquantified formulas* (without using this specific term). Biquantified formulas allow only future temporal operators and restricted quantification in the following sense: the quantifiers can be either *external* (not in the scope of any temporal connective) or *internal* (no temporal connective in their scope). Moreover, all external quantifiers are universal. We have proposed in [1] the class of *conservative formulas* which are of the form $\Box\, \phi$ where ϕ is a past formula. The constraints C_0, C_1, and C_2 are all conservative formulas. (We should point out that only *safety* formulas [8], those whose violations can be detected by considering a finite prefix of a temporal database, make sense as integrity constraints.)

Our main theoretical results are as follows:

- for biquantified safety formulas with no internal quantifiers (called *universal*), constraint fulfillment is decidable (in exponential time)[4],

- for biquantified safety formulas with a single internal quantifier (existential or universal), constraint fulfillment is undecidable [4],

- for conservative formulas constraint fulfillment is undecidable [3] (every conservative formula is a safety formula).

As far as we know those are the only known characterizations of the computational complexity of checking temporal integrity constraints formulated in *FOTL*.

4 History-less constraint checking

For a conservative formula $\phi = \Box\, \psi$, define:

$$P_\phi = \{(D_0, \ldots, D_i) : \psi \text{ is true at } i \text{ in } (D_0, \ldots, D_i)\}.$$

The set P_ϕ is a set of superprefixes of ϕ [3] and provides the formal underpinning of our temporal integrity checking method. The membership in P_ϕ, in contrast to $Pref(\phi)$, can be efficiently checked.

The satisfaction of the past formula ψ in the current history can be checked in two distinct ways: a history-based or a history-less one. Essentially, in a history-based approach the whole current history is stored and used for checking constraints. In a history-less approach every state D_i is augmented with new *auxiliary* relations to form an *extended state* D_i'. Only the last extended state is stored and used for checking constraints. For a fixed set of constraints, the number of auxiliary relations should be fixed and their size should depend only on the set of domain values that appear in database relations. History-less constraint checking is possible at the price of

forgetting some information contained in past databases states. Only the information relevant for checking constraints is kept. Consequently, general temporal queries can not be precisely answered in this approach because some of the past information may be missing.

Below we describe a specific history-less method [1, 3], applicable to integrity constraints that are closed conservative formulas. Considering every constraint $\phi = \Box\,\psi$ in turn, the extended database schema contains, in addition to database relations, an *auxiliary* relation r_α for every temporal subformula α of ψ, i.e., for every subformula of the form $\bullet\, A$ or $A\ \mathbf{since}\ B$. For each free variable of α, there is a different attribute of r_α.

Example 2: Consider the constraint C_1. This constraint has a single temporal subformula $\alpha_1 = (\neg Reinstated(x)\ \mathbf{since}\ Dropout(x))$. Thus the extended schema contains a unary auxiliary relation symbol r_{α_1} (α_1 has one free variable) such that:

$$r_{\alpha_1}(x) \Leftrightarrow (\neg Reinstated(x)\ \mathbf{since}\ Dropout(x)).$$

This new relation has a very intuitive meaning: it contains all students that dropped out and have not been subsequently reinstated.

Consider now the constraint C_2. The extended schema contains additionally two unary auxiliary relation symbols r_{α_2} and r_{α_3} such that:

$$r_{\alpha_2}(x) \Leftrightarrow \blacklozenge\, Reinstated(x).$$
$$r_{\alpha_3}(x) \Leftrightarrow \blacklozenge\, (Reinstated(x) \wedge \blacklozenge\, Reinstated(x)).$$

Clearly, the number of auxiliary relation symbols is fixed for a fixed set of constraints. Also, the size of auxiliary relations depends only on the set of values that appear in database relations. (It is in fact polynomial in the cardinality of this set.) Thus, our method is indeed *history-less*.

An auxiliary relation r_α at time i should contain exactly those domain values that make α true at i. Thus, auxiliary relations are defined *inductively*. First, their instances at time 0 are defined, and then it is shown how to obtain the instances at time $i+1$ from those at time i. The inductive definitions are automatically obtained from the syntax of a constraint [1, 3].

Example 3: Consider the constraint C_1 from Example 1. We obtain for the first state $r^0_{\alpha_1}(x)$ which is identical to *False*, and for every $i \geq 0$:

$$r^{i+1}_{\alpha_1}(x) \Leftrightarrow r^i_{\alpha_1}(x) \wedge \neg Reinstated^{i+1}(x) \vee Dropout^i(x) \wedge \neg Reinstated^{i+1}(x).$$

The relation symbols with the superscript $i+1$ denote relations in the extended state after the update, and those with i denote relations in the extended state before the update. Moreover, the above definition *does not depend on the value of i itself*, thus can be used as a definition of a single relational view that references relations in the states before and after the update. This view should be materialized in every state, except for the first one, in the same way. In the first state, the view is initialized to an empty relation.

We have built a prototype implementation of our method [10]. The constraints are compiled into a set of STARBURST rules [11] that update the materialized views corresponding to auxiliary relations and determine whether the constraints are satisfied after an update (a transaction). The implementation is completely transparent – no changes to the rule system are necessary.

5 Related and further work

A framework similar to ours, based on the notion of a *relational automaton*, was recently proposed in [6] to overcome the limitations of the *composite events* model of [5]. Reference [6] addresses also the issue of incremental

evaluation and extends the language of past formulas with a limited form of recursion. Their ideas can be easily incorporated into our framework. To handle the notion of a *clock*, it is necessary to extend *FOTL* with *real-time* constructs. Such an extension, implementable using the approach described in Section 4, was proposed in [2].

For temporal triggers of the form "**if** *C* **then** *A*", trigger *firing* can be defined as a notion dual to constraint fulfillment [3]. The issue of controlling such firings and the operational semantics of real-time triggers require, however, further study.

Another important issue is *temporal aggregation*. For example, the constraint *"the number of submitted orders that have not been processed should not exceed K"* is expressible as a different *FOTL* formula for different values of K. It would be better to have a single formula with K as a parameter. Moreover, other aggregation operators like *sum* or *average* may also be useful. One possible solution is to design a temporal extension of SQL and try to generalize the history-less method of Section 4 to that context.

Still another important issue concerns dealing with evolving sets of constraints. It is not even clear what kind of semantics is appropriate there. For example, should past database states satisfy integrity constraints introduced later?

References

[1] J. Chomicki. History-less Checking of Dynamic Integrity Constraints. In *IEEE International Conference on Data Engineering*, Phoenix, Arizona, February 1992.

[2] J. Chomicki. Real-Time Integrity Constraints. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Diego, California, June 1992.

[3] J. Chomicki. History-less Checking of Temporal Integrity Constraints. Technical Report TR-CS-93-16, Kansas State University, February 1994. Submitted.

[4] J. Chomicki and D. Niwinski. On the Feasibility of Checking Temporal Integrity Constraints. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Washington, D.C., May 1993.

[5] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[6] H.V. Jagadish, I.S. Mumick, and O. Shmueli. Incremental Evaluation of Sequence Queries. Manuscript, February 1994.

[7] U.W. Lipeck and G. Saake. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, 12(3):255–269, 1987.

[8] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In *Current Trends in Concurrency*. Springer-Verlag, LNCS 224, 1986.

[9] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[10] D. Toman and J. Chomicki. Implementing Temporal Integrity Constraints Using an Active DBMS. In *Proc. 4th IEEE International Workshop on Research Issues in Data Engineering: Active Database Systems*, Houston, Texas, February 1994.

[11] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *International Conference on Very Large Data Bases*, 1991.

Transitional Monitoring of Dynamic Integrity Constraints

Udo W. Lipeck, Michael Gertz
Institut für Informatik, Universität Hannover, Lange Laube 22, D-30159 Hannover, Germany
{ul|mg}@informatik.uni-hannover.de

Gunter Saake
Fakultät für Informatik, TU Magdeburg, Universitätsplatz 2, D-39106 Magdeburg, Germany
saake@iti.cs.tu-magdeburg.de

1 Introduction

Database specification should include not only the static structure of a database system, but also its dynamic behaviour. In order to determine *admissible* sequences of states *dynamic integrity constraints* may be utilized. They give conditions on state transitions as well as long-term relations between database states in a descriptive manner. Complementarily, predefined *transactions,* i.e. basic elements of application programs, or ad-hoc transactions completed by *triggers* induce *executable* state sequences in an operational manner.

We have investigated how to monitor dynamic constraints without looking at entire state sequences, i.e. database histories, but considering only single state transitions, as it is usual in monitoring. Such monitoring can be achieved either by a universal (application-independent) monitor algorithm, or by (application-specific and thus more efficient) transactions or triggers into which the constraints are transformed and specialized during database design.

In our approach, dynamic integrity constraints are expressed by formulae of temporal logic (sect. 2). We have developed procedures to construct from such formulae so-called *transition graphs,* whose paths correspond to admissible state sequences (section 3). On the one hand, these graphs can control execution of a monitor that reduces analysis of state sequences to tests on state transitions. On the other hand, these graphs can be transformed systematically into refined pre-/postconditions of transactions or into triggers, such that each executable sequence becomes admissible (section 4). Such transformations can partially be supported by an environment for integrity-centered database design.

2 Dynamic Integrity Constraints

Static integrity constraints describe properties of database *states;* they restrict the set of possible states to those states which are admissible with respect to the constraints. To express such constraints, it is usual to take formulae of first-order predicate logic or related calculi.

In order to specify admissibility of state *sequences* by *dynamic* constraints, we make use of temporal logic that extends predicate logic by special operators relating states within sequences [LEG85, LS87, Lip89, Lip90] (compare also references there to other work on database specification using temporal logic). *Temporal formulae* are built from nontemporal formulae by iteratively applying logical connectives and

- a **currently** operator referring to the current state,
- a **next** operator referring to the next state,
- temporal quantifiers **always** and **sometime** referring to every or some state in the future,

- and temporal bounds **from**, **after**, **before**, and **until** which limit the range of temporal quantifiers by start or end conditions

The nontemporal subformulae may contain quantifiers ($\forall$, $\exists$) over all/some objects existing at a given state. Temporal formulae are interpreted in complete state sequences over all possible objects or data. Unlike [Cho92], we restrict the usage of object quantifications to these cases. To allow nesting of temporal operators, nontemporal subformulae are evaluated in single states, and temporal subformulae in tail subsequences.

Ordinary constraints, however, affect only objects which exist in the database, and only during their time of existence. Since objects may be inserted and deleted during database runtime, their existence intervals form – often finite – subsequences of the complete database state sequence. To deal with such *existence-restricted* constraints, formulae are interpretable in finite sequences as well, including the eventually empty tail sequence. A corresponding modification of the logic was given in [LZ91]; an alternative approach utilizing two kinds of **next**-operators can be found in [Saa91].

Example 1: The following dynamic constraint refers to an automobile registration database. It specifies that a newly produced car is not registered initially, but then it must be registered sometime in the same or next year (with its manufacturer as first owner), whereupon it remains registered:

constraint DYC_REG: **during-existence**(c: CAR):
currently (**not** c.registered **and** c.year-of-production = current-year)
and sometime c.registered **before** current-year>c.year-of-production+1
and from c.registered (**currently** c.owner=c.manuf **and always** c.registered).

3 Transition Graphs

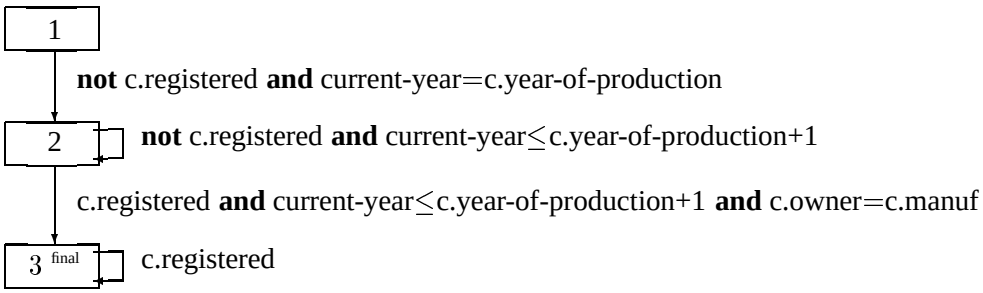
The essential means of monitoring dynamic integrity constraints on state transitions are *transition graphs*. Such a transition graph $T = \langle V, E, F, \nu, \eta, v_0 \rangle$ for a temporal formula φ consists of

- a directed graph $\langle V, E \rangle$ having a finite set V of nodes and a set $E \subseteq V \times V$ of edges,
- a set of *final nodes* $F \subseteq V$,
- a *node labelling* ν with temporal formulae,
- an *edge labelling* η with nontemporal formulae,
- and an *initial node* $v_0 \in V$ labelled with φ.

A transition graph can be used to analyze a state sequence by searching for a corresponding path whose edge labels successively hold in the states of the sequence. For that, the nodes which have been reached at each moment must be *marked*. Then a state sequence is *accepted* up to a current state iff there is a node marked after passing that state; a finite state sequence is *completely accepted* iff a final node is marked after the last state.

To guarantee that accepted state sequences correspond to admissible sequences with respect to the constraint, the nodes must be labelled by temporal formulae indicating what remains to be monitored in the future (as it is the case for the initial node). The edges, however, are labelled by nontemporal formulae which have to be checked in the next state such that the right target node gets marked. Additionally, the labels of final nodes must hold in the empty sequence. To monitor existence-restricted constraints, such a "correct" transition graph is needed for the constraint body, and it is applied to check admissibility of the existence interval for every database object.

Example 2: The following transition graph is correct for the constraint of example 1; the node labels are 1: the constraint body itself, 2: the tail formula **sometime** ..., and 3: **always** c.registered.



Obviously, the graph reflects the admissible object life-cycles with respect to the dynamic constraint. The nodes correspond to *situations* in such life-cycles, the edges give the (changing) static conditions under which transitions are allowed. In situation 2, the car c is waiting for timely registration; there is no transition for the case that the deadline is exceeded, i.e. the constraint would be violated. Only the final situation 3 allows a deletion of the object from the database.

Since in general an object may reach different situations throughout such a life-cycle, a monitor has to keep track of the marked nodes for each object (or object combination) involved in the constraint. In a deterministic graph, at most one node is marked per object. Thus the database structures must be extended to store object situations. Such markings seem to be the minimal historical information needed for transitional monitoring. Whereas the approach in [Cho92] uses object sets for all temporal subformulae of the given constraint, our approach stores object sets only for the formulae corresponding to the nodes. To avoid infinitely many object/data combinations in the presence of data variables, there are techniques for finite representation as described in [HS91].

Influenced by work on deciding satisfiability of propositional-temporal formulae like in [MW84], we have developed procedures to construct strongly correct transition graphs from temporal formulae. The algorithms in [LS87, SL87, Saa91] derive graphs by iteratively transforming formulae into a disjunctive normal form corresponding to the correctness condition above. Another algorithm [LF88, Lip89] constructs deterministic graphs by composing graphs for subformulae in a bottom-up way corresponding to the formula structure; [LZ91] gives the additions to distinguish final nodes.

In [SS92] transition graphs have been utilized for monitoring past-oriented temporal formulae, basically by reversing edges of graphs constructed for analogous future-oriented formulae. Currently we are working on transition graphs for formulae with mixed usage of past and future operators.

4 Transformation into Transactions or Triggers

In order to prepare efficient monitoring already at database design time, the processing of transition graphs (checking edge conditions, storing marked nodes) can be transformed systematically into additional pre-/postconditions of transactions such that each state sequence induced by transactions becomes admissible, too.

Example 3: For the database above, a transaction 'register' might intuitively be specified as:

transaction register (c: CAR):
pre **not** c.registered **post** c.registered.

The transition graph above, however, requires 'register' to behave as follows, since it can fit only the transition from situation 2 to 3:

transaction register (c: CAR):
pre c.situation=2 **post** c.registered **and** c.owner=c.manuf **and** c.situation=3.

To specialize transformation results for the given transactions (as already done in this example), we have developed appropriate constraint simplification techniques: Classic simplification relies on assuming that a (static)

constraint is valid before executing a transaction and on checking the same constraint after the transaction. Such techniques utilize that many constraint parts and instances remain invariant and thus need not be checked explicitly after the transaction. For dynamic constraints and transition graphs, the disjunction of ingoing edge labels can be assumed as a precondition in each situation instead. For a large class of constraints, namely those having so-called iteration-invariant transition graphs, invariance of these preconditions again leads to considerable simplifications. But even partial invariants may be erased, if the condition to be checked afterwards is different from the condition before. More application-specific simplifications are, of course, left to the designer. Rules for these transformations and simplifications have been introduced in [Lip86, Lip89, Lip90].

In the case that ad-hoc transactions (arbitrary unforeseen sequences of basic database operations like insertions and deletions) may occur, integrity maintaining triggers are needed instead.

Example 4: Situation 2 in the transition graph above gives rise to the following trigger for checking outgoing edges (written in an SQL-like style):

trigger on CAR **when updated**(registered):
  if exists (**select** $*$ **from** CAR.updated_old c_old, CAR.updated_new c_new
    where c_old.situation=2 **and** c_old.serialno=c_new.serialno
    and (current-year>c_new.year-of-production+1
      or (c_new.registered **and not** c_new.owner=c_new.manuf)))
  then rollback.

('CAR.updated_old/new' denote tables with the updated CAR tuples before/after updating.) Another trigger must care for updating the 'situation' attribute, if no rollback has occurred before:

trigger on CAR **when updated**(registered):
  update CAR c **set** c.situation=3
  where exists (**select** $*$ **from** CAR.updated_old c_old
    where c_old.situation=2 **and** c_old.serialno=c.serialno **and** c.registered)

In [GL93], we have given rules for how triggers for checking transitional conditions (like in the example) and triggers for actively updating situations can be derived at design time for every critical database operation and every situation.

5 Outlook

We consider the specification and transformation of integrity constraints into integrity maintaining mechanisms to be a central task in the database design process. Since such transformations require a lot of administration for combining, refining, and revising elements of a database schema like static and dynamic integrity constraints, transactions and triggers, a computer-aided design environment is strongly needed. We are developing a tool box called ICE ("Integrity-Centered Environment") that shall support dynamic integrity constraints in particular [GL94]. In addition to the construction of transition graphs from temporal formulae, transformations of these graphs into transactions and triggers are the main lines of the project. Tests for invariants and automatic detection of simplifications are subjects of future extensions.

Currently, we are investigating how to specify active *repairing* of constraint violations in order to extend the derivation of triggers [Ger94]. These efforts shall be generalized to dynamic constraints within the transitional monitoring approach as presented in this paper.

References

[Cho92] J. Chomicki. History-less checking of dynamic integrity constraints. In *Proc. 8th Int. Conf. on Data Engineering (ICDE'92)*, IEEE Computer Society Press, 1992, 557–564.

[Ger94] M. Gertz. Specifying reactive integrity control for active databases. In *Proc. 4th Int. Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS'94)*, IEEE Computer Society Press, 1994, 62–70. (∗)

[GL93] M. Gertz and U.W. Lipeck. Deriving integrity maintaining triggers from transition graphs. In *Proc. 9th Int. Conf. on Data Engin. (ICDE'93)*, IEEE Comp. Soc. Press, 1993, 22-29. (∗)

[GL94] M. Gertz and U.W. Lipeck. ICE: An environment for integrity-centered database design, Informatik-Bericht 02/94, Universität Hannover 1994. (∗)

[HS91] K. Hülsmann and G. Saake: Theoretical foundations of handling large substitution sets in temporal integrity monitoring. *Acta Informatica*, 28:365–407, 1991.

[LEG85] U.W. Lipeck, H.-D. Ehrich, and M. Gogolla. Specifying admissibility of dynamic database behaviour using temporal logic. In A. Sernadas, J. Bubenko, and A. Olive (eds.), *Theoretical and Formal Aspects of Information Systems (Proc. IFIP Work. Conf. TFAIS '85)*, North-Holland Publ. Co., Amsterdam 1985, 145–157.

[LF88] U.W. Lipeck and D.S. Feng. Construction of deterministic transition graphs from dynamic integrity constraints. In J. van Leeuwen (ed.), *Graph-Theoretic Concepts in Computer Science (Proc. Int. Workshop WG'88)*, LNCS 344, Springer-Verlag, Berlin 1988, 166–179.

[Lip86] U.W. Lipeck. Stepwise specification of dynamic database behaviour. In C. Zaniolo (ed.), *Proc. SIGMOD'86 Int. Conf.*, ACM, 1986, 387–397.

[Lip89] U.W. Lipeck. *Dynamische Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung* (Dynamic Database Integrity: Foundations of Specification and Monitoring, in German). Informatik-Fachberichte 209. Springer-Verlag, Berlin 1989, 140pp.

[Lip90] U.W. Lipeck. Transformation of dynamic integrity constraints into transaction specifications. *Theoretical Computer Science*, 76:115–142, 1990.

[LS87] U.W. Lipeck and G. Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, 12(3):255–269, 1987.

[LZ91] U.W. Lipeck and H. Zhou. Monitoring dynamic integrity constraints on finite state sequences and existence intervals. In J. Göers, A. Heuer, and G. Saake (eds.), *Proc. 3rd Workshop on Foundations of Models and Languages for Data and Objects*, Informatik-Bericht 91/3, TU Clausthal 1991, 115–130. (∗)

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Progr. Lang. Systems*, 6(1):78–85, 1984.

[Saa91] G. Saake. Descriptive specification of database object behaviour. *Data & Knowledge Engineering*, 6(1):47–73, 1991.

[SL87] G. Saake and U.W. Lipeck. Foundations of temporal integrity monitoring. In C. Rolland, M. Leonard, and F. Bodart (eds.), *Temporal Aspects in Information Systems (Proc. IFIP Work. Conf. TAIS'87)*, North-Holland Publ. Co., Amsterdam 1988, 235–249.

[SS92] S. Schwiderski and G. Saake. Monitoring temporal permissions using partially evaluated transition graphs. In U.W. Lipeck and B. Thalheim, editors, *Modelling Database Dynamics (Proc. Int. Workshop, Volkse 1992)*, Springer-Verlag, London 1993, 196–220.

The papers marked by (∗) are available on the ftp-server "wega.informatik.uni-hannover.de".

Integrity Maintenance in a Telecommunications Switch

Timothy Griffin
Howard Trickey
AT&T Bell Laboratories
{griffin, howard}@research.att.com

Abstract

AT&T's telecommunications switch 5ESS® uses an embedded relational database for controlling all aspects of its operation, making data integrity crucial for its proper functioning. We describe a system for maintaining data integrity that is being developed for the 5ESS.

1 Background

We have designed and implemented a system to maintain data integrity in a real-world application — AT&T's telecommunications switch 5ESS [7]. The 5ESS uses an embedded relational database for controlling all aspects of its operation. The database contains information concerning telephone customers, switch configuration, and network topology. Data integrity is crucial for the proper functioning of the switch since all application programs — performing tasks such as call processing, switch maintenance, and billing — are driven off of information stored in the database.

The 5ESS database is large and complex, containing nearly one thousand relations. Before our work, the switch developers had designed and implemented a formal language for declaring static integrity constraints — a variant of the tuple relational calculus with range restrictions. Tens of thousands of integrity assertions, some very complex, are used to generate code for a process that continually audits database integrity and reports errors as they are found.

However, it is better to avoid introducing errors in the first place. This can be done by writing transactions that will never cause violations of the integrity constraints. Such transactions are called *safe*.

The process of writing safe transactions for the 5ESS required enormous effort and was prone to error. It involved the laborious tasks of (1) writing an initial transaction in C using a low-level interface to a database manager, (2) locating and understanding the relevant integrity constraints, (3) determining which checks had to be made to ensure data integrity, (4) encoding these checks in C and inserting them into the transaction. Over a half million lines of transaction code have been handwritten in this way.

Our goal was to partially automate this process. We eased the burden of step (1) by replacing C with a high-level transaction language, and we replaced steps (2) through (4) with a tool that automatically transforms an input transaction into a safe one.

2 The Approach

Our transaction language is based on Qian's *first-order transactions* [14], which includes the basic operations of insert, delete, and update together with control constructs for sequencing, conditionals, and a restricted form of iteration.

The transformation method is based on the scheme presented by Qian in [12]. Let σ be the integrity constraint and let $T(\vec{x})$ represent a transaction with free variables $\vec{x}$ (representing parameters to be instantiated at run-time). The basic idea is to transform transaction $T(\vec{x})$ to the transaction

$$\textbf{if } \alpha(\vec{x}) \textbf{ then } T(\vec{x}) \textbf{ else } \text{ABORT}$$

where the formula $\alpha(\vec{x})$ holds before $T(\vec{x})$ is executed if and only if σ holds afterwards. Formula $\alpha(\vec{x})$ is called an *update constraint* [8]. By definition, one such formula is a *weakest precondition* [6] of $T(\vec{x})$ with respect to σ, which we denote as $\mathrm{wpc}(\sigma, T(\vec{x}))$.

It has been noted (for example [1, 5, 8, 9, 10, 11]) that the weakest precondition can be simplified by exploiting the fact that the database integrity constraint σ holds before a transaction is executed. This observation can be formalized by "factoring" the weakest precondition into the conjunction of formulas

$$\mathrm{wpc}(\sigma, T(\vec{x})) \leftrightarrow (\Phi(\vec{x}) \wedge \Delta(\vec{x})),$$

where Φ is implied by σ. If we assume σ holds, then $\Delta(\vec{x})$ is equivalent to $\mathrm{wpc}(\sigma, T(\vec{x}))$. In the best case $\Delta(\vec{x})$ is the formula TRUE, indicating that no check is needed, and in the worst case it is a formula of the same complexity as the weakest precondition. In practice the formula $\Delta(\vec{x})$ is usually much simpler than the weakest precondition.

Our starting point for generating update constraints was the algorithm presented by Qian in [12, 13]. We improved on the algorithm in several ways [3]. First, we reformulated it as a recursive algorithm that computes Δ in a one-pass bottom-up traversal of the integrity constraint σ. This makes it simpler and easier to prove correct and allows for analysis of special cases based on local information and keys. It also produces better results in that it reduces the size of the resulting update constraints. Second, we extended the algorithms to handle an aggregate term constructor, $| \{x \in R \text{ where } \alpha(x)\} |$, that counts the number of tuples t in relation R that satisfy $\alpha(t)$

Besides much syntactic sugar, the transaction language of [14] was extended with an explicit ABORT operation. This requires that the weakest precondition be reformulated since it no longer obeys the standard rule for composition

$$\mathrm{wpc}(\sigma, T_1; T_2) = \mathrm{wpc}(\mathrm{wpc}(\sigma, T_2), T_1).$$

To solve this problem we define a predicate $\mathcal{N}(T)$ that holds if and only if the transaction T will not abort, and define the weakest precondition to be

$$\mathrm{wpc}(\sigma, T) = \textbf{if } \mathcal{N}(T) \textbf{ then } \mathrm{w}(\sigma, T) \textbf{ else } \sigma$$

where $\mathrm{w}(\sigma, T)$ is a (compositional) weakest precondition for transactions that do not abort. The composition rule then becomes

$$\mathrm{wpc}(\sigma, T_1; T_2) = \textbf{if } \mathcal{N}(T_1) \wedge \mathrm{w}(\mathcal{N}(T_2), T_1) \textbf{ then } \mathrm{w}(\mathrm{w}(\sigma, T_2), T_1) \textbf{ else } \sigma.$$

This also allows us to partition the constraints into two sets; one that is enforced at the application level (here represented by σ), and the other that is enforced at a lower level by the database manager.

The approach as stated leaves a single update constraint at the top of a transformed transaction. For complex transactions the update constraint will often contain subformulas that correspond to the control of the transaction. This caused problems in our setting since transaction developers must craft informative messages that are associated with error points (ABORT). Our solution is to further transform a guarded transaction by "pushing down" update constraints into the transaction as far as possible. As a simple example, a result such as

if $(\alpha \rightarrow \Delta_1) \wedge (\neg\alpha \rightarrow \Delta_2)$
then if α
    then T_1
    else T_2
else ABORT

is transformed in one step to

if α
then if Δ_1
    then T_1
    else ABORT
else if Δ_2
    then T_2
    else ABORT

and then Δ_i may in turn be pushed into T_i. This makes the resulting transaction easier to read and provides more accurate error messages.

We have found that further simplification of formulas is crucial in obtaining good results. Given a set of formulas Γ assumed to be true and a formula α, we want to produce a formula β such that

$$\Gamma \models \alpha \leftrightarrow \beta.$$

We have implemented a simplification algorithm based on techniques of Dalal [2] to simplify both the input constraint σ and all generated update constraints. For update constraints, Γ includes σ, key constraints, $\mathcal{N}(T)$, and contextual information from the the program context into which it has been "pushed down." We have also experimented with general theorem proving techniques such as resolution, but have found these to be too expensive when dealing with large collections of constraints.

3 Results and future work

The 5ESS development organization plans to gradually replace most of the hand written C code transactions with high level transactions. In the last year over fifty transactions have been developed using our system, with good results. Developers have to write less code, the turnaround time is greatly reduced, and correctness is guaranteed (assuming the correctness of the constraints and our implementation). We conclude by mentioning a few problems we encountered.

Feedback to the developer is very important [15]. Two problems arise in our context. First, update constraints generated by machine may be difficult to understand. Second, the weakest precondition of a transaction may be inconsistent with the integrity constraints and a modification to the input is required. In both cases the developer should be given an explanation of how the results were generated.

Generation of update constraints and simplification could be improved with a better cost model. Logically equivalent formula can lead to implementations of vastly different efficiency. How can we determine which formulas are easier to check? The problem with the naive symbol-counting notion of complexity is that it is better suited to measuring the complexity of worst-case scenarios than of the computationally more relevant average case. A situation that arises frequently in the generation of update constraints serves to illustrate this point. If $\eta \rightarrow \alpha$, and η is expected to be easier to compute than α, then we might consider $\eta \vee \alpha$ to be in some sense "simpler" than α. Another example of generating such sufficient conditions can be found in the setting of distributed databases [4] where the cost model requires η to contain only references to *local* relations.

References

[1] P. Bernstein, B. Blaustein, and E. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Sixth International Conference on Very Large Data Bases*, pages 126–136, 1980.

[2] M. Dalal. Efficient propositional constraint propagation. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92)*, July 1992.

[3] T. Griffin, H. Trickey, and C. Tuckey. Update constraints for relational databases. Technical Memorandum, AT&T Bell Laboratories, 1992.

[4] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD 1993 International Conference on Management of Data*, pages 49–59, 1993.

[5] L. J. Henschen, W. W. McCune, and S. A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Database Theory*, pages 145–170. Plenum Press, 1984.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[7] W. Howard, editor. *The 5ESS Switching System*, volume 64. AT&T Technical Journal, July–August 1985. Special issue on the 5ESS switch.

[8] A. Hsu and T. Imielinski. Integrity checking for multiple updates. In *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 152–168, 1985.

[9] W. W. McCune and L. J. Henschen. Maintaining state constraints in relational databases: A proof theoretic basis. *Journal of the Association for Computing Machinery*, 36(1):46–68, January 1989.

[10] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[11] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Database Theory*, pages 170–209. Plenum Press, New York, 1984.

[12] X. Qian. An effective method for integrity constraint simplification. In *Fourth International Conference on Data Engineering*, 1988.

[13] X. Qian. *The Deductive Synthesis of Database Transactions*. PhD thesis, Stanford University, 1989.

[14] X. Qian. An axiom system for database transactions. *Information Processing Letters*, 36(4):183–189, 1990.

[15] D. Stemple, S. Mazumdar, and T. Sheard. On the modes and meaning of feedback to transaction designers. In *Proceedings of ACM-SIGMOD 1987*, pages 374–386, 1987.

Constraint Management On Distributed Design Databases *

Ashish Gupta
Department of Computer Science
Stanford University, CA 94305
agupta@cs.stanford.edu

Sanjai Tiwari
Center for Integrated Facility Engineering
Stanford University, CA 94305
tiwari@cive.stanford.edu

1 Introduction

Engineering design, such as building and aircraft design, involves many component specialties. For instance building design involves architects, structural designers, mechanical designers, electrical designers *etc.*. Each specialist has a domain-specific view of the design data, but needs to cooperate with other specialists throughout the design and construction phases of the building. The architect comes up with a plan for the building. The various designers work on the preliminary designs to satisfy the code and standards requirements. The contractor is responsible for executing the design. We refer to this collaboration between Architect, Engineers, and Contractors as the AEC framework [TH93]. Inconsistencies in design lead to expensive change orders and redesign, which in turn lead to delays and cost overruns for the project.

We propose the use of *integrity constraints* to specify design requirements and as a means of formally stating the interdependencies between various disciplines. We also describe the architecture of a distributed constraint management system (DCMS) that facilitates early detection of design inconsistencies via constraint checking. Consistency requirements may be imposed on a single site, or may involve multiple sites. That is, integrity constraints may either refer to a single database or refer to multiple databases. We refer to such constraints as *local* and *global* respectively, as illustrated by the following example.

Example 1: Consider a construction scenario and two relations from the contractor's database:

`owns`$(Crane\_id, Cost\_per\_hour)$ *% gives operating costs of the contractors cranes.*
`crane`$(Crane\_id, Floor\_id, Capacity)$ *% gives location and lifting capacity of cranes.*

Now consider a relation in the structural designer's database:

`column`$(Column\_id, Floor\_id, Weight)$ *% gives location, weight of concrete support columns.*

A *local* constraint C_l on the designer's database might require that every column on floors 4 and higher weigh less than 10 tons in order to avoid heavy structural elements on high elevations. A *global* constraint C_g might require every floor to have at least one crane that has the capacity to lift the heaviest column on that floor.

Not all the global design constraints are available at the time of database creation, instead they arise during the actual design process, *i.e.*, when the databases interact while running domain specific applications. Therefore, we need a facility for specifying and checking constraints that works independently of the underlying design data. Our proposed architecture for the Distributed Constraint Management System is designed to be built on top of existing databases. It treats constraints as first class objects, and it allows constraints to be queried and updated because constraints are often as important as the data on which they are being imposed. The layered architecture of the DCMS emphasizes autonomy of individual databases. Our architecture tries to minimize the dependence on the functionality provided by the underlying databases, because different databases may provide radically different functionality.

*Work supported by NSF grants IRI-91-16646 and IRI-911668.

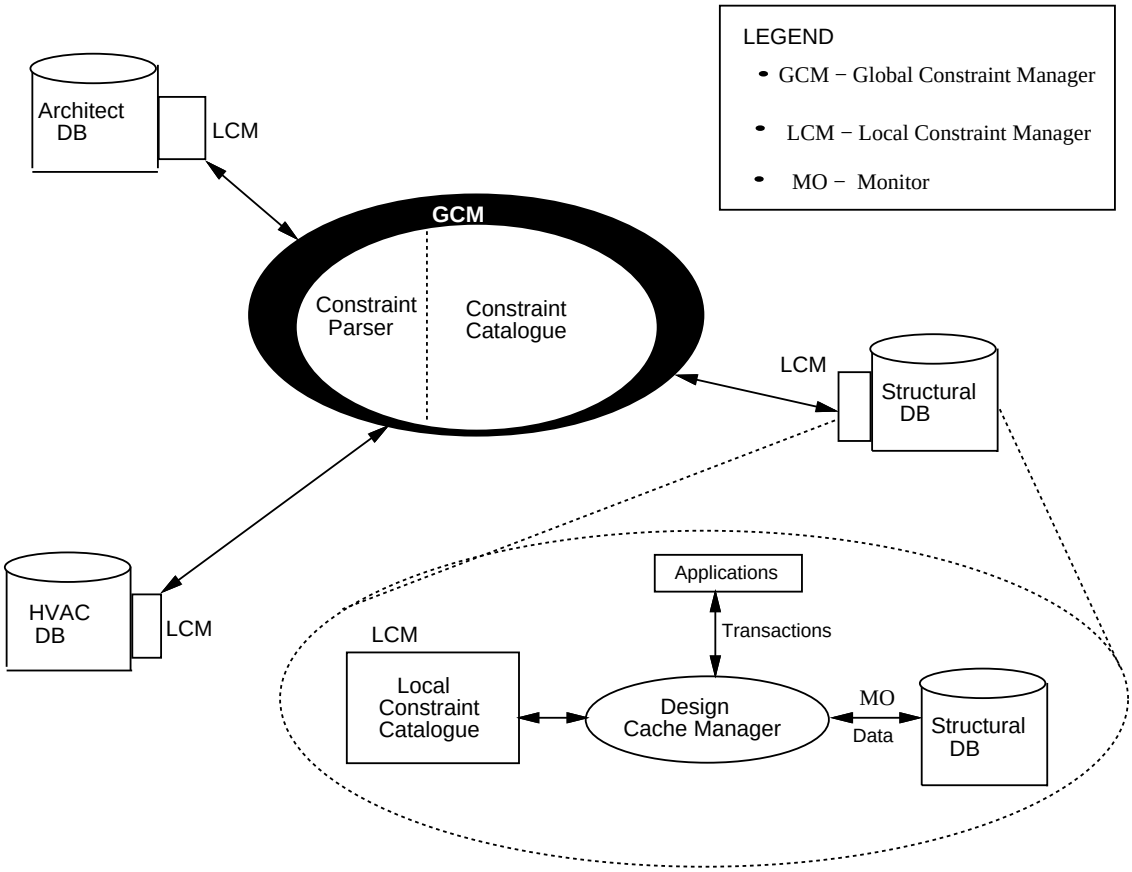


Figure 1: The Distributed Constraint Management System

Another dimension of the problem involves checking constraints when the underlying databases are updated. Checking global constraints can be very expensive due to the need to access remote data. Thus, compile time constraint optimizations are important for optimizing the run-time checking process. Our architecture is designed to incorporate many stages of optimizations. We consider *checking* constraints but not correcting their violations. Instead, we issue notifications to the participants affected by the constraint violations. A prototype of our architecture is being implemented on top of the ORACLE database system.

2 Architecture of DCMS

The proposed architecture for the *Distributed Constraint Management System* is outlined in Figure 3. Applications do domain specific reasoning and analysis and in the process alter the design database objects. The **design cache manager** at each site supports interaction of the applications with design databases. The cache manager allows checkin-checkout transactions and keeps track of the changes made to the design [KL94].

Constraints regulate the design changes made by applications. Constraints are specified, preprocessed and stored by the **Global Constraint Manager** (GCM). The GCM fragments constraints, derives optimization information, and distributes this derived information to the participating sites. At individual sites the **Local Constraint Managers** (LCM) are responsible for constraint checking. The compiled information sent to the LCMs is stored on the individual sites in **persistent catalogs**. Catalogs are also used by the GCM for global checking. Catalogs support efficient constraint checking by providing fast access to the compiled information at run-time when the GCM and LCMs use the information to check constraints. Catalogs also support query and update of the constraints. **Monitors** track changes to the objects in the design databases and detect the occurrence of operations that potentially violate constraints [CW90]. The LCM initiates the constraint checking process when prompted by a monitor.

3 Constraint Specification and Compilation

The constraint language used in our system is a variant of the language proposed in [CW90], extended to express constraints on multiple autonomous databases. Constraints are specified as inconsistent design states, *i.e.*, the condition that becomes true upon a constraint violation.

Example 2: Consider constraint C_g of Example 1. The violation condition for C_g is expressed as follows. Designer::`Columns` refers to the `Columns` relation on the Designer's site.

Designer::`Columns`.$Weight \geq$ all (Select `Cranes`.$Capacity$ From Contractor::`Cranes`
Where `Cranes`.$Floor\_Id$=`Columns`.$Floor\_Id$)
Actions: Notify(Designer, Contractor, Project Manager);

The above specification says that constraint C_g is violated if there is a column whose weight is greater than the capacity of every crane on that floor. If such columns exist, the designer, contractor, and project manager should be notified.

The constraint specification language is declarative and has many advantages over a procedural specification language. It allows a uniform high-level representation of global constraints and avoids ad-hoc user-programmed constraints that rely on the varying semantics of the underlying systems. As a consequence, the designer does not need to know the internal details of how individual sites manage integrity constraints. Uniform representation also facilitates a high-level interface to the constraint repository for constraint querying and update. Declarative specifications also facilitate extensive compile-time optimizations [Ull89].

3.1 Constraint Compilation

From the high-level specification of constraints, the compilation phase extracts the information needed for constraint checking. A constraint is syntactically parsed to form its parse tree that is used to derive the query that evaluates the constraint violation condition. The parse tree is also used to derive the set of invalidating operations, *i.e.*, the operations on the database objects that might violate a constraint [CW90]. For instance, the sample constraint C_g is compiled to obtain the following query and invalidating operations.

Example 3: Query to evaluate the constraint

Define `Bad_Set` as: (Select * From Designer::`Columns`
Where $Weight \geq$ all (Select `Crane`.$Capacity$ From Contractor::`Cranes`
Where `Cranes`.$Floor\_Id$=`Columns`.$Floor\_Id$))

Action: Notify(Designer, Contractor, Project Manager) using contents of `Bad_Set`;

`Bad_Set` contains the columns whose weight exceeds the capacity of all the cranes assigned to the corresponding floor. *Actions* specify the action to be taken on a constraint violation.

The set of invalidating operations that cause the above query to be computed:
Designer Site: Insert(Designer::`column`), update(`column`.$Weight$), update(`column`.$Floor\_Id$).
Contractor Site: Delete(Contractor::`cranes`), update(`cranes`.$Capacity$), update(`cranes`.$Floor\_Id$).

The invalidating operations are monitored at the local databases. Often a cache manager provides the monitoring facility for design databases. The set of invalidating operations can be optimized in many ways. For instance, if the floor_id of the columns is non-updatable (columns cannot be moved from floor to floor) then there is no need to monitor such an operation in Example 3.

Another optimization strategy we have devised and implemented is *Local Checking* [GSUW93]. E.g., consider constraint C_g and suppose the designer adds a new column col_1 of weight 10 tons on floor fl_1. We need to ensure that there is a crane on floor fl_1 that can lift column col_1. Suppose floor fl_1 already had a column weighing 12 tons. Assuming that C_g was satisfied before adding col_1, we can infer that floor fl_1 has a crane with

capacity of at least 12 tons and thus adding col_1 of weight 10 tons will not violate C_g. Note, the remote relation *crane* on the contractors site was not read for the above check. The compilation phase derives local tests and stores them in the LCM catalogs at various sites. Other constraint checking optimizations strategies can also be incorporated at compile time (as in [Nic82, KSS87, LST87, BMM92]).

Compilation also fragments global constraint into database specific local components that can be evaluated efficiently with fewer run-time translation overheads. For instance, some of the data and schema translations needed to check a global constraint can be compiled into the local fragments produced for the individual sites (unit conversions like inches to cms, or mapping a beam in the designer's database to a girder in the contractor's database). Fragmentation also reduces the amount of information that local sites need to communicate to the GCM for constraint checking. The constraint compilation process is described in detail in [TH93].

4 Constraint Checking and Implementation

Constraint checking is the most expensive phase of constraint management and potentially is needed every time a participating database is updated. Monitors on the database objects inform the cache manager of the updates that can potentially violate any constraint. The cache manager computes net changes made to the database and thereby avoids redundant checks; for example, a deleted object that is reinserted should not be considered as having changed. Only the relevant changes are sent to the LCM. A more detailed discussion of the change management issue can be found in [Hal91, KL94].

The LCM checks each constraint against the relevant updates by running the local tests stored in the constraint catalog at that site. If the local tests fail, then the GCM is informed about the updates. A global query – stored in the global constraint catalog – is initiated in order to check the constraint globally. Example 3 shows the global query for constraint C_g. The database updates are used to instantiate and restrict this global query as much as possible in order to make the global query efficient [Nic82, KSS87, LST87, BMM92]. For instance, if column col_1 is added to floor fl_1, then only the cranes on fl_1 need to be checked and not all cranes in the building. Distributed query optimization techniques can be used to execute the global query produced by the GCM [CP84, OV91].

4.1 Notifications

Notification is the final phase of constraint management, activated only if a constraint is violated. Often notifications need to be issued to selected participants who are responsible for resolving the design inconsistency arising from the constraint violation. For instance, if constraint C_g is violated due to adding a column, the contractor may be responsible for allocating a crane with sufficient capacity; the designer need not be notified about this constraint violation. This asymmetric participation of sites can be modeled by listing the participants in order of their responsibility in the constraint specification. The notifications could be broadcast to all the sites referred by the constraint if a notification list is not provided.

4.2 Implementation

A prototype constraint manager has been implemented in the Starburst database system at IBM-Almaden Research center and currently is being ported to ORACLE. The GCM uses *lex* and *yacc* to parse constraints and to derive compile time information like invalidating operations and local checks. Constraint catalogs are maintained as database tables and can be queried using SQL. The GCM and LCMs are implemented in C++ and C and they interact with the underlying database (and catalogs) using Starburst API and ORACLE OCI and ProC. Our architecture is general enough to be ported on top of other commercial database systems. Production rules [CW90] and triggers are used to monitor the database operations at run-time (in Starburst and ORACLE respectively). The current prototype uses a single site project database.

5 Future Work

Many interesting directions have emerged from our research. One issue is to keep track of design conflicts (constraint violations) that are detected and further track those that have been resolved or are in the process of being resolved. Thus conflicts that do not receive any attention for a given time period can be detected and reminders issued to participants. We are also considering a more general checkin-checkout mechanism for design data and a design cache-manager that would provide the LCM with a set of changes made to the design. There is also a need for allowing *what-if* changes to be made – a sort of pseudo commit process – which will be very useful for iterative design.

References

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. *LNAI 592 (subseries of LNCS)*, 114–139, 1992.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. *VLDB*, 1990.

[GSUW93] A. Gupta, S. Sagiv, J. D. Ullman, and J. Widom. Constraint Checking with Partial Information. PODS 1994.

[KL94] Karthik Krishnamurthy and Kincho Law. A Versioning and Configuration Scheme for Collaborative Engineering. In *First Congress on Computing in Civil Engineering, Washington, D.C.* ASCE, 1994.

[Hal91] K. Hall. *A Framework for Change Management in a Design Database*. PhD thesis, Stanford University, Department of Computer Science, (report number STAN-CS-91-1379), 1991.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. *VLDB*, 1987.

[LST87] J.W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[Nic82] J. M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[OV91] T. M. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[SAL92] D. Sriram, S. Ahmed, and R. Logcher. A Transaction Management Framework for Collaborative Engineering. *Engineering with Computers*, 8(4), 1992.

[TH93] S. Tiwari and H. C. Howard. Distributed AEC Databases for Collaborative Design. *The Journal of Engineering with Computers*", Springer-Verlag, 1994.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems 1 and 2*. CSP, 1989.

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

CALL FOR PAPERS

Research Surveys and Correspondences on Recent Developments

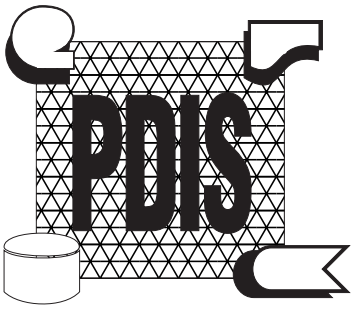
We are interested to publish in the *IEEE Transactions on Knowledge and Data Engineering* research surveys and correspondences on recent developments. These two types of articles are found to have greater influence in the work of the majority of our readers.

Research surveys are articles that present new taxonomies, research issues, and current directions on a specific topic in the knowledge and data engineering areas. Each article should have an extensive bibliography that is useful for experts working in the area and should not be tutorial in nature. Correspondences on recent developments are articles that describe recent results, prototypes, and new developments.

Submissions will be reviewed using the same standard as other regular submissions. Since these articles have greater appeal to our readers, *we will publish these articles in the next available issue once they are accepted*.

Address to send articles:	Benjamin W. Wah, Editor-in-Chief Coordinated Science Laboratory University of Illinois, Urbana-Champaign 1308 West Main Street Urbana, IL 61801, USA Phone: (217) 333-3516 (office), 244-7175 (sec./fax) E-mail: b-wah@uiuc.edu
Submission Deadline:	None
Reviewing Delay:	One month for correspondences, three months for surveys
Publication Delay:	None; articles are published as soon as they are accepted
Submission Guidelines:	See the inside back cover of any issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu (128.174.197.211) in file /pub/tkde/submission.guide.ascii
Length Requirements:	40 double-spaced pages for surveys, 6 double-spaced pages for correspondences
Areas of Interest:	See the editorial in the February'94 issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu in file /pub/tkde/areas.of.interest

CALL FOR PARTICIPATION



Third International Conference on

PARALLEL AND DISTRIBUTED INFORMATION SYSTEMS

September 28-30, 1994 • Four Seasons Hotel • Austin, TX

SPONSORS

IEEE Computer Society TC on Distributed Processing, ACM SIGMOD
Grants provided by Bellcore, US WEST

COMMITTEES (PARTIAL LIST)

General Chair: Susan Davidson, Univ. of Pennsylvania
Steering Committee:
Sushil Jajodia (Chair), George Mason University
Hector Garcia-Molina, Stanford University
Masura Kitsuregawa, University of Tokyo
Sham Navathe, Georgia Tech.
Napthali Rishe, Florida International University
Program Chairs: Hank Korth, Panasonic Technologies, Inc., Amit Sheth, Bellcore
Publicity Chair: Yelena Yesha, U. of Maryland Baltimore County and NIST
For more information: yeyesha@cs.umbc.edu

SCOPE

The scope of this conference covers parallel and distributed systems for databases, large-scale knowledge bases and information management.

CONFERENCE HIGHLIGHTS

The conference includes 22 papers and several synopses. Topics include: Parallel I/O and File Systems, Query Processing and Joins, Mobile Computing and Information Exchange, Parallel Processing and Replicated Data, Transaction Processing and Active Databases, Federated Databases, and Client Server Architectures and Cache Management.

It also includes keynote addresses and 2 panels entitled : "Legacy Systems: the Achilles Heel of Downsizing" chaired by Dr. Michael Stonebraker from Berkeley University, and "Charting the Web: Pro-active Information Gathering Techniques" chaired by Dr. Michael F. Schwartz from the University of Colorado.

Tutorials are scheduled for Wednesday, September 28. The first tutorial, entitled Parallel Database Systems — The First Generation, is given by Dr. Chaitanya K. Baru from IBM Toronto Labs. The second tutorial is on Data Management Issues in Mobile Computing, and is presented by Dr. Tomasz Imielinski from Rutgers University.

One of the highlights of this year's conference is its Industrial Program. Topics to be addressed include: Data Management in Distributed Computing Environments, Parallel Computers in Developing Countries, and Information Systems on the Information Highway.

PDIS94 REGISTRATION FORM

Mail to: Vijay Garg, PDIS94 Registration, Electrical & Computer Eng. Dept., University of Texas at Austin, Austin, TX 78712;
Phone: (512) 471-9424, **Fax:** (512) 471-5907; **Email Registration:** pdis94@pine.ece.utexas.edu.

Name _____

Affiliation _____

Street _____

City/State/Zip/Country _____

Phone _____ Fax _____

Email _____ IEEE CS/ACM Membership # _____

Do you have any special needs? _____

Wednesday, Sept. 28: Tutorials, luncheon, reception.
Thursday & Friday, Sept. 29-30: Conference program, luncheon.
Tutorial Registration includes: All events for Sept. 28.
Conference Registration includes: Reception, proceedings and all events for Sept. 29 & 30.

TUTORIAL FEES (please circle appropriate fee)

	Before Sept. 1	After Sept. 1
IEEE-CS/ACM Members	$160/1 tutorial $270/both	$195/1 tutorial $340/both
Non-Members	$200/1 tutorial $340/both	$240/1 tutorial $420/both
Full-time Students	$50/1 or both	$70/1 or both

I will attend (please check)
_ Tutorial 1 (Parallel Database Systems)
_ Tutorial 2 (Mobile Computing)

CONFERENCE FEES (please circle appropriate fee)

	Before Sept.1	After Sept.1
IEEE-CS/ACM Members	$285	$345
Non-Members	$360	$440
Full-time Students	$80	$90

AMOUNT ENCLOSED
Tutorial Fees: $_____ + Conference Fee: $_____ =
Total Due: $_____

Payment can be made by check, money order, or credit card. Please make checks or money orders payable, in US currency, to PDIS94. Use credit card to register by fax or email.
Credit Card: __ Visa __ MasterCard

Card Number _____

Exp. Date _____

Cardholder Name _____

Signature _____

To receive student rate, students must have advisor's name and signature at the time of registration. Student registration does not include lunches.

Advisor name _____

Signature _____

Written requests for refunds must be postmarked no later than 9/1/94. Refunds are subject to a $50 processing fee. All no-show registrations will be billed in full. Registrations after 9/1/94 will be accepted on-site only. Receipts will be given out at the conference.

ABOUT AUSTIN...

The conference will be held at the The Four Seasons Hotel, situated on scenic Town Lake in downtown Austin. The Four Seasons enjoys a naturally restful setting while being close to evening entertainment. Austin contains many city sights including the State Capitol, the University of Texas at Austin, the L.B.J. Library, L.B.J. National Wildflower Research Center, Texas Memorial Museum and the Governor's Mansion. Austin is a high-tech city with unique consortia such as MCC and Sematech, and over 400 software companies and a notable presence of companies such as AMD, Dell, IBM, Motorola, Schlumberger, Tandem and 3M. Daytime temperatures during the last week of September are typically low to mid 80's. The airport is located 10 minutes away via I-35 or by taxi.

HOTEL RESERVATIONS

Four Seasons Hotel, 98 San Jacinto Blvd.,
Austin, TX 78701 USA; **Reservations:** (512) 478-4500

DEADLINE: September 6, 1994 (5PM Central Time)
Single $110, Double $120

Mention IEEE CS to receive the discounted rate. All rates subject to 13% occupancy tax. Reservations received after the deadline or after the reserved block of rooms is filled will be confirmed on a space available basis. All reservations will be held until 6pm. Arrivals after 6pm must be covered by first night's deposit on your Amex, VISA, Mastercard, Diners or Carte Blanche card. If a reservation is held on a guaranteed basis (for arrival after 6pm) the individual will be held responsible for payment of first night only. Guaranteed reservations are held (without occupancy) for one night only and not for entire length of stay requested.

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398