Bulletin of the Technical Committee on

Data Engineering

December, 1992 Vol. 15 No. 1 - 4

IEEE Computer Society

SPECIAL ISSUE ON ACTIVE DATABASES

Data Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Data Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed software design and implementation, database utilities, database security machines, intelligent front ends, mass storage for very large databases, distributed software design and implementation, database utilities, database security and related areas.

Contributions to the Bulletin are hereby solicited. News conference calls, and letters, etc. should be sent to the Editor-in-Chief. Letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers to appear in special issue should be sent directly to the issue editor. Technical papers are not refereed

Opinions expressed in contributions are those of the idividual authors rather than the official position of the TC on Data Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Data Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. Any member of the IEEE Computer Society may join the TC.

Letter from the TC Chair

It is a great pleasure to see the Data Engineering Technical Bulletin resurrected after a lapse of more than a year. David Lomet, the new Chief Editor of the Bulletin, deserves much of the credit for making it happen. It is due to his foresight, his perseverance, and his organizational skills that we are on-line again. I believe that the Data Engineering Bulletin has provided a unique service to the database community, and I am hopeful that this tradition will continue.

I want to take this opportunity to apprise you of what caused an interruption in the publication of the Bulletin, and to bring you up to date on the current state of TCDE. My understanding is that the Bulletin was earlier funded by a grant from the IEEE Computer Society Technical Activity Board. Sometime last year, it was decided that all the technical committees should become self-sufficient. In particular, the Bulletin needed to be financed using the revenues generated from the TCDE sponsored activities. The then TCDE Chair also resigned in June. The result was that TCDE was left with no budget and no organization.

In March this year, I agreed to become the TC Chair for a year. I am pleased to inform that TCDE now has a strong Executive Committee, consisting of Nick Cercone (Vice Chair), Amit Sheth (Secretary/Treasurer), Benjamin Wah (Conferences Co-ordinator), David Lomet (Publications Co-ordinator), Erich Neuhold (European Co-ordinator), Shojiro Nishio (Asian Co-ordinator), and Ron Sacks-Davis (Australian Co-ordinator). I feel having a diverse Executive Committee will provide the much-needed continuity to TCDE.

We were able to get a one-time grant of $4000 for 1992 from the Technical Activity Board for restarting the Bulletin. We also now have a budget of $8000 for 1993 for the Bulletin. However, in the long run, the Bulletin cannot depend on the vagaries of the TCDE budget. We have spent considerable time exploring the long term financial viability of the the Bulletin without compromising its quality and unique role. Given the high cost of paper publication and distribution, it is unlikely that we can afford to distribute the paper version for free. We don't have all the details in place, but we are moving towards a model where the Bulletin is published in a combination of electronic and paper forms.

I want to close by applauding Won Kim for his services to the Data Engineering Bulletin. He devised the current format for the Bulletin, gave it a novel personality, and steered it for more than 10 years. Thanks, Won.

Please e-mail me or any member of the Executive Committee if you have any suggestions or if you would like to volunteer your time for TCDE.

Rakesh Agrawal
Chair, TC on Data Engineering

Re-Introducing The Data Engineering Bulletin

I have long believed that the Bulletin provides a unique and valuable service. It is unique in that each issue is devoted to a different special topic. Leading researchers on the special topic provide papers describing their work and their assessments of the field in a format that would not be appropriate for conferences or journals. But the articles are highly valuable for TC members wishing to understand what is happening in a field. It is this unique role, established by Won Kim, that was the inducement for me to accept TC Chair Rakesh Agrawal's invitation to become editor-in-chief of the Bulletin.

This current issue continues the Bulletin's role by containing a special issue on Active Databases. Active databases is not only an area of current research interest, but commercial vendors of database systems are providing "active" functionality to their users, e.g., via triggers and constraints. As you can see from the table of contents, the issue contains contributions from many of the top database researchers. I would like to thank Professor Sharma Chakravarthy of the University of Florida, who acted as guest editor for this issue, and Professor Ahmed Elmagarmid who, as associate editor, arranged Sharma's role and helped with the editing.

I want to continue having the Bulletin provide focused issues on areas of interest to the database community. In addition to its traditional role of covering special areas of research interest, I would like to include from time to time issues on the state of industrial practice in such areas. My experience suggests that knowledge within our technical community of the functionality and the directions of commercial database offerings is spotty at best. I think this would be a real eye-opener in some fields as practice sometimes is in advance of research.

The editorial board for the Bulletin is in transition. Won Kim, the Bulletin's former editor, established the Bulletin and sustained it for many years. He also started the practice of appointing associate editors for two year terms. His last appointed associate editors, Ahmed Elmagarmid, who played a role in putting together the current issue, Yannis Ioannidis, who is acting as issue editor for the March issue, Kyu-Young Whang, and Rakesh Agrawal, who is now the TC chair, have all served more than two years. I want to thank them all for their valued contributions to the bulletin during their terms.

I will soon appoint a new editorial board. I am pleased to announce that my first appointment is Meichun Hsu, formerly a faculty member at Harvard and currently on the staff at Digital's Lab in Mt. View, California. Mei has a distinguished publication record, and brings to the editorship some of the industrial focus that I look forward to seeing in the Bulletin. I am sure that the Bulletin will be well served by her efforts.

Let me close by directing your attention to the announcement on page 3. It contains a request for you to provide information so that we can continue to bring you the bulletin in a timely and cost effective way. Your continued receipt of the bulletin requires your response. I hope to hear from you soon.

David Lomet
Editor-in-Chief

Important Membership Announcement

The IEEE Computer Society Technical Committee on Data Engineering needs to rebuild its membership list. Our current lists are now quite old, we are uncertain whether the address information remains current, and because we plan to exploit e-mail much more in the future, we need e-mail addresses as well as postal addresses. Electronic mail is a low cost way for the TC to reach its members.

Particularly important for the future of the Data Engineering Bulletin, we are planning the electronic distribution of the Bulletin. As most of you are aware, this is the first issue of the Data Engineering Bulletin that has been published since September, 1991. The Bulletin was shut down for over a year very simply because of insufficient funding. An effort was made in 1990 to establish a membership fee for the Technical Committee, but this did not work out well.

Our goal is to continue to bring you the interesting special issues of the Bulletin at a cost that ensures its long term survival. Our plan for continued and low-cost publication of the Bulletin has two parts:

1. All TC members will receive announcements of each issue of the Bulletin as it is published. They will be able to obtain electronically, the postscript version, and perhaps a latex version of the issue.

2. Hardcopy of each issue will also be available, but only to subscribers, and at a cost that helps to cover the TC's printing and distribution costs. The annual subscription fee for four issues is expected to be in the $10 to $15 range.

To proceed with these plans, we request that you re-enroll as a TC member using the following procedure:

1. Send e-mail to TCData@crl.dec.com and include in the subject header the word "ENROLL".

2. You will then be sent via an e-mail reply, an electronic membership form that will request:

   Name, IEEE membership no., postal address, e-mail address

   In addition to the above information, you will be asked a number of questions on issues that will affect how the Bulletin's distribution will be handled. These questions will include whether you are interested in subscribing to the hardcopy version of the bulletin. This information will enable us to plan our print runs and determine the parameters of electronic distribution.

3. You should then e-mail the electronic application form to TCData@crl.dec.com, with the word "APPLICATION" in the subject header.

This procedure permits us to electronically establish our member list without any manual transcription process. Please be aware that no response will mean that you will not remain a TC member, and hence that you will not receive the Bulletin.

David Lomet
Editor-in-Chief

Letter from the Guest Issue Editor

Research on active databases has been prompted by a genuine need for supporting database functionality deemed important for a number of non-traditional applications, such as Computer Integrated Manufacturing (CIM), stock trading, and network management. Although the concept of condition monitoring itself is not new, its formulation in the context of active databases has received substantial attention in the last five years – both from researchers and developers. Already, there are a number of research prototypes, and commercial products with primitive active database features.

Concepts that have emerged from the active database research have provided a uniform framework for supporting a number of functionality that were being supported in an *ad hoc* manner. Although HiPAC, Postgres, and the work at Karlsruhe pioneered the work on active databases, a number of groups are currently working on a wide range of issues, as can be seen from the coverage of topics in this issue. If the number of papers on active databases submitted to conferences in the last two years is any indication of ongoing research, undoubtedly there is an enormous following for this area.

For a change, it is not just the academician who is interested in this area of research. Assessing from the number of commercial database management systems supporting active capability (albeit a primitive one), it is clear that the technology is here to stay and promises a faster transition from research results to products. Triggers will certainly make their way into the SQL3 standard.

This issue intends to provide a fish-eye view of ongoing research in the area of active databases. The Paper by Navathe, Tanaka, and Chakravarthy addresses the modeling and design of active databases and proposes an extension to the entity-relationship modeling approach using petri-nets. The paper by Ceri, Fraternali, Paraboschi, and Tanca discusses how ECA rules can be generated from specifications in function-free first order language. The paper by Widom provides a brief overview of the Starburst rule system. The paper by Gehani and Jagadish summarizes the active database facilities in Ode.

The paper by Gatziu and Dittrich discusses SAMOS, an active object-oriented database system currently under development at the University of Zurich. The paper by Risch and Sköld discusses another active object-oriented database system which uses OSQL and log screening filters for condition evaluation. The paper by Berndtsson and Lings discusses a prototype implementation of active functionality on top of ONTOS.

The next two papers cover a lot of ground as they discuss several ongoing research projects. The paper by Chakravarthy, Hanson, and Su highlights the results from three projects: Sentinel – an active object-oriented database system, Ariel – a database system dealing with efficient rule condition evaluation, and an active KBMS with its own language, data model (OSAM*), and support

for parallel and distributed computation. The paper by Dietrich, Urban, Harrison, and Karadimce discusses how active, deductive, and object-oriented paradigms are being integrated in an ambitious project.

The next four papers explore the applicability of the active database approach to a number of problems. The paper by Buchmann, Branding, Kudrass, and Zimmermann reports ongoing work on integrating heterogeneous repositories using a mediator based on the active paradigm. The paper by Sistla and Wolfson highlights the need for supporting triggers on database histories and discusses temporal languages for that purpose. The paper by Seligman and Kerschberg discusses how the active database approach can be used for approximate consistency maintenance in a federated environment. Finally, the paper by Urpí and Olivé outlines the deductive approach for supporting active database functionality.

I would like to thank all the authors for their contributions and cooperation in meeting the deadline on such a short notice. Also, I would like to thank all the authors for condensing their work to a mere 4 (in two cases 5) pages. I would like to thank Mr. Lionnel Maugis for providing feedback on all the papers as well as helping me with LaTeX. I sincerely hope that this issue provides a fish-eye view of the ongoing work on active databases (as intended) and the reader will benefit from the diversity of the topics covered.

Sharma Chakravarthy

Associate Professor
Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611
email: sharma@snapper.cis.ufl.edu

Active Database Modeling and Design Tools: Issues, Approach, and Architecture*

S.B. Navathe A.K. Tanaka† S. Chakravarthy
Georgia Institute of Technology University of Florida
College of Computing Department of C.I.S.
sham@cc.gatech.edu sharma@cis.ufl.edu

1 Introduction

Although a lot of research is being done on active databases, and a few commercial relational DBMSs already provide support for some active database capabilities (e.g. Ingres, InterBase, Oracle, and Sybase), to the best of our knowledge, currently there are no design tools that can take full advantage of these new capabilities.

Using current relational database design methodology, the specification of active behavior in the form of triggers/rules and stored procedures has to be done after-the-fact, i.e., after the translation of the conceptual schema into the relational schema. This implies that major design decisions regarding the behavior of the database are deferred to a later stage of the design process, where the semantics of the real- world situations may be obscured by the intricacies of the implementation model. At this stage, designing the active behavior of a database for a given set of applications is usually a difficult task, because of the inherent complexity and non-deterministic aspect of rule-based programming. There is evidence that users do not adequately exploit the functionalities of rules, triggers, and stored procedures because of the complexity of their design at the DBMS level. Actually, some DBMS vendors offer the "knowledge management" component as an optional rather than a standard resource of their products.

We propose the modeling of active database behavior at earlier stages of the database design process, by extending the entity-relationship (ER) approach with events and rules as objects of the model (which we call $(ER)^2$ model [TNCK91]), as well as providing language and tool support for translation of events and rules into the language constructs of target DBMSs. Furthermore, we provide a graphical interface as an extension to ER diagrams to facilitate the modeling of events, rules, and their interaction with ER objects. It turns out that this representation can be mapped into high level Petri nets [Jen91], which we use as a formalism for the semantics of event and rule processing. The resulting graph, called event/rule network, is then used for the purpose of analysis and validation of the design. The extended database design process is illustrated in Figure 1, where the shaded boxes represent the steps that have been added or extended (the dashed lines illustrate another dimension of the database dynamics, the process dimension realized by transactions, that is orthogonal to the kind of dynamics related to active database behavior).

2 Design and Translation of Active Functionality

Current design tools contain capabilities for specifying the conceptual schema by editing ER diagrams and automatically mapping it into an equivalent normalized relational schema, usually generating the data definition statements to create the schema for the target relational DBMS. Moreover, some advanced tools such as the LBL tools [SM91, MF91] are able to generate rule/trigger definitions for enforcing referential integrity constraints when supported by the DBMS, as well as to store design information (ER schema, relational schema, and their mapping) in a meta-database defined in the DBMS.

*This work is (in part) supported by the Office of the Naval Technology and the Navy Command, Control and Ocean Surveillance Center RDT&E Division.
†Supported by the Brazilian Army and CNPq

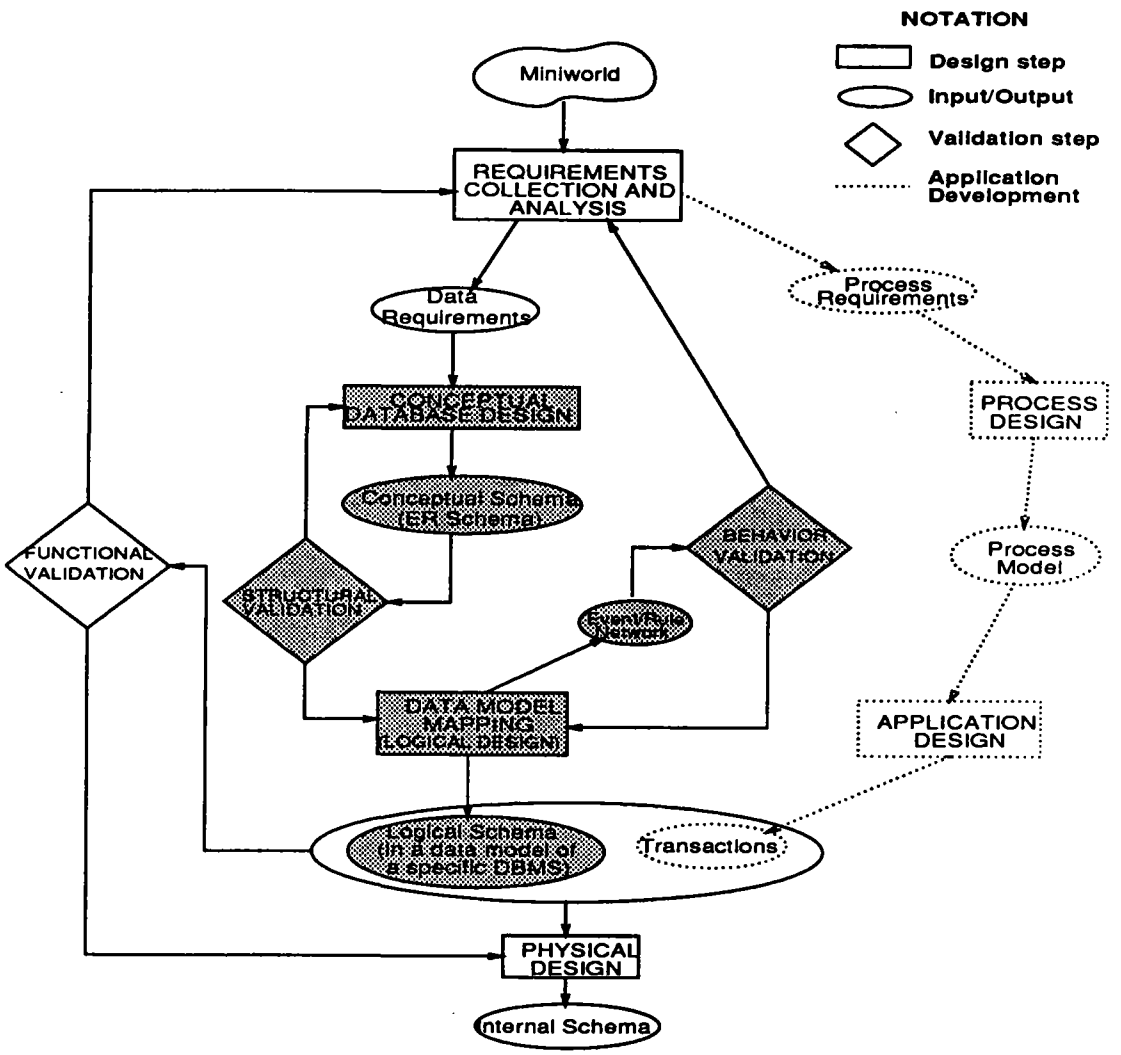

Figure 1: Extension to the Database Design Process using the $(ER)^2$ Model

Our approach relies on the design information stored in the meta-database to translate the specification of events and rules into corresponding DBMS language constructs (rules, triggers, procedures).

Figure 2 shows the self-representation of the $(ER)^2$ model, i.e., a meta-schema that shows events and rules as objects of the model in addition to entities and relationships, as well as the inter-object connections. We represent events as circles and rules as parallelograms; directed edges represent connections between events and rules, and between events and ER objects. In an actual $(ER)^2$ diagram, there is no need to label "Fires" and "Raises" arcs, since the connections between events and rules are implicit: an event "fires" rules while a rule "raises" events. Also, the "Precedes" relationship between events is implicitly defined by the unique time of occurrence of each event. The "Priority" relationship between rules needs not be explicitly represented in the diagram, as it is specified in the textual definition of the rules. The "Affects" and "Affected_by" connections are labeled with the type of the database event (modification, insertion, deletion, or retrieval) or the name of the signal in the case of non-database events.

The language construct that provides the extension has the following general format:

behavior_sentence ::= WHEN *event* FIRE *rule*

where an *event* can be a database event or a signal issued by the external environment (the underlying system, applications, or users) and a *rule* consists basically of an optional condition and a list of actions. The active behavior of a given database is specified as a set of *behavior_sentences*.

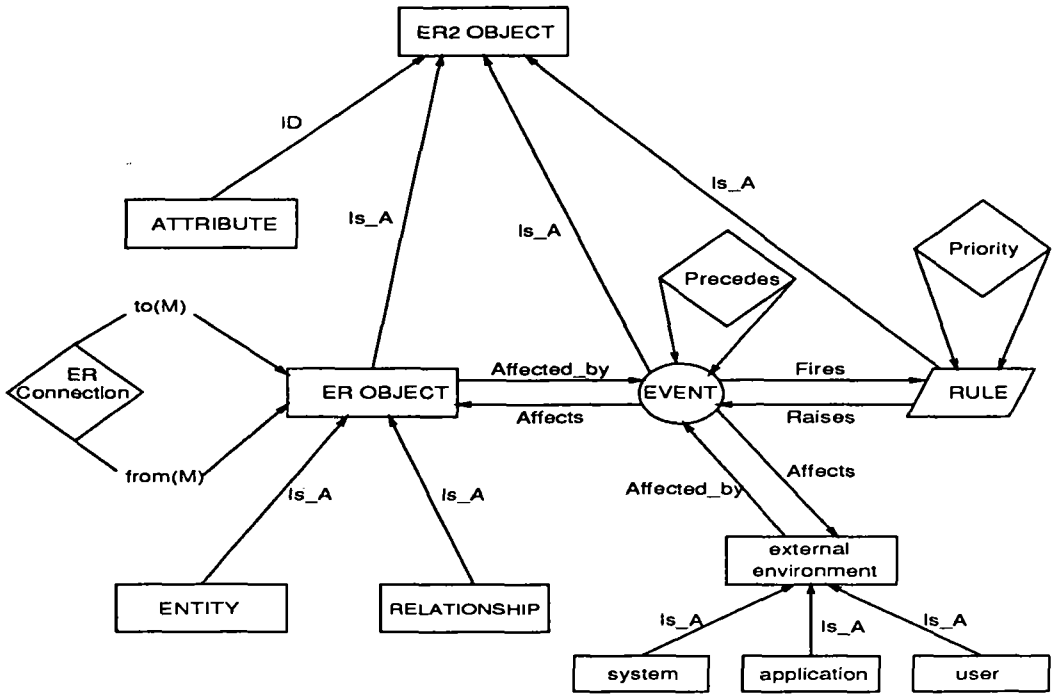

Figure 2: Meta-schema of the $(ER)^2$ Model

Although our approach is general, from a practical viewpoint, we restrict the power of the specification language for events, conditions and actions to the capabilities that are currently present in the target DBMSs. This ensures that the gap between conceptual and logical design is bridged for extant systems, and further extension to the DBMS capabilities will give rise to corresponding extensions hopefully in a straightforward manner. Because of the higher level of abstraction at which active behavior is modeled, the specification language is much more concise and more meaningful. Furthermore, special operations like PROPAGATE and REJECT can generate several SQL statements to enforce general integrity rules.

Constraint maintenance can be achieved by deriving *behavior_sentences* from a declarative constraint specification in a manner similar to derivation of rules from SQL-based constraints [CW90], and then translating them along with other *behavior_sentences*. Not all types of constraints need to go through this constraint to *behavior_sentence* to rules/triggers mapping. Dynamic constraints, that refer to the consistency of state transitions rather than to a single state (e.g. "a salary never decreases") is more simply specified directly with a *behavior_sentence* instead of trying to extend the constraint language to consider multiple states. A special type of constraint, that is implied by the invariant properties of the ER model [SSW80], if not supported declaratively by the DBMS, is specified as "meta-behavior", i.e., behavior over all entities and all relationships, and instantiated to *behavior_sentences* by the tool for a particular database (again the meta-database of the design is central to these transformations).

3 Validation of Active Behavior

The representation of active database behavior in an $(ER)^2$ diagram can be mapped into a high level Petri net, in which the *places* are events, the *transitions* are rules, and the *net inscriptions* are the definitions of events and rules in their specification language. Also, the underlying ER schema is implicitly declared as the net declaration part containing the token types. For database events, the tokens are tuples of ER objects that are carried from events to rules and referred in the evaluation of conditions and execution of actions. Non-database events, when supported by the DBMS, will have signal parameters as attributes as well. The combination of the individual event/rule nets (e/r-nets) representing *behavior_sentences* by merging common places results in an e/r-network, a (possibly disconnected) bipartite directed graph that

represents the processing done by the DBMS to support active capability.

The processing model implied by the e/r-network cannot be used as the execution model of the active DBMS because it does not include the processing of application transactions. Rather we use the e/r-network as an analysis tool that helps the database designer to detect inconsistencies in the set of *behavior_sentences* by using a Petri net editor/simulator.

An e/r-network is consistent if: 1) There are no conflicting situations involving rules that are not mutually exclusive; 2) There are no coordination situations involving events that are not conjunctive; and 3) The execution of every cycle terminates.

Because of the required knowledge of the applications semantics, the analysis of an e/r-network based on this definition of consistency must rely on the intervention by the user. Some limited assistance can be given by the tool like in production rule systems or truth maintenance systems.

4 Conclusions

We have taken the approach of considering events as first-class objects too, rather than only rules [DBM88]. The extensions we have made are on the conceptual design level, and serve as tools for specifying, validating, and translating active behavior into executable specifications at the DBMS level.

We are implementing the extended tools architecture on top of the LBL tool set.

The following benefits will result from the extended modeling and design methodology: reduced database design and application development effort with automatic generation of meta-behavior and translation of active behavior into executable DBMS language constructs; better control of application development; and better quality of the overall design.

Further details of the work reported here may be found in [Tan92].

5 Acknowledgements

The authors thank Victor Markowitz and Arie Shoshani for making the LBL tools available for this work.

References

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the International Conference on Very Large Data Bases*, 1990.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database management system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1988.

[Jen91] K. Jensen, editor. *High-Level Petri Nets: Theory and Applications.* Springer-Verlag, 1991.

[MF91] V.M. Markowitz and W. Fang. SDT: A database schema design and translation tool - reference manual. Technical Report LBL-27843, Lawrence Berkeley Laboratory, 1991.

[SM91] E. Szeto and V.M. Markowitz. ERDRAW: A graphical schema specification tool - reference manual. Technical Report PUB-3084, Lawrence Berkeley Laboratory, 1991.

[SSW80] P. Scheuermann, G. Schiffner, and H. Weber. Abstraction capabilities and invariant properties modeling within the entity-relationship approach. In P.P.S. Chen, editor, *Proceedings of the International Conference on the Entity Relationship Approach.* North-Holland, 1980.

[Tan92] A.K. Tanaka. *On Conceptual Design of Active Databases.* PhD thesis, Georgia Institute of Technology, 1992.

[TNCK91] A.K. Tanaka, S.B. Navathe, S. Chakravarthy, and K. Karlapalem. ER-R: an enhanced ER model with situation-action rules to capture application semantics. In T.J. Teorey, editor, *Proceedings of the International Conference on the Entity Relationship Approach*, 1991.

Constraint enforcement through production rules: putting active databases at work

Stefano Ceri, Piero Fraternali, Stefano Paraboschi, Letizia Tanca
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.zza Leonardo da Vinci 32
20133 Milano - Italy
e-mail: {ceri,fraterna,parabosc,tanca}@ipmel2.elet.polimi.it

Abstract

This paper presents an approach to the automated correction of constraint violations produced by transactions, in the context of active databases with integrity constraints. Constraints are expressed as formulas in a function-free first-order language; we then automatically generate production rules that detect constraint violations and propose repair actions. In this proposed architecture, transaction execution can lead to inconsistent states; production rules are then run to compensate violations and achieve a final state which is consistent and represents the user's intended semantics.

Our mechanism exhaustively considers compensations that can be syntactically generated from a given constraint; then it eliminates some of them which are obviously not correct. We remain with a rule set that is normally redundant and contains rules that may trigger each other, possibly leading to nonterminating execution. Therefore, a rule analyzer is used to choose a subset of these rules, so that termination of execution is ensured, a high number of constraints is compensated, and the user's intentions are respected.

In this paper, we outline the structure of such constraint-enforcement architecture and illustrate the problems that need to be solved for effectively compensating constraints. We also present some experimental results obtained by a prototype, and compare rules generated manually with rules generated by our prototype.

1 Introduction

The availability of production rule systems inside DBMSs, yielding so-called *active database systems*, is a challenging opportunity in order to extend the power of current database technology. In particular, integrity maintenance is one of the most promising areas for active databases.

When a database is being conceptually designed, a major effort is devoted to capture all those data interactions and restrictions that must hold in order for a physical configuration of the database to make sense. Unfortunately, due to the lack of an appropriate technology, a minor part of this effort becomes part of the physical design of the database—typically, only very simple constraints are captured by keys or referential integrity. The remaining prescriptions are then doomed to become part of the application requirements under the control of application designers, so that database integrity maintenance becomes more a matter of software discipline than an inherent property of the database schema. To contrast this phenomenon, two directions are currently being pursued by researchers in the database community:

- The evolution of data models towards object orientation, sustained by a parallel development of their deductive capabilities, to make some design concepts directly enforceable by new generation systems.
- The definition of a new conceptual and technological support to data integrity, in the form of constraint specifications and of techniques for their enforcement, independent of the data model and of the physical data representation.

Though the above directions may seem to diverge, they are complementary. The richer semantics of object-oriented models enables expressing constraints both on the structure and the evolution of databases.

However, not all of the semantics of a problem domain can be captured through structure and behavior constraints of database models; some semantic constraints require arbitrary, application-dependent statements. Thus, a declarative formulation of properties and/or behaviors must be supported in addition to schema structures; these properties must be enforced through several, alternative actions to be executed upon violation.

Our efforts are directed to provide a methodological framework for the automatic derivation of production rules maintaining a given set of constraints, specified through high-level declarations. We express data, constraints, and rules using the relational data model; however, our results can be restated in the context of a more powerful, object-oriented data model.

The idea behind this approach is the following. Assume that a transaction, applied to an initial state S, performs a sequence of operations and produces a state S' which falls outside the domain of the possible values. A typical reaction to this situation would be to roll back the transaction. In our approach, instead, we execute additional changes to database states, by means of production rules, until we reach an admissible final state S''. Further, we design rules so that the final state S'' be as "close" as possible to S', thus trying to capture the user's intentions.

This work is an extension of the approach described in [2] where production rules were used in order to enforce integrity. Rules in [2] are semi-automatically produced from constraints; our approach describes a system capable of providing a fully automatic solution to the same problem. The system still requires user's support, but only in the form of supervision of the process. This work used some results of [3]; in particular we adopted the constraint language introduced in [3].

2 Architecture

Fig. 1 shows the components of the architecture which we propose for an integrity maintenance system based on the active database paradigm.

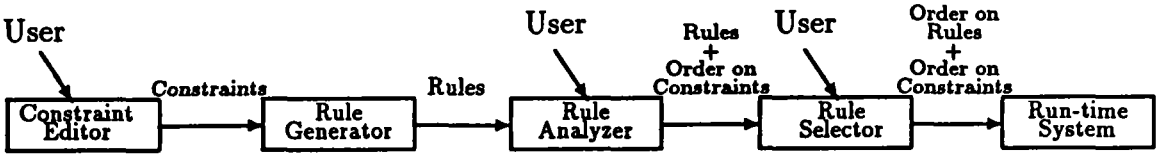

Figure 1: Architecture of the integrity maintenance system

The *Constraint Editor* is used for collecting constraints definitions.

The *Rule Generator* translates automatically constraints into the complete set of production rules that enforce them.

The *Rule Set Analyzer* is a component, possibly interactive, that analyzes and resolves situations in which rules may trigger each other in cycle, so that the termination of constraint-enforcing rules can be ensured. The actual product of this component is a partial order on the constraint set.

The *Rule Selector* provides a total order on the rule set, by identifying the compensating actions that should be used to maintain database integrity. This ordering may be decided at compile-time or at run-time; the former approach guarantees higher efficiency, but run-time selection can be more accurate, because more information is available about the transaction history and the database instance. In our prototype we implemented a compile-time approach.

The *Run-time System* is responsible of execution control after a user-supplied transaction. It should either be built on top of an existing active DBMS, or be provided by the run-time system of an active DBMS by writing rules appropriately.

3 Constraint language

An important characteristic of our system is the language used to express constraints: a constraint in *standard conjunctive form (s.c.f.)* is an evaluable closed formula of Domain Relational Calculus having the following pattern:

$$\forall \vec{x}\, \exists \vec{w}\, \neg(p_1(\vec{x}) \wedge \ldots \wedge p_n(\vec{x}) \wedge \neg q_1(\vec{x}, \vec{w}) \wedge \ldots \wedge \neg q_m(\vec{x}, \vec{w}) \wedge G(\vec{x}))$$

Following [3], we also call this form *denial*[1]. We also assume, without loss of generality, that all literals p_i, q_j represent database relations and are positive; G is a generic predicate. We apply the following evaluability conditions:

- Every universal variable x_i that appears in a literal q_i must also appear at least in one literal p_i.
- Variables that appear in the subformula G must also occur at least in one literal p_i.

This constraint language is less expressive than relational algebra or first order logic and it does not express aggregate functions and recursion, but we are working to extend its expressive power. It is important to say that a large fraction of real systems' constraints is expressible by the above language.

A typical integrity constraint is the following, that says that each type in the WIRE table has to appear in the WIRE-TYPE table:

```
∀ w-type, power ∃ max-volt, maxpow, cross-section
¬(WIRE(w-type, power) ∧
  ¬WIRE-TYPE(w-type, max-volt, max-pow, cross-section))
```

This is an example of a Starburst rule maintaining the above constraint:

```
create rule A on wire
when (inserted, updated(type)),
then 'delete from wire
  where type not in (select type from wiretype)';
```

We have illustrated in [4] how it is possible to generate the set of all the possible compensations for every possible constraint expressed with our language. Rule generation will produce a redundant set of compensating rules.

4 Rule Analysis and Selection

The computational behavior of a rule-based system for constraint maintenance is, in general, neither terminating nor deterministic, since the rule set comprises compensating actions that enforce the same constraint in different ways and rules that can trigger each other.

Our strategy for guaranteeing termination is the following: we determine a partial order on the constraint set such that normally the compensation of constraints at a given level can only violate lower-order constraints. If no rule may violate higher-order constraints, then termination is ensured. Thus, the rule analysis tool aims at selecting constraint-enforcing rules so that such partial order is established; this may not be possible in general. The run-time system is then responsible of carefully executing those rules which may violate higher-order constraints.

Rule analysis assumes as input a Triggering Hypergraph (THG) which describes rule interferences. A THG is a directed labeled hypergraph; nodes represent constraints and hyperarcs represent rules so that an hyperarc from C_1 to C_2 and C_3, labeled r_1, indicates that rule r_1 compensates constraint C_1 but may violate constraint C_2 and C_3. Potential violation is statically determined in a conservative way, by inspecting the signature of actions used by the rules. In fact, rule action execution at run-time depends on the actual values of its variables and may not violate any constraint.

Rule analysis is then reformulated as follows: determine the "optimal" set of hyperarcs to be removed from a THG, so that it is reduced to a directed acyclic hypergraph (DAHG) such that the rules in DAHG

[1] The s.c.f. is a semantically equivalent alias of the more common implicative form $\forall \vec{x}\,(p_1(\vec{x}) \wedge \ldots \wedge p_n(\vec{x}) \rightarrow \exists \vec{w}\, q_1(\vec{x}, \vec{w}) \vee \ldots \vee q_m(\vec{x}, \vec{w}) \vee G(\vec{x}))$, used in [3].

satisfy the "maximal" number of constraints. If a system executes only these rules, then termination is guaranteed.

A first approach to tackle rule analysis is to rely completely on user's intervention. In this case, the system assists the user in the manual browsing of the THG, by detecting cycles and presenting them to the user, who keeps the entire responsibility of choosing which rules are to be removed or modified [2]. However, in real-world cases, the THG tends to be intricate even with few constraints, which make extensive browsing impractical and suggest the convenience of developing an automated problem-solver to perform rule analysis.

The automatic problem-solver has to consider the relative adequacy, from a semantic viewpoint, of alternative compensating actions. This adequacy was represented with a weight assigned to each hyperarc, yielding to the following problem formulation:

Given a directed hypergraph H: $\{V, A\}$, a weight function $f : A \rightarrow N$.

Question: find a subset A' of A such that:

- the hypergraph H': $\{V, A'\}$ is acyclic
- for every subset A'' of A, distinct from A', such that the corresponding hypergraph H'': $\{V, A''\}$ is acyclic, holds that $\sum_{\forall v_i \in V} \text{weight}'(v_i) > \sum_{\forall v_i \in V} \text{weight}''(v_i)$

  $\text{weight}(v_i) \equiv \text{Max}(f(a_j),\ \text{tail}(a_j) = v_i)$

The above formulas indicate that the hypergraph weight is equal to the sum of the weights of the heaviest arc exiting from every node. This problem is known to be NP-complete [5]. Since the dimension of the problem can be quite relevant, it is impossible to solve the problem with an exhaustive approach, but it is necessary to develop approximate techniques. Therefore, we have developed several heuristic techniques, by investigating different goals of the problem-solver and different metrics to use in the resolution process. Experiments, which are next reported, showed that the difference between the automatic and human solution could be greatly reduced by improving the accuracy of the weights of the compensating actions.

5 Run-Time System

In order for the system to be correct, i.e. always compensating and terminating, it is necessary that each constraint be compensated by at least one rule contained in the DAHG, which is not guaranteed in general. Therefore, it is necessary to implement a Run-time System that executes a strategy for conflict resolution and execution control. This strategy can be outlined as follows:

a. While there is a rule in the DAHG that can enforce a violated constraint, execute it.

b. While there are constraints whose violation can only be compensated by a flagged rule (i.e., one not in the DAHG), then simulate its execution; if the rule does not violate higher-order constraints, then make the effects of its execution permanent; otherwise, try another flagged rule.

c. If both (a) and (b) fail, then rollback the transaction.

6 Experimental results

To execute tests and provide an experimental base of our proposal, we built a prototype of rule analysis and generation components; we are going to conduct in the near future some experiments using an active database to test the whole architecture. The prototype is a program that accepts as input a description of the relations with their attributes, keys and modifiability level. It then accepts constraints, described in the standard conjunctive form. The prototype generates as output a set of Starburst rules.

The best way to evaluate the quality of the solution was to compare a set of rules that was illustrated in [2] for a particular problem, developed manually by Ceri and Widom, with the solution provided automatically by the system for the same problem. The results of this comparison were encouraging, since 10 out of the 15 constraints in the example were enforced exactly in the same way; 3 constraints were compensated in a slightly different manner; 1 constraint had a different compensation, though reasonable; and 1 constraint was not compensated.

7 Conclusion

The objective of the research reported in this paper is developing a system that writes production rules for enforcing a given set of constraints with the minimum help from the user. We believe that user supervision cannot be eliminated, but a tool may be very helpful due to the regularity in writing compensating rules. Full exploitation of these regularities will drive us in the development of a powerful constraint definition language, that will permit to specify the user's preferred resolution strategies together with constraints.

Our work will pursue also other goals in the near future:

- Increment the experimental base: the experiments that we have already done were very useful in directing our work and we expect additional indications from further experiments
- Implement the global architecture: we have tested only the compile-time component.
- Evaluate the user's interface and interaction pattern.
- Consider incrementability: we will investigate how to deal with small variations in the constraint set, by changing the compensating rules accordingly.

Acknowledgment

We thank Jennifer Widom for giving us the opportunity of testing our ideas in the context of the Starburst Production Rule System.

References

[1] A. Aiken, J. Widom, J. M. Hellerstein "Behavior of database production rules: termination, confluence and observable determinism", Proc. ACM-SIGMOD, pp. 59-68, S. Diego, May 1992.

[2] S. Ceri, J. Widom "Deriving production rules for constraint maintenance", Proc. 16th VLDB, pp. 566-577, Brisbane, Australia, August 1990.

[3] S. Ceri, F. Garzotto, G. Gottlob "Specification and management of database integrity constraint through logic programming techniques", Tech. Rep., Laboratorio di Calcolatori, Dipartimento di Elettronica, Politecnico di Milano, 1991.

[4] P. Fraternali, S. Paraboschi, L. Tanca "Automatic rule generation for correction of constraint violations in active databases", 4th Int. Workshop on Foundations of Models and Languages for Data and Objects, Volkse, Germany, October 1992.

[5] P. Fraternali, S. Paraboschi "Selecting rules for constraint maintenance: its complexity and a heuristic solution", Tech. Rep. 76-92, Laboratorio di Calcolatori, Politecnico di Milano.

The Starburst Rule System: Language Design, Implementation, and Applications

Jennifer Widom
IBM Almaden Research Center *

Abstract

This short paper provides an overview of the Starburst Rule System, a production rules facility integrated into the Starburst extensible database system. The rule language is based on arbitrary database state transitions rather than tuple- or statement-level changes, yielding a clear and flexible execution semantics. The rule system was implemented rapidly using the extensibility features of Starburst; it is integrated into all aspects of query and transaction processing, including concurrency control, authorization, recovery, etc. Using the Starburst Rule System, we have developed a number of methods for automatically generating database rule applications, including integrity constraints, materialized views, deductive rules, and semantic heterogeneity.

1 Introduction

The *Starburst Rule System* is a facility for creating and executing *database production rules*; it is fully integrated into the Starburst extensible relational database system at the IBM Almaden Research Center. Production rules in database systems (also known as *active database systems*) allow specification of database operations that are executed automatically whenever certain events occur or conditions are met. In most active database systems, including Starburst, production rules are a persistent part of the database and are created using a *rule definition language*. As users and applications interact with data in the database, rules are triggered, evaluated, and executed automatically by a *database rule processor*. In developing the Starburst Rule System we had two major goals:

- Design of a rule definition language with a clearly defined and flexible execution semantics
- Rapid implementation of a fully integrated rule processor using the extensibility features of Starburst

As we developed and experimented with our language and system, we discovered that the inherently unstructured nature of rule processing makes production rules quite difficult to program. Consequently, we added as a third goal:

- Development of methods for specifying common classes of database rule applications in high-level languages and compiling these specifications into Starburst rules

The remaining three sections of this short paper outline the approaches we have taken to meeting each of these three goals. Further details on language design appear in [WF90, Wid92], further details on implementation appear in [WCL91], and further details on applications appear in [CW90, CW91, CW92a, CW92b, Wid91].

2 Language Design

There are two important aspects in the design of a database production rule language: the syntax for creating (as well as modifying, deleting, and grouping) rules, and the semantics of rule processing at run time. Most database production rule languages have a similar syntax, relying on and extending the syntax of the database query language. However, the semantics of rule processing varies considerably.

In Starburst, the syntax for creating a rule is:

create rule *name* **on** *table*
when *triggering operations*
[**if** *condition*]
then *action*
[**precedes** *rule-list*] [**follows** *rule-list*]

*Address: 650 Harry Road, San Jose, CA 95120 E-mail: widom@almaden.ibm.com

The *triggering operations* are one or more of **inserted**, **deleted**, and **updated**($c_1, .., c_n$), where $c_1, .., c_n$ are columns of the rule's *table*. The optional *condition* is an arbitrary SQL predicate over the database. The *action* is an arbitrary sequence of database operations, including SQL data manipulation commands, data definition commands, and **rollback**. The optional **precedes** and **follows** clauses are used to partially order the set of rules: if a rule r_1 specifies a rule r_2 in its **precedes** list, or if r_2 specifies r_1 in its **follows** list, then r_1 has higher priority than r_2. Commands also are provided to **alter**, **drop**, **deactivate**, and **activate** rules. *Rule sets* may be created; each set contains zero or more rules, and each rule belongs to zero or more sets.

Rules are processed at *rule processing points*. There is an automatic rule processing point at the end of each transaction, and there may be additional user-specified processing points within transactions. We first describe end-of-transaction rule processing. The semantics is based on *transitions*—arbitrary database state changes resulting from execution of a sequence of SQL data manipulation operations. The state change created by the user transaction is the first relevant transition, and some rules are triggered by this transition. As triggered rule actions are executed, additional transitions are created which may trigger additional rules or trigger the same rules additional times. Rule processing is an iterative algorithm in which:

1. A triggered rule R is selected for *consideration* such that no other triggered rule has priority over R (for further details on Starburst's rule ordering strategy see [ACL91])
2. R's condition is evaluated
3. If R's condition is true, R's action is executed

For step 1, a rule is triggered if one or more of its triggering operations occurred in the composite transition since the last time the rule was considered, or since the start of the transaction if the rule has not yet been considered. The effect of this semantics is that each rule sees each modification exactly once. Rule processing terminates when a **rollback** action is executed (in which case the entire transaction aborts), or when there are no more triggered rules.

Within a transaction, rule processing can be initiated by commands **process rules**, **process ruleset** S, or **process rule** R. Command **process rules** invokes rule processing with all rules eligible to be considered and executed; command **process ruleset** S invokes rule processing with only rules in set S eligible to be considered and executed; command **process rule** R invokes rule processing with only rule R eligible to be considered and executed. The semantics of rule processing in response to each of these commands is identical to end-of-transaction rule processing.

Rule conditions and actions may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule conditions and actions may refer to *transition tables*, which are logical tables reflecting the changes that have occurred during a rule's triggering transition. Transition table **inserted** in a rule refers to those tuples of the rule's table that were inserted by the triggering transition; transition tables **deleted**, **new-updated**, and **old-updated** are similar.

3 Implementation

The Starburst rule language as described in Section 2 is fully implemented, with all aspects of rule definition and execution integrated into normal database processing. The implementation took about one woman-year to complete; it consists of about 28,000 lines of C and C++ code including comments and blank lines (about 10,000 semicolons). The implementation relies heavily on several extensibility features of the Starburst database system [H+90].

We briefly outline the rule system's general design; many details necessarily are omitted. Rule and rule set information is stored in *rule catalogs*, portions of which are cached in global main memory structures (i.e. structures shared by all processes). During query processing, the system monitors data modifications that may trigger rules. Monitoring takes place using Starburst's *attachment* extensibility feature: based on the current set of rules, attachment procedures are registered by the rule system to be called on relevant tuple-level insert, delete, and update operations. These procedures enter the modifications in a local main memory structure (i.e. one structure per process) called a *transition log*. Automatic end-of-transaction rule processing is invoked by Starburst's *event queue* extensibility feature: if a transaction may trigger rules, a rule processing procedure is placed on an event queue to be invoked at the commit point of the transaction. (Hence, there is no overhead at all if a transaction does not trigger rules.) Rule processing also may be invoked by user commands, as described in Section 2. During rule processing, triggered rules are determined

using the transition log, and they are stored in a local main memory sort structure reflecting their ordering. The Starburst query processor is called to evaluate rule conditions and execute rule actions. Transition tables are implemented using Starburst's *table function* extensibility feature: table functions are referenced as tables in SQL but their contents are generated at run time by registered procedures. The rule system registers four procedures (*inserted*, *deleted*, *new-updated*, and *old-updated*) that produce the transition tables from the transition log as needed during condition evaluation and action execution. Finally, note that since the transition log is central to rule processing, it is highly structured for efficiency in its various operations.

A few special features were needed to fully integrate the rule system into Starburst. Since the query processor is called to execute rule conditions and actions, concurrency control for these operations is handled automatically. However, concurrency control for rule creation, modification, and deletion must be handled separately. The rule system includes concurrency control mechanisms that ensure consistency with respect to rules and data (i.e. the set of rules triggered by a given transaction's modifications cannot change during the course of the transaction) and with respect to rule ordering (i.e. the ordering between triggered rules cannot change during the course of rule processing). The rule system also includes an authorization component for rules and rule sets, and rollback recovery mechanisms for rule system data structures.

4 Applications

The Starburst Rule System provides a powerful mechanism that can be used for traditional database functions such as integrity constraints and derived data, for non-traditional database functions such as situation monitoring and alerting, and as a platform for large knowledge-base and expert systems. Unfortunately, developing a set of rules to correctly realize such applications can be a difficult task: rule processing is inherently dynamic and unstructured, it interacts with arbitrary database changes, and its behavior can be unpredictable and difficult to specify.

We have taken two approaches to the problem of developing rule applications. In the first approach, support is provided to the rule programmer in the form of *analysis tools*. These tools perform static analysis on a set of Starburst rules to predict (conservatively) whether the rules are guaranteed to terminate, whether they are guaranteed to produce a unique final database state independent of the ordering between non-prioritized rules, and whether they are guaranteed to produce a unique stream of "observable" actions independent of the ordering between non-prioritized rules; this work is reported in [AWH92]. Our second approach is based on the observation that, unlike rules themselves, many common rule applications are static, structured, and easy to specify. Hence, we have developed a suite of methods whereby the user can specify rule applications using a high-level declarative language, and these specifications are translated (fully- or semi-automatically) into rules that implement them. We briefly describe four such classes of applications.

Integrity constraints – Integrity constraints are static predicates over the database that must be true at certain *consistency points* (usually the end of each transaction). Starburst rules can be used to monitor and enforce integrity constraints: for each constraint, a rule is triggered by any database modifications that may violate the constraint, the rule's condition checks whether the constraint actually is violated and, if the condition is true, the rule's action restores the constraint or rolls back the transaction. We have developed a method whereby the user specifies constraints as SQL predicates over the database. From an arbitrary set of constraints, the system semi-automatically derives a set of rules that are guaranteed to maintain the constraints [CW90].

Materialized views – Views are logical tables specified as queries over *base* (stored) tables. When a view is materialized, the view table is stored in the database and must be kept consistent with the base tables. Starburst rules can be used to maintain materialized views: whenever base table modifications may affect the value of a view, rules are triggered whose actions modify the view accordingly. We have developed a method whereby the user specifies views using SQL, then the system automatically derives a set of Starburst rules that are guaranteed to maintain materializations of the views in an incremental fashion [CW91].

Deductive rules – Similar to views, deductive rules specify logical tables derived from base tables. However, deductive rules use a recursive logic programming formalism, which is more powerful than SQL views. Similar to materialized views, Starburst rules can be used to maintain materialized derived tables specified by deductive rules: whenever base table modifications may affect the value of a derived table, rules are triggered whose actions modify the derived table accordingly. If derived tables are non-materialized (i.e. produced on demand rather than stored in the database), then Starburst rules can be used to perform the

iterative *semi-naive* evaluation mechanism often used for deductive rules. We have developed methods for automatically deriving Starburst rules from deductive rules for both of these approaches [CW92a, Wid91].

Semantic heterogeneity – Semantic heterogeneity occurs when multiple databases model the same real-world entities in different ways. Whenever possible, it is desirable to maintain consistency across such databases, despite the heterogeneity. While Starburst rules alone are not sufficient for this, they can be used together with a *persistent queue* mechanism. At each database, rules are triggered by any modifications that may violate consistency with other databases. Semantic heterogeneity is encoded in rule conditions and actions so they can perform remote read operations to determine whether consistency actually is violated and, if so, perform local or remote update operations to restore consistency. Since multidatabase environments usually do not support transactions across sites, persistent queues are used for reliable and correct execution of remote operations. We have developed a method whereby the user specifies consistency requirements across semantically heterogeneous databases in a high-level language, then the system automatically derives Starburst rules that are guaranteed to monitor and enforce consistency [CW92b].

Acknowledgements

I am ever grateful to Stefano Ceri, Bobbie Cochrane, Shel Finkelstein, and Bruce Lindsay, all of whom made important contributions to one aspect or another of the Starburst Rule System.

References

[ACL91] R. Agrawal, R.J. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, September 1991.

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[CW92a] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. IBM Research Report, IBM Almaden Research Center, November 1992.

[CW92b] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. IBM Research Report, IBM Almaden Research Center, October 1992.

[H+90] L. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

[Wid91] J. Widom. Deduction in the Starburst production rule system. IBM Research Report RJ 8135, IBM Almaden Research Center, San Jose, California, May 1991.

[Wid92] J. Widom. A denotational semantics for the Starburst production rule language. *SIGMOD Record*, 21(3):4–9, September 1992.

Active Database Facilities in Ode

N. H. Gehani and H. V. Jagadish
AT&T Bell Laboratories, Murray Hill, NJ

The Ode object-oriented database provides powerful facilities for specifying constraints and triggers. These are associated with class (object type) definitions. In an active database, a trigger "fires" (executes its action part) upon the occurrence of the event specified in the trigger. In Ode, these events can be composite, specified as a pattern of primitive events.

1. INTRODUCTION

The Ode object database supports the C++ object paradigm. The primary user interface is O++, a database programming language based on C++ The O++ object facility is based on the C++ object facility and is called the *class*. O++ extends C++ by providing facilities to create persistent objects.

Ode provides two kinds of triggering facilities: "constraints" for maintaining database integrity and "triggers" for automatically performing actions depending upon the database state. While the constraint facility could be mapped into the more general trigger facility, we believe there is value to keeping the two distinct, from the perspective of both semantic clarity and implementation efficiency. A few key differences are:

1. Constraints ensure consistency of the object (and database) state. If this consistency cannot be maintained (on an object update basis or on a transaction basis), then the transaction is aborted. Triggers are not concerned about object consistency. They are fired whenever the specified conditions become true.
2. Constraints apply to an object from the moment it is created to the moment it is deleted. Triggers must explicitly be activated after the object has been created.
3. All objects of a given type have the same constraints. Different triggers may be activated for different objects even though the objects maybe of the same type. For example, an object representing stock A may have an active trigger to sell the stock if its price follows below some threshold. But the object representing stock B may not have any active triggers.

In this paper, we provide an overview of the constraint facilities of Ode, then the trigger facilities, and finally the composite event mechanism. See [1-5] for more detail.

2. CONSTRAINTS

Constraints are used to maintain a notion of consistency beyond what is typically expressible using the type system. Updates that violate the specified constraints should not be permitted. Interpretations of consistency are usually application specific and may be arbitrarily complex. Constraints, which are Boolean conditions, are associated with class definitions, and can be inherited like all other class properties. All objects of a class must satisfy all constraints associated with the class. Violation of a constraint, if not rectified, will abort the transaction causing the violation.

Constraints in Ode consist of two parts: a predicate and an action (or handler). This action is executed when the predicate is *not* satisfied. Constraint checking can be performed after accessing the object or at some later point in time. For example, in design applications, it is sometimes appropriate to defer constraint checking to just before the transaction commit instead of performing it right after accessing the object. This allows for temporary violations of constraints (which is likely to happen when the constraints of two objects depend upon each other's values and one of the objects is updated) that are rectified in actions following the object update before the transaction attempts top commit. Consequently, to support these two modes of constraint checking we support two kinds of constraints: hard and soft.

2.1 Hard Constraints

Hard constraints are specified in the constraint section of a class definition as follows:

```
constraint:
    constraint_1 : handler_1
    constraint_2 : handler_2
    ...
    constraint_n : handler_n
```

constraint$_i$ is a Boolean expression that refers to components of the specified class and *handler*$_i$ is a statement that is executed when a constraint is violated. Constraints are checked only at the end of constructor and member (friend) function calls (but not at the end of destructor calls). Although we do not prohibit accessing the public data components of an object directly, it is the programmer's responsibility to ensure that such accesses do not violate any constraints because no constraint checking is performed for such accesses.

The granularity of hard constraint checking is at the member function level. This has two important advantages: objects are always in a consistent state (except possibly during an update operation) and the implementation of constraint checking is simplified. The notion is that each public member function must leave the object in a consistent state.

Here is an example of a hard constraint:

```
class supplier {
    Name state;
    ...
constraint:
    state == Name("NY") || state == Name(""):
        printf("Invalid Supplier State\n");
};
```

After a `supplier` object has been created or accessed, the constraint is checked. The constraint is violated if the supplier's location is specified and it is not in New York (NY). The statement associated with the constraint will be executed and the constraint checked once again. If the constraint is still not satisfied, as it will not be in this particular example, then the transaction is aborted.

2.2 Soft Constraints

When multiple objects are involved in a constraint, it is usually not feasible to require that the constraint be satisfied after each object update. To handle such cases, we need a *deferred* or *transaction-level* constraint checking mechanism. Transaction level constraint checking is supported with *soft* constraints in Ode. Soft constraints are specified like hard constraints except that the keyword `soft` precedes the keyword `constraint`, e.g.,

```
class person
{
    ...
    persistent person *spouse;
public:
    ...
soft constraint:
    (spouse == NULL) || (this == spouse->spouse);
};
```

The above constraint specifies that if a person has a spouse, then the spouse's spouse must be the person himself/herself. If this were a hard constraint, it would never be possible to record a marriage or a divorce since the update of the first of two objects would violate the constraint temporarily and cause the transaction to abort.

2.3 Inter-Object Constraints

A constraint is said to be *intra-object* if it is associated with a (single) specific object, and the condition associated with it is evaluated only when this object is updated. Otherwise, a constraint is said to be *inter-object*.

An intra-object constraint can refer to other objects both in evaluating the condition and in the subsequent action. However, updates to these referenced objects do not require the condition part of the constraint or trigger to be checked.

Each inter-object constraint into one or more equivalent intra-object constraints manually or with the help of a pre-processor. See [4] for systematic technique for performing this conversion. Language support is provided in O++ for intra-object constraints, and for two particularly important cases of inter-object constraints: referential integrity and relational integrity.

2.3.1 Referential Integrity Referential integrity requires that any object referenced by another object actually exist. In an object-oriented system, references are recorded by means of object identifiers. Since the user has no way of generating or modifying object identifiers accidentally, the system can easily guarantee that a reference is valid at the time of creation

Suppose that an object to be deleted still has a reference to it. There are three standard maintenance options. The reference can be deleted as part of the transaction deleting the object (by placing a NULL in the reference pointer), the referencing object can be deleted, or the deletion of the object can be disallowed. We use the keywords `nullify`, `ripple`, and `abort`, respectively for the three possible actions.

2.3.2 Relational Integrity A binary relationship, known at schema definition time, is stored in an object oriented database as a directional reference (or set of references) from either participant in the relationship to the other. When such a relationship is to be updated, multiple updates have to be performed, one for each participant in the relationship, giving rise to the possibility that the relation is recorded differently at the different logical locations. *Relational integrity* in an object-oriented database is the proper maintenance of relationships recorded at multiple logical locations, ensuring that the recording is consistent. The keywords `ripple` and `abort` are used once again, meaning respectively that the action is to fix the reverse

pointer and that the action is to abort the transaction. While it is required that a pair of inverse attributes each declare the other as its inverse, it is permissible to have two different action policies for the two directions.

A few sample inverse declarations are given below:

```
class Emp {
  ...
  Dept* dept                inverse emps[[]] abort ;
  Emp*  officemates[[4]]    inverse officemates[[]] abort ;
  ...
}

class Dept {
  ...
  Mgr*  head()              reference abort ;
  Emp*  emps[[50]]          inverse dept ripple       reference nullify ;
  ...
}
```

The example above states that when a deletion is attempted on a `Mgr` object, the transaction should be aborted if this object is listed as the `head` of some `Dept` object in the database. When a `Emp` object is deleted, any reference to this object from the `Dept` this employee works should be nullified and the deletion allowed to commit. The `dept` attribute of class `Emp` is the inverse of the `emps` attribute of class `Dept`. An attempted change to the former without a coresponding change in the latter causes the transaction to abort, while an attempted change to the latter ripples any necessary change to the former.

3. TRIGGERS

Triggers, like integrity constraints, monitor the database for some conditions, except that these conditions do not represent consistency violations. A trigger, like a constraint, is specified in the class definition and it consists of two parts: an event predicate and an action. Triggers apply only to the specific objects with respect to which they are activated. Triggers are parameterized, and can be activated multiple times with different parameter values. If a trigger is active, the action associated with it is executed when the predicate becomes true.

3.1 The Mechanism

Triggers are associated with objects; they are activated explicitly after an object has been created. A trigger T_i associated with an object whose id is *object-id* is activated by the call:

object-id->T_i (*arguments*)

The trigger activation returns a trigger id (value of the predefined class `TriggerId`) if successful; otherwise it returns `null_trigger`. The object id can be omitted when activating a trigger from within the body of a member function.

An active trigger "fires" when its predicate becomes true (as a result of updates by a transaction). Firing means that the action associated with the trigger is "scheduled" for action as a separate transaction. Only active triggers can fire. No performance penalty is incurred for triggers that have not been activated.

Trigger activation must be done explicitly for each individual object. However, the class designer can automate trigger activation by putting the trigger activation code in constructors. Since a constructor function is called at object creation time to initialize the object, the trigger automatically gets activated when an object is created. Because triggers are activated explicitly (by the programmer or by the class designer), different objects of the same type may have different sets of triggers active at any given time.

Triggers can be deactivated explicitly before they have fired using the `deactivate` function:

`deactivate` (*trigger-id*)

The trigger with identifier *trigger-id* is deactivated. If successful, `deactivate` returns one; otherwise, it returns zero.

Multiple activations of the same trigger associated with an object (possibly with different arguments) are allowed. For example, there can be multiple activations of the buy trigger associated with a stock object with each buy trigger being activated with different price and quantity arguments.

Like constraints, neither the order of placement of triggers in a class definition, nor the order in which the triggers are activated, can be used to determine the order in which the triggers will be evaluated or executed.

3.2 The Constructs

Ode supports two kinds of triggers: *once-only* (default) and *perpetual* (specified using the keyword `perpetual`). A once-only trigger is automatically deactivated after the trigger has "fired", and it must then explicitly be activated again, if desired. On the other hand, once a perpetual trigger has been activated, it is reactivated automatically after each firing.

Triggers are specified within class definitions:

```
trigger:
    [ perpetual ] T1 (parameter-decl1) : trigger-body1
    [ perpetual ] T2 (parameter-decl2) : trigger-body2
    ...
    [ perpetual ] Tn (parameter-decln) : trigger-bodyn
```

T_i are the trigger names. Trigger parameters can be used in trigger bodies, which have one of these forms:

```
event-expression => trigger-action
within expression ? event-expression => trigger-action
                                    [ : timeout-action ]
```

The second form is used for specifying *timed* triggers. Once activated, the timed trigger must fire within the specified period (floating-point value specifying the time in seconds); otherwise, the timeout action, if any, is executed.

event-expressions are formed by using database events such as an object update or a transaction commit, event composition operators, and "mask expressions". In general, this is a powerful facitlity for expressing declaratively patterns of events of interest. In its simplest form, an event expression is simply an object update followed by a mask expression. For example,

```
after update & qty <= reorder_level()
```

The *trigger-action* (and the *timeout-action*) is written as one of:

```
{ ... }
independent { ... }
immediate { ... }
deferred { ... }
```

where `{ ... }` represents a statement of a set of statements in curly braces. By default, the action is executed as a separate transaction with a commit dependency (i.e., the triggered transaction is not allowed to commit until the triggering transaction has committed). However, a different coupling mode can be specified if desired by using the appropriate keyword from those shown above. `independent` causes execution in a separate transaction with no dependencies created, so that the triggered action can commit even if the triggering action does not. The other two possibilities are for execution of the triggered action within the same transaction, either immediately, or at the end of the transaction. These keywords have been introduced for solely for ease of expression. Given the powerful event expression capability, the only coupling mode required is immediate: all other couplings can be obtained by writing the appropriate event expressions.

4. CONCLUSIONS

We have provided facilities for constraints and triggers in O++ that match the object-oriented programming style of C++. Although constraints and triggers can be implemented using similar techniques, we have provided separate facilities for them since they are conceptually and semantically different. A pwerful event expression facility is used to specify the points at which trigger predicates should be checked.

REFERENCES

[1] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[2] N. H. Gehani, H. V. Jagadish and O. Shmueli, "Event Specification in an Active Object-Oriented Database", *Proc. ACM-SIGMOD 1992 Int'l Conf. on Management of Data*, San Diego, CA, 1992.

[3] N. H. Gehani, H. V. Jagadish and O. Shmueli, "Composite Event Specification in Active Databases: Model & Implementation", *Proc. of the 18th Int'l Conf. on Very Large Databases*, Vancouver, BC, Canada, Aug 1992.

[4] H. V. Jagadish and X. Qian, "Integrity Maintenance in an Object-Oriented Database", *Proc. of the 18th Int'l Conf. on Very Large Databases*, Vancouver, BC, Canada, Aug. 1992.

[5] H. V. Jagadish and O. Shmueli, "Synchronizing Trigger Events in a Distributed Object-Oriented Database", *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Alberta, Canada, Aug. 1992.

SAMOS: an Active Object–Oriented Database System

Stella Gatziu, Klaus R. Dittrich
Database Technology Research Group
Institut für Informatik, Universität Zürich
{gatziu, dittrich}@ifi.unizh.ch

1 Introduction

Most new developments in database technology aim at representing more real-world semantics in the database which would otherwise be hidden in applications. For instance, object-oriented database systems (ooDBS) provide for mechanisms to model complex structures and to express user-defined (procedural) behavior. *Active* database systems (aDBS) are able to recognize specific situations (in the database and beyond) and to react to them without direct explicit user or application requests. An aDBS registers ***situations*** (e.g. the occurrence of specific database operations), ***actions*** (executable programs including database operations) and the association between situations and (re)actions, by means of ***situation/action rules***. A situation is generally specified through an event and a condition, whereby an event indicates a point in time specified explicitly or by an occurrence in the database system or its environment. A condition relates to the current database state and has to be evaluated when the corresponding event is signalled; if it holds, the associated action has to be executed. We therefore talk about *ECA-rules* (Event-Condition-Action).

The combination of active and object-oriented characteristics within one, coherent system is the overall goal of SAMOS (*Swiss Active Mechanism-Based Object-Oriented Database System*). SAMOS addresses the three principal problems of an aDBS, namely rule specification, rule execution and rule management. We focus especially on rule specification and rule management. Rule specification is concerned with the nature of events, conditions and actions and their relationship to the data model. A specific contribution of SAMOS is its extensive collection of event specification features. Thus, an array of events indicating various kinds of occurrences in the database system and its environment is supported. Additionally, time specification facilities are integrated within event definitions. Rule management incorporates tasks for the internal processing of rules like event detection and selection of all rules that have to be fired.

SAMOS does not intend to propose yet another object-oriented data model (ooDM), rather it aims at exploiting how active mechanisms can be integrated with object-oriented database systems in a reasonable way. Because current ooDBS differ in their data models and further functionalities, we assume only characteristic properties provided by nearly all ooDMs like inheritance, user-definable types and operations, encapsulation, etc. The prototype implementation of SAMOS is based on the commercial ooDBS ObjectStore (to avoid programming an entire DBMS) and allows us to demonstrate active database features and to investigate their strengths and possible problems in concrete application environments.

In this paper, we give a short overview of SAMOS. Section 2 addresses the specification of events and section 3 presents the aspects of the integration of active mechanisms into an object-oriented database system. In Section 4, we talk briefly about the execution of rules within the framework of the transaction management component. Section 5 contains some implementations issues (e.g. event detection).

2 Event Specification in SAMOS

The broad usability of an aDBS requires that a variety of real-world situations can be modelled. SAMOS provides like Snoop [CM 91] and ODE [GJS 92] an ***event language*** that includes several constructs for the specification of events. An event can always be regarded as a specific point in time. The way how this point in time is specified, e.g. as an explicit time definition (e.g. at 18:00) or as the beginning or the end of a database operation, leads to various kinds of ***event classes*** *. Events can be roughly subdivided into two categories: ***primitive*** events which correspond to elementary occurrences, and ***composite*** events that are

*We distinguish between *event classes* and *event instances*: an event class is what we specify within a rule definition, while an event instance relates to the actual occurrence of an event (of a specific class). In the sequel, we will simply talk about *events* whenever the distinction is clear from the context

described by an *event algebra*. An expression of the event algebra is composed out of other composite or primitive events based on *event constructors*.

Primitive events

A primitive event describes a point in time specified by an occurrence in the database (*method* or *value* events), in the DBMS (*transaction* events), or its environment (*time* or *abstract* events). First of all, an event can be specified as an explicitly defined point in time. Those events are called *time* events. They can be defined as absolute points in time (`on February 28, 1992 at 22:00`), as periodically reappearing points in time (`every day at 18:00`) or relative to occurring events (`1 min after event E1`). In an object-oriented environment, users manipulate objects by sending messages to them. Thus, each message gives rise for two events (*method* events): the point in time when the message is "arriving" at the object and the point in time when the object has finished the execution of the appropriate method. A method event relates to one or more classes or to particular objects. In the first case, the event is signalled before or after the execution of the appropriate method on any arbitrary object of this class. Furthermore, an event can be (semantically) related to the modification of (parts of) the value of an object, which can take place in various methods. Thus, using message events only, it would be necessary to define one rule for each method where this modification occurs. Instead, we permit the definition of value operations as events (*value* events). Obviously, due to encapsulation, appropriate rules have to be regarded as part of the class definition and are definable/visible for the class implementor only. *Transaction* events are defined by the start or the termination of (user-defined) transactions and are raised as soon as any arbitrary transaction execution starts (or ends). Assuming that transaction programs are named, a transaction event can be restricted to transactions executing this one transaction program. Up to now, we have introduced several kinds of events which are conveying specific semantics known by SAMOS such that their occurrence can be detected by the system itself. However, users and applications may need other events according to their specific semantics as well (*abstract* events). They are defined and named by users and can be used in rules like any other event. Abstract events are not detected by SAMOS, but users/applications have to notify the system about their occurrence by issuing an explicit *raise* operation.

Composite events

The kinds of primitive events described above correspond to elementary occurrences and are not adequate for handling events that occur when some combination of other events happens. Thus, SAMOS supports composite events built from others by means of six event constructors. The *disjunction* of events (`E1|E2`) occurs when either `E1` or `E2` occurs. The *conjunction* of events (`E1,E2`) occurs when `E1` and `E2` occur, regardless of order. A *sequence* of events (`E1;E2`) occurs when first `E1` and afterwards `E2` occurs. The following three constructors monitor the occurrence of event instances of a specific event class during a predefined time interval. A composite event with the "`*`" -constructor (`*E`) will be signalled (and the corresponding rules will be executed) only once (after the first occurrence of `E`), even if the event `E` occurs several times during the specified time interval. A *history* event `TIMES(n,E)` is signalled when the event `E` has occurred with the specified frequency `n` during the specified time interval. A *negative* event (`NOT E`) occurs if `E` *did not* occur in a predefined time interval. Without giving a specific time interval, we assume the time between the event definition and infinitive. Thus, only negative events always require the definition of an explicit time interval. The way a time interval can be defined is discussed below in this section.

Event parameters

Event classes can be parameterized. The actual parameters are bound to the formal parameters of an event class during instantiation. The set of permitted formal parameters is fixed (except for abstract events). We differentiate between *environment* parameters (e.g. *occ_point(E)* as the point in time of the event occurrence and *occ_tid(E)* as the id of the transaction in which the event has occurred (*occurring transaction*)) and *event_kind* parameters depending on the event kind (e.g. method events have as parameters those of the method and the object_id of the object executing the method). The definition of composite events can be extended with the constructor *same(parameter_kind)* to denote that the event parts of the composition must have the same parameter of this specific kind. For instance the sequence (`E1;E2):same(tid)`) is signalled when `E1` and afterwards `E2` have occurred in the same transaction (the occuring transaction of `E1`). The composite event `*(E):same(object)` monitors the multiple occurrence of instances of the method event `E` executing on the same object.

Time intervals

Many cases require that a (primitive or composite) event `E` is signalled only in case it has occurred during a specific time interval `I`. Therefore, the notion of *monitoring intervals* is introduced for those time intervals during which the event has to be occurred. The event definition can be extended by the definition of the monitoring interval (e.g. `E IN I`). Especially, history, negative events and the "`*`"-constructor require a time interval. This leads to definitions like: `(*E)IN I` or `TIMES(n,E)) IN I`. A time interval generally is specified by two points in time, a *start_time* and an *end_time*. It can also be computed from other time

intervals: we provide two operators, *overlap* and *extend*, to represent the intersection and the union of intervals, respectively. Now, we present possibilities of the definition of *start_time* and *end_time*. They can be explicitly defined as absolute points in time e.g. `17.8.90,15:00`. However, the desired points in time may not be known in advance at event definition time. For instance, the *start_time* may in turn be defined as the point in time when an event occurs (i.e. relative to *occ_point*) or the execution of a rule completes. Therefore, SAMOS allows the implicit definition of points in time and supports a variety of specification facilities for time intervals. For example, the event `NOT E IN overlap([1.8.92-31.8.92],[occ_point(E)+2DAYS])` monitors the occurrence of `E` during two days in August after its first occurrence.

3 Integrating active Components into an OO Environment

The combination of active and object-oriented characteristics within one, coherent system is another major goal of SAMOS [GGD 91]. Using ooDBS characteristics like user-defined types, methods, inheritance or encapsulation increases the flexibility of an active mechanism twofold: first in that method and value events are supported and second in that rules are subject to encapsulation and inheritance and are represented as objects itself.

Rules and classes

Rules may be classified according to their relationship to the classes (or objects) they are defined for. In some active object-oriented systems ([GJ 91], [DPG 91]), each rule is defined as part of the class the appropriate event refers to. The kinds of events are restricted to method and value events (i.e. to operations on objects). We propose in SAMOS a different kind of relationship between rules and classes. In particular, certain rules may be permitted (*class-internal* rules) or prevented (*class-external* rules) to operate on or to access object values. Class-internal rules are part of the class definition. They are encapsulated within instances and are visible to the class implementor only. Therefore, value events (restricted to refer to objects of this specific class) are permitted beyond method events (not restricted to refer to objects of this class), time and abstract events (and combinations thereof). Conditions and actions of class-internal rules are also allowed (but not restricted) to operate directly on values. This property leads to a high level of object autonomy; specific tasks (e.g. some integrity constraint maintenance) which are relevant only to a certain object can be kept completely local to that object. However, actions of class-internal rules or methods may desire to notify the "outside world" about the occurrence of a specific state of an object (i.e. about a specific situation), although (or even because!) the state of the object is encapsulated and thus cannot be examined conventionally from outside the object. Such a situation may be turned into an abstract event by the class implementor. This event is made part of the class interface and "exported" to the outside world. Thus, the definition of class-external rules referring to such object-specific situations is made possible. Class-external rules can be defined by any user or application intependently of a class definition. Obviously, value operations cannot be used for class-external rules; i.e. neither for events, nor for conditions, nor for actions.

Inheritance of rules

In an object-oriented environment, rules are subject of inheritance. First, (class-internal or class-external) rules with method or value events relate to one or more classes. Thus, they can be propagated along class hierarchies. For example, suppose that `K2` is a subclass of `K1` and `M1` is a method of `K1`; a method event defined on `M1` is signalled because of the execution of `M1` on any object of `K1` or `K2`. Furthermore, class-internal rules are defined according to a specific class and thus propagated like methods along the class hierarchy.

Rules and rule components as objects

To stay in the same "world" and exploit its advantages, rules and rule components are represented as objects themselves and are instances of a class (e.g. "rule" or "event"). Obviously, all object-oriented characteristics are available for such a class like for any other class. For example, the instances of that class can be manipulated and accessed by means of methods like *define*, *delete* and *raise*.

4 Rule Execution

An aDBS executes rules in addition to conventional user transactions. Execution of a rule with an event E starts whenever E (the *triggering event*) occurs and consists of the evaluation of the condition and (if it is satisfied) the execution of the associated action (*triggered operations*). First of all, the rule definer specifies in SAMOS when a condition has to be evaluated and/or an action has to be executed relative to the triggering event by means of *coupling modes* (*immediate*, *deferred*, *decoupled* like in HiPAC). We examined in [GGD 91] the integration of the execution of triggered operations within a transaction model based on (generalized) multi-level transactions and semantic concurrency control. In this approach, condition evaluation and action execution are implemented as own (sub-)transactions. On the basis of semantic concurrency control on the

level of methods, the system has to be told about conflict relations over the set of methods of a class. Class-external rules call methods which in turn are synchronized with other methods and rules. Class-internal rules, on the other hand, can manipulate values of objects directly, and thus behave comparable to methods. Consequently, a class implementor has to provide conflict relations with condition or action parts of class-internal rules. Finally, SAMOS also handles the execution of multiple rules which are triggered by the same event, by means of *priorities*. However, this is only necessary when condition and action have the same coupling mode. In this case, the effect of the rules depends on the execution order: the action of one rule may invalidate the condition of others.

5 Implementation Issues

In addition to the usual functionalities of (passive) DBMS, an active DBMS has to perform tasks like the definition and management of rules and the efficient detection of events. First of all, an *analyzer* is responsible for giving correct rule and event definitions to the *rule* and *event manager*, respectively, that have to manage the *rule-* and *eventbase*. The *event detector* has to maintain the necessary data structure for the detection of the appropriate event. As soon as an event is detected, it is inserted in the so-called *event register*. Based on the (updated) information in the event register, the rule manager has to get activated and has to determine the rules to be executed. Afterwards, the *rule execution component* is involved for the condition evaluation and the action execution. In summary, the architecture of SAMOS augments the architecture of a (passive) ooDBMS (ObjectStore in our case) by new components like an analyzer, a rule and event manager, an event detector and a rule execution component. Since ObjectStore is a "black box" for our implementation, these components are located on top of it (at the expense of lower performance).

Obviously, the implementation of an efficient event detector is a crucial task for the efficiency of an active database system. Especially, the variety of event constructors makes the detection of composite events rather complex. In SAMOS, we introduced *Petri nets* for their modelling and detection. A Petri net consists of *states* (input and output) modelling event classes, and of *transitions*. As soon as an event has occurred, the appropriate input state is marked. According to the "switch" rules of Petri nets, one or more output states can be marked that correspond to the signalling of the appropriate composite event(s). For each constructor, we introduced a Petri net "pattern". The system manages a combination Petri net that includes all Petri nets for all defined composite events. An event can participate in more than one composition, while in the combination Petri net only one state for each event exists. An advantage of the use of Petri Nets is that composite events can be detected step by step, after each occurrence of a primitive event, and do not need the requested inspection of a large set of (primitive) events stored in the event register.

6 Conclusion

We gave an overview of the active object-oriented database system SAMOS. Its main contributions are the combination of active and object-oriented characteristics within one system and the support of comprehensive event definition facilities. In the longer term, we plan to provide (design) tools for active databases. In detail, a graphic editor, a debugger and tools analyzing interrelationships among various rules (e.g. to detect cycles) can help the user to overcome the complexity of applying an active database system.

References

[CM 91] S. Chakravarthy, D. Mishra. *An Event Specification Language (Snoop) For Active Databases and its Detection.* Technical Report September 91.

[DPG 91] O. Diaz, N. Patom, P. Gray. *Rule Management in Object-Oriented Databases: A Uniform Approach.* Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 91.

[GGD 91] S. Gatziu, A. Geppert, K.R. Dittrich. *Integrating Active Concepts into an Object-Oriented Database System.* Proc. of the 3. Intl. Workshop on Database Programming Languages, August 91.

[GJ 91] N.H. Gehami, H.V. Jagadish. *Ode as an Active Database: Constraints and Triggers.* Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 91.

[GJS 92] N.H. Gehami, H.V. Jagadisch, O. Schmuelli. *Event Specification in an Active Object-Oriented Database.* Proc. ACM SIGMOD, June 92.

Active Rules based on Object-Oriented Queries

Tore Risch
torri@ida.liu.se

Martin Sköld
marsk@ida.liu.se

Department of Computer and Information Science
Linköping University
Sweden

Abstract

We present a next generation object-oriented database with active properties by introducing rules into OSQL, an Object-Oriented Query Language. The rules are defined as Condition Action (CA) rules and can be parameterized, overloaded and generic. The condition part of a rule is defined as a declarative OSQL query and the action part as an OSQL procedure body. The action part is executed whenever the condition becomes true. The execution of rules is supported by a rule compiler that installs log screening filters and uses incremental evaluation of the condition part. The execution of the action part is done in a check phase, that can be done after any OSQL commands in a transaction, or at the end of the transaction. Rules are first-class objects in the database, which makes it possible to make queries over rules. We present some examples of rules in OSQL, some implementation issues, some expected results and some future work such as temporal queries and real-time support.

Key Words: Active Database, Object-Oriented Query Language, Object-Oriented Rules

1 Introduction

A powerful query language will be an essential part of the next generation Object-Oriented (OO) database systems. When active properties are introduced into these databases, the query language should be extended to support them.

The HiPac[4] project introduced *ECA rules* (Event-Condition-Action). The event specifies when a rule should be triggered. The condition is a query that is evaluated when the event occurs. The action is executed when the event occurs and the condition is satisfied.

In Ariel[6] the event is made optional, making it possible to specify *CA rules*, which use only the condition to specify *logical events* which trigger rules. Rules in OPS5[1] and monitors in [8] have similar semantics. In ECA rules the user has to specify all the relevant *physical events* in the event part. We believe that CA rules are more suitable for integration in a query language, since they are more declarative. CA rules make physical events implicit, just as a query language makes database navigation implicit.

We define active rules by extending the OO query language OSQL of Iris[5]. OSQL is based on functions for associating stored and derived attributes with objects. OSQL permits functional overloading on types, and types and functions are first-class objects. Likewise, rules are first-class objects in the database too[3]. This makes it possible, e.g., to make queries over rules. By implementing rules on top of OSQL, overloaded and generic rules are possible, i.e. rules that are parameterized and that can be instantiated for different types. We also utilize the optimizations performed by the OSQL compiler[7].

Each rule is defined by a pair <Condition,Action>, where the condition is a declarative OSQL query and where the action is an OSQL database procedure body. The rule language thus permits CA rules, where the action is executed (i.e. the rule is triggered) whenever the condition becomes true, similar to OPS5 and Ariel. Unlike those systems, the condition can refer to derived functions (which correspond to views). Data can be passed from the condition to the action of each rule by using shared query variables. By quantifying query variables set-oriented action execution is possible[11].

We are implementing our ideas in the research prototype AMOS[1] (Active Mediators Object System) by extending a Main-Memory version of Iris, WS-Iris[7]. OSQL queries are compiled into execution plans

[1]The AMOS project is supported by Nutek (The Swedish National Board for Industrial and Technical Development) and CENIIT (The Center for Industrial Information Technology), Linköping University

in an OO logical language. The system logs all side effect operations on the database. The rule compiler analyzes the execution plan for the condition of each rule. It then generates 'log screening filters' which check events that are added to the log. When a log event passes a log screening filter associated with a condition, it indicates that the event can cause the corresponding rule to fire. The screening of the log is often complemented with incremental evaluation[9, 10] of the condition.

Distributed execution of AMOS is being implemented too, and we plan to introduce temporal queries and real-time facilities as well.

2 Object-Oriented Query Rules

The syntax for rules conforms to that of OSQL functions as closely as possible:

create rule *rule-name param-spec* **as**
    when [*for-each-clause* | *predicate-expression*]
    do [once] *action*
where
for-each-clause ::=
    for each *variable-declaration-commalist* **where** *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction and negation. Rules are activated and deactivated by:
activate rule-name ([*parameter-value-commalist*])
deactivate rule-name ([*parameter-value-commalist*])

The semantics of a rule are as follows: If an event of the database changes the boolean value of the condition from *false* to *true*, then the rule is marked as *triggered*. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to logical events[2]. In the *check phase* (usually done before committing the transaction), the actions are executed of those rules that are marked as triggered. If an action is to be executed only once per activation, the rule is deactivated after the action has been executed. We can also introduce an *immediate* coupling mode[4] by instructing the system that the check phase is to be done immediately after each OSQL command.

Example 1:

The salary changes of employees and managers are to be monitored. We want to ensure that only managers can have their salaries reduced. First we define the employee and manager types and the respective income functions, where managers receive an additional bonus:

```
create type person;
create type employee subtype of person;
create type manager subtype of employee;
create function name(person) -> charstring as stored;
create function mgrbonus(manager) -> integer as stored;
create function income(employee) -> integer as stored;
create function income(manager m) -> integer i
    as select i where i = employee.income(m) + mgrbonus(m);
create employee(name,income) instances
    :joe ('Joe Smith',30000);
create manager(name,employee.income) instances
    :harold ('Harold Olsen',80000);
setmgrbonus(:harold) = 10000;
```

Then we define procedures for what to do when a salary is decreased:

```
create procedure compensate(employee e)
    as set income(e) = previous income(e); /* employee income cannot be decreased */
create procedure compensate(manager); /* dummy procedure, managers are not compensated */
```

[2]To support physical events the system should provide functions that change values whenever a physical event occurs and thus can be referenced in the condition of a rule.

The function **compensate** uses the system operator **previous** to fetch the value of a function at the previous checkpoint.

Finally we define the rule to detect decreasing salaries for all employees:

```
create rule no_decrease() as
    when for each employee e
    where income(e) < previous income(e)
    do compensate(e);
```

Activate the rule:

```
activate no_decrease();
```

If an employee that is not a manager gets his salary decreased, the rule will automatically set the salary back to the old value at check time:

```
set income(:joe) = 20000; /* => reset income(:joe) to 30000 at check time*/
```

Note: Since the rule is defined for all employees, and manager is a subtype of employee, the rule is overloaded for managers. (Because the functions **income** and the procedure **compensate** are overloaded). If a person of type manager gets a salary reduction, no action is taken. This is an example of a set-oriented rule. The action is executed for every binding of the universally quantified variable e for which the condition is true.

Example 2:

Rules can be parameterized and instantiated with different arguments. Take a rule that ensures that a specific employee has an income below a certain maximum income, and the transaction is rolled back if an employee receives an income above the threshold. This maximum income is fixed for all employees, but can vary for individual managers.

```
create function maxincome(employee) -> integer
    as select 50000;
create function maxincome(manager) -> integer as stored;
create rule exceeding_maxincome(employee e) as
    when income(e) > maxincome(e)
    do rollback;
```

Set the income limit for Harold:

```
set maxincome(:harold) = 120000;
```

Activate the rule for a particular employee Joe and manager Harold:

```
activate exceeding_maxincome(:joe);
activate exceeding_maxincome(:harold);
set income(:joe) = 75000; /* rollback at check time because 75000 > 50000 */
set maxincome(:harold) = 90000; /* rollback at check time because 90000 + 10000 > 90000 */
set mgrbonus(:harold) = 45000; /* rollback at check time because 80000 + 45000 > 120000 */
```

It is non-trivial to determine the physical events that trigger an OSQL rule with many interdependent and overloaded functions, such as the rule above. Hence we let the compiler determine this. This illustrates the convenience of CA rules.

Example 3:

Since types are first class objects, one can write generic rules that are instantiated for a specific object type:

```
create rule exceeding_maxincome(type t) as
    when for each employee e
    where typesof(e) = t and
          income(e) > maxincome(e)
    do rollback;
```

Activate the rule for all managers:

```
activate exceeding_maxincome(typenamed('manager'));
```

Since rules are first-class objects in the database, one can make queries over rules. For example, the system could provide a function that returns all active rules dependent on a certain object type or a function that takes a rule as argument and returns all the functions it depends on.

3 Expected results

The extension of OSQL with rules is expected to give a powerful language to express active properties in an object-oriented database. The overloading of rules provides a way to specify reusable rules that can be applied uniformly in different situations. One of the goals in the project is to investigate if CA rules can be implemented as efficiently as ECA rules. This involves efficient event detection as well as incremental evaluation of rule conditions. We will verify the applicability of OO rules by investigating how they can be used for various applications, e.g. in CIM.

4 Future work

Temporal rules can be introduced by having functions that vary over time and by time-stamping events in the database. The condition can then refer to the time when a certain event occurred. By introducing a timer event, a rule can be triggered at a certain time. These extensions do not support all the possible reasoning that can be made in an event algebra such as [2]. However, it allows for reasoning about whether one event happened before another or vice versa (by comparing time-stamps).

Introducing real-time in the database would require to take the cost of executing an action into account. Active database facilities are important for real-time applications that, e.g., monitors combinations of sensor data and perform actions whenever 'interesting' situations occur. The rule language will need to be complemented with timeliness constraints for rule conditions and actions.

References

[1] Brownston L., Farell R., Kant E., Martin A.: Programming Expert Systems in OPS5, *Addison-Wesley, Reading Mass.* 1986

[2] Chakravarthy S., Mishra D.: An Event Specification Language (Snoop) for Active Databases and its Detection, *UF-CIS Technical Report, TR-91-23*, sept. 1991

[3] Dayal U., Buchman A.P., McCarthy D.R.: Rules are objects too: A Knowledge Model for an Active, Object-Oriented Database System, *Proc. 2nd Intl. Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer 88

[4] Dayal U., McCarthy D., The architecture of an Active Database Management System, *ACM SIGMOD*, 1989, pp. 215-224

[5] Fishman D. et. al: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989

[6] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD*, 1992, pp. 49-58

[7] Litwin W., Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* Vol. 4, No. 6, December 1992

[8] Risch T.: Monitoring Database Objects, *VLDB conf. Amsterdam* 1989

[9] Rosenthal A., Chakravarthy S, Blaustein B., Blakely J.: Situation Monitoring for Active Databases, *the VLDB conf. Amsterdam*, 1989

[10] Paige R., Koenig S.: Finite Differencing of computable expressions, *ACM Trans. Prog. Lang. Syst.* 4.3 (July 1982) pp. 402-454

[11] Widom J., Finkelstein S.J.: Set-oriented production rules in relational database system *ACM SIGMOD Int. Conf. on Management of Data* pp. 259-270, Atlantic City, New Jersey 1990

On Developing Reactive Object-Oriented Databases

Mikael Berndtsson
University of Skövde, Sweden
spiff@his.se

Brian Lings
University of Exeter, UK
brian@uk.ac.exeter.dcs

1 Introduction

This paper outlines the ongoing joint work in Reactive Object Oriented Database Systems between the Departments of Computer Science in the University of Exeter(UK) and the University of Skövde(Sweden). The group is currently designing a monitoring system based on a reactive object oriented database with the objective of supporting efficient interaction between the active DBMS and applications (including intelligent systems).

Initial work has centred on a prototype reactive object-oriented system built on top of ONTOS, a commercial OODBMS which has C++ as its base language. The prototype is referred to as ACOOD (ACtive Object Oriented Database system) ([Ber91]). We briefly discuss this prototype, showing how reactive behaviour has been incorporated into a full OODBMS albeit with some restrictions. We also outline our plans for its future extension, and how these are motivated.

2 Reactive Behaviour in ACOOD

The underlying data model provides ACOOD with the features shown in Figure 1.

Following HiPAC([CBC+89]), rules in ACOOD are in the form of ECA rules (Event Condition Action), and their semantics are: when the event occurs, evaluate the condition, and if the condition is satisfied execute the action. Rules in ACOOD are represented as first class objects with the attributes shown in Figure 2.

As methods are used to send messages and to manipulate objects in an object-oriented environment, they correspond to primitive events. Also, an event can be generated before (PRE) or after (POST) the triggering event's operation. We have therefore introduced support for pre-triggers and post-triggers in ACOOD.

This still requires refinement in the sense of ensuring that events based on method invocation are treated correctly with respect to inheritance of methods ([DPG91]); i.e. that rules triggered by such events are not over-generalised. One approach is to treat method invocations as events ([BM91]), where event is identified as: (event=class+method). This acknowledges that the method gets its meaning from a class: the method name alone cannot be treated as a triggering event as the semantic interpretation of the method depends upon in which class the method was invoked. Events in ACOOD are detected and generated by implementing two invocations to the rule manager for each method that is going to generate a primitive event.

Rather than evaluating all pre-triggers and post-triggers for an event on the same database state, we have chosen to evaluate them on the state produced by the execution of a rule. By adopting this approach actions of executed rules will be able to affect the evaluation of conditions of other rules.

The rule manager is responsible for selecting the appropriate rules that are to be fired. The rule manager receives information about event type and transaction mode when an event is signalled. In order to select

Persistent objects	a facility provided by an Object class which is the superclass to all persistent classes;
Nested transactions	where each level of nesting is atomic on its own;
Shared transactions	where a transaction only commits if all of its cooperating processes commit.

Figure 1: ONTOS features

RuleName	a unique identifier
Event	an event for which the rule should be triggered, e.g. "aircraft is landing".
TransactionMode	determines when the rules should be evaluated and executed. That is, either it should be triggered and evaluated before (PRE) or after (POST) a triggering event's operation.
Status	indicates if the rule is enabled or disabled
Priority	rules are executed according to priority
Condition	an (ONTOS) SQL statement
Action	a name of the method that should be executed

Figure 2: Rule attributes in ACOOD

```
Item::putObject()                          Item::updateQty()
{                                          {
ruleManager("Save Item","PRE");            ruleManager("Update Qty","PRE");
user code.....                             user code.....
Object::putObject();                       Item::putObject();
user code.....                             user code.....
ruleManager("Save Item","POST");           ruleManager("Update Qty","POST");
}                                          }
```

Figure 3: Rule Manager calls in ACOOD

appropriate rules, the rule manager browses through the database for rules that match the given event and transaction mode and are enabled for firing. When it has found the appropriate rules, it executes them one after another according to their priority. Rules with equal priority can introduce an element of non-determinism.

The approach adopted in ACOOD will not give rise to situations in which actions from rule executions invalidate conditions of rules which have already been added to a contention set based on the state of the system at some eralier time. As the state is changed in response to rule actions any triggers whose conditions are satisfied by the resulting state can be activated. Rule firing in ACOOD is therefore very firmly an integral part of the current transaction, and conforms to a sequential model of activation.

The semantics of the C++ code in Figure 3 are: when someone updates the quantity, then the method Item::updateQty() is invoked. The following will happen as a response to the update: the rule manager will be informed that event "Update Qty" has occurred and that it should trigger appropriate pre-triggers defined on "Update Qty". After this a message is sent to another method, Item::putObject(), as a request to store the item. This request will first cause all appropriate pre-triggers defined on event "Save Item" to be executed. Then the item is stored by sending a message to the Object class, which is provided by ONTOS. This has in turn caused another database state which will cause all appropriate post-triggers defined on event "Save Item" to be executed. Finally, we return to method Item::updateQty(), where the last event in our example is generated: event "Update Qty" for post-triggers.

The underlying data model (ONTOS) supports ACOOD with basic mechanisms such as persistence and nested transaction. Hence, we were able to rapidly build a prototype of a reactive system and test our ideas, since we did not have to implement our own OODBMS. The method chosen, to effectively create a layer of reactive behaviour around the OODBMS, means that we have to create a subclass of any persistent class which we wish to inherit reactive behaviour. The disadvantages with this approach are that we sometimes have to "work around" the underlying data model in order to support our ideas, and we cannot generate primitive events for system defined classes.

The only alternative to this would be to use the programmatic interface available in ONTOS 2.2. Using this interface one can access all aspects of the C++ schema, and alter it dynamically. One can thereby add triggers to ONTOS objects' methods ([Boo]). The main limitation of this approach is that all applications wanting to take advantage of the feature would have to use the programmatic interface. ONTOS itself uses standard C++ for internal calls, so the situation within ONTOS is not different from the chosen method.

3 The future development of ACOOD

The notion of events has become more important and complex in recent proposals for reactive DBMS. When an event is signalled it must carry information with it, for use in condition evaluation in rules. Control of this, not least in order to cater for set-oriented rules, is imperative. It is also necessary to allow a broad interpretation of what constitutes an event in a system. Event specification languages for reactive object oriented proposals tend to support several different types of event such as database events, time related events and explicit events. Further, methods are used to send messages and to manipulate objects, they should therefore be useful for generating primitive events.

The need for supporting complex situations is satisfied in many proposals by introducing composite events ([CM91]). Advances in event specification languages for such systems have refined the notion of event into an event hierarchy. Briefly, events can be decomposed into i) primitive events such as database events, explicit events and temporal events and ii) composite events, where a composite event consists of a set of primitive events or composite events related by defined operators.

Support for complex events is difficult to accomplish by treating events as rule attributes. By representing events as first class objects we can construct complex events and achieve advantages similar to those obtained with rules as first class objects. For example, we may define an event type (say 'change_of_salary') associated with the method 'put_salary' of a class 'employee'. We may then define ECA rules with change_of_salary as the triggering event. Supposing there are further rules associated with an event type 'privileged_event' and we wish to express the fact that 'change_of_salary' is to be considered a triggering event for any such rules also. This can be expressed by defining an is-a hierarchy based on event types. Individual occurrences of an event will then be considered to be a triggering event of all inherited event types. The event type is inheriting all rules based on event types higher in the inheritance network. Multiple inheritance is important here: a change_of_salary event may also be defined to be an 'update_event'.

This could, of course, be achieved by defining composite events with disjunction for each event type which should act as a trigger. For example, change_of_salary could be listed as one alternative for a composite event 'privileged_event' and a composite 'update_event'. Maintenance would become difficult in these circumstances, forcing schema changes to be reflected in a number of updates rather than as a natural consequence of inheritance.

Given the above issues we are therefore currently looking at events as first class objects in order to introduce composite events systematically and efficiently, and thereby support a dynamic event specification language. The event specification language to be used will depend on the application domain under consideration, namely causal reasoning in diagnostic, object oriented expert systems (OBOES[NJ91]). The main concern of OBOES is the subsumption of object and class-based diagnostic reasoning within the paradigm of default reasoning: i.e. class-based (inheritance-based) representations and path-based reasoning are used for diagnosis. Events as first class objects fit well into the paradigm in question, offering direct support for the expression of causal relationships in the system itself.

ACOOD is a rapid prototype of an object-oriented reactive system built on top of a complete OODBMS. It is currently being enhanced with event specification, concentrating on event class hierarchies rather than the definition of composite events - which we acknowledge as a necessary but orthogonal concept.

We are also investigating the implementation of monitoring systems based on ACOOD, with the aim of efficiently supporting complex application domains including, as mentioned, diagnostic systems in which some causal reasoning is expressed through inheritance.

The design of the interface with such application systems will reflect decisions made concerning a suitable coupling between the chosen model of causation and an ECA rule set, in order to facilitate efficient support for object-oriented knowledge representation structures. The degree to which causal knowledge and causal chains can be represented in ECA rules is not known, but we expect that at least shallow relationships between component instances, and perhaps some relationships between component classes, will be expressible. The mapping of causal chains into ECA rules forms part of the project plan as it will guide the design of our event specification language and rule language.

Acknowledgement. The authors would like to thank the editor for a critical reading of an early draft of this paper.

References

[Ber91] M Berndtsson. ACOOD: an approach to an active object oriented dbms. Master's thesis, Dept of Computer Science, University of Skövde, 1991.

[BM91] C Beeri and T Milo. A model for active object-oriented databases. *Proceedings of the 17th International Conference on VLDB*, pages 337–349, 1991.

[Boo] P Boonstra. Private communications. ONTOS Inc, Burlington MA 01890, 1991-2.

[CBC+89] S Chakravarthy, B Blaustein, MJ Carey, U Dayal, D Goldhirsch, M Hsu, R Juahari, 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, 1989. Final Technical Report.

[CM91] S Chakravarthy and D Mishra. An event specification language (Snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, University of Florida, 1991.

[DPG91] O Diaz, N Paton, and P Gray. Rule management in object oriented databases: A uniform approach. In *Proceedings of the 17th International Conference on VLDB*, pages 317–326, 1991.

[NJ91] A Narayanan and Y Jin. Object-oriented representations, causal reasoning and expert systems. *Expert Systems: The International Journal of Knowledge Engineering*, 8(1):13–18, 1991.

Active Data/Knowledge Base Research At The University of Florida

S. Chakravarthy E. Hanson S. Y. W. Su
Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611

1 Introduction

A number of research projects addressing various issues on active data/knowledge bases are currently underway at the University of Florida. The **Sentinel** project focuses on the research, design, and implementation issues for an object-oriented active database system. The **Ariel** database system deals with rule condition testing optimization and reliable client-server interaction. The research emphasis in the third project is on the development of concepts and techniques of **Active Object-oriented Knowledge Base Management Systems**. In this short paper we highlight the accomplishments of each project and briefly outline ongoing and future work.

2 Sentinel

Sentinel is a (re)active Object-Oriented DBMS currently under development using Zeitgeist from Texas Instruments as the underlying platform. The emphasis of this project is on the systems and research issues in the context of an object-oriented active DBMS. This project extends the results obtained in HiPAC [C+89] to expressive event specification language and its implementation[CM91], distributed situation monitoring[CG91], seamless integration of ECA rules into a DBPL[ANW92], communication among application processes using active database paradigm. The long term objective is to use the resulting system: i) as a basis for a self-monitoring (or adaptive) DBMS, ii) to provide support for cooperative problem solving [CNTK90], and iii) to support multi-media active DBMS for scientific applications. Below, we highlight some of the results obtained so far.

2.1 Snoop

Snoop[CM91] is an expressive model independent event specification language. We have defined an event precisely and distinguished between an event and a condition. We proposed an event hierarchy consisting of primitive and composite (or complex) events. Primitive events are further classified into database, time, and external/abstract events. A number of event operators (disjunction, sequence, all, aperiodic and periodic with cumulative and non-cumulative variations) were defined along with the grammar for constructing complex events. We also showed how contingency plans can be translated into Snoop using the above operators and an aggregate count operator.

The notion of *parameter contexts* is used to compute parameters of complex events. Three contexts – recent, chronicle, and cumulative – were introduced to match the semantics required for widely understood classes of applications. Snoop is being implemented in the context of an OODBMS.

2.2 Seamless integration of ECA Rules into an OODBMS

In [ANW92] we classify objects into passive, reactive, and notifiable types. Passive objects are conventional C++ objects. Reactive objects generate primitive events when their methods are invoked. Notifiable objects are recipients of events generated by reactive objects. In contrast to other work in this category (notably, ADAM and Ode), we provide support for an *external monitoring viewpoint* which permits notifiable objects to dynamically subscribe to monitor changes to reactive objects and take appropriate actions. Both events and rules are supported as first class objects; composite events are implemented using a hierarchy thereby facilitating the detection of events using the structure provided by the class hierarchy. The proposed framework supports event and rule specification on any class including the rule class; immediate and deferred

coupling modes are currently supported. Only recent context for parameter computation is currently supported. This work, in summary, combines the strengths of the approaches taken in Ode and ADAM and further extends them in several significant ways.

2.3 Extended Relational Algebra (ERA)

One of the optimization techniques for the **Changes** operator proposed in HiPAC [C+89] was incremental evaluation. ERA[CG91] provides a mathematical basis for evaluating changes to arbitrary, non-aggregate expressions of relational algebra. Incremental versions for all relational operators (select, project, join, union, and difference) were developed and their correctness shown. Optimizing transformations using incremental versions of operators have been developed. The restriction on the chain rule developed in HiPAC was relaxed to obtain a generalized chain rule. Finally, alternative ways of optimizing expressions with the **Changes** operator were developed and analyzed.

2.4 AI and Database Integration

Our approach [CBM91] to AI and Database integration is to support production rule systems on a shared database. Towards this end, we developed a methodology for translating an OPS5 class of production rule applications into relations and triggers; execution equivalence (assuming the same conflict resolution strategy) is guaranteed. A correspondence was established between the production rule system concepts and active database concepts and an algorithm was developed to translate the source code of an OPS5 program to DDL for an active DBMS that supports simple forms of triggers (only disjunction of events and multiple triggers capability for a relation is required). The advantages of this approach to AI and database integration (as opposed to the approach of adding database functionality to a production rule system) are that all the database features such as persistence, access methods support, and optimization become readily available. Work on the concurrency control and recovery issues is underway to support multiple OPS5 applications simultaneously on a shared database.

In addition to the above, we are currently: i) implementing a nested transaction model for concurrent rule evaluation in Sentinel, ii) studying the performance of seamless integration of rules and events, and iii) addressing the translation of high-level active OODBMS specification to rules and objects in Sentinel.

3 Ariel

The Ariel project is focussed on extending database systems with an active database capability based on the production system model. Our goals in this project are to make the system reliable, efficient, and carefully integrated with traditional DBMS functions. Ariel currently processes rules of the following form:

define rule *rule-name* [**in** *ruleset-name*]
[**priority** *priority-val*]
[**on** *event*]
[**if** *condition*]
then *action*

The condition of an Ariel rule can refer to an event (such as insertion, deletion, or modification of data in a particular table), a condition (similar to the **where** clause of a query), or both. Transition conditions are supported using a special keyword **previous** that can appear before an expression of the form tupleVariable.field to get the old value of the field. An example of a rule with a transition condition is:

define rule ToyRaiseLimit
if emp.dno = dept.dno
    and dept.name = "Toy"
    and emp.salary > 1.1 * **previous** emp.salary
then replace emp (salary = 1.1 * **previous** emp.salary)

The effect of this rule is to limit the size of a raise a Toy department employee can get to ten percent. Previous work on Ariel has focussed on efficient rule condition testing, including

1. fast testing of new tuple values against a large number of single-relation selection conditions [HCKW90],
2. comparison of the Rete [FOR82] and TREAT [MIR87] algorithms for database rule condition testing [WH92], and
3. design of an integrated active database system based on a variation of the TREAT algorithm called A-TREAT that is optimized for the database environment [HAN92].

Our current work is examining:

1. use of optimization techniques to build a hybrid Rete-TREAT discrimination network tuned for a particular database, set of rules, and update pattern, and
2. reliable transmission of requests from rule actions to application programs.

Regarding the first item, we observe that a discrimination network, like a Rete network, has a structure very close to that of a relational database query plan. We are focusing on an approach that extends the traditional approach to *query* optimization by taking into consideration additional information about update frequency, which is a key variable in discrimination network optimization but is not relevant for query optimization.

Regarding the second item, active database systems now typically support a capability whereby the action of a rule in the DBMS can invoke some computation in an application program, or set of programs, running *outside* the DBMS, typically on client workstation, (e.g., put a pop-up window on my display if the price of my favorite stock looks cheap). However, current implementations of this suffer from a reliability problem, i.e., requests from the DBMS to applications may be lost (we call this the *lost dependent operation* (LDO) problem), or requests based on non-committed updates may be processed by applications (we dub this the *dirty dependent operation* (DDO) problem. We are striving to develop solutions to the LDO and DDO problems and use them in an extended version of Ariel that will support reliable interaction with application programs.

In the future, we plan to investigate implementation of production-system-style active database systems on parallel architectures. Our hope is that by working to make active database powerful, reliable, and efficient, they can be used as an effective tool in large-scale transaction processing systems.

4 Active Object-oriented Knowledge Base Management Systems

Another major research emphasis is the development of active KBMS concepts and techniques.

4.1 An Active KBMS Based on the OSAM* Model

The design of the KBMS [SR88, RS88] was based on the object-oriented paradigm. It features an object-oriented semantic association model OSAM* [SKL89] which provides strong support for semantic association types and knowledge rule specification and processing facilities, in addition to the traditional features of the OO paradigm. Five system-defined association types are provided to capture different semantic relationships existing among object classes and their object instances. Additional (user-defined) association types are introduced by modeling association types as object classes and their semantic properties by methods and knowledge rules defined in these classes [YSL91]. In addition to structural properties and methods, knowledge rules with triggers are defined in object classes as part of their behavioral properties. They capture semantic integrity, security, and other business and organizational constraints found in an application domain. Just like attributes and methods, rules can be inherited in a generalization hierarchy or lattice. Knowledge rules which are applicable to different sets of objects are naturally distributed among object classes and are used by the KBMS to enforce various constraints when objects are processed. A rule processor of the KBMS automatically triggers rules under different specified trigger conditions. Activation of a rule may trigger other rules. The above features make the KBMS an active KBMS. Documentations on the design and implementation of the knowledge rule specification language and the KBMS can be found in [ASL90, SA91, LAM89, SL90, LAM92].

4.2 A High-level Knowledge Base Programming Language K

Research is being carried out to develop a high-level knowledge base programming language called K [SS91]. K provides high-level modeling constructs to capture complex structural and behavioral semantics in terms of object classes, associations, methods and knowledge rules offered by the OSAM* model. It also contains set-oriented retrieval and manipulation constructs which use pattern-based specifications [ASL89] instead of the traditional attribute-based specification found in the existing relational query languages. The language also contains the traditional computational constructs such as If-Then-Else, do loops, case statements, etc., to make the language computationally complete. A compiler for a version of K and its supporting KBMS have been developed [SHY92, ARR92]. In K.1, a rule consists of the following 4 parts: 1) Trigger: Trigger condition and trigger time, 2) Condition: a complex object pattern, 3) Action: Action to be performed when the condition is satisfied, and 4) Otherwise: Action to be performed when the condition is not satisfied. K.1 provides the following active features in addition to the features of OSAM*.KBMS. K.1 supports user-defined operations as a Trigger. It also allows a sequence of data conditions to be specified in a guarded expression and be evaluated in the specified order so that a violation of a condition can terminate the evaluation of a rule. K.1 also supports cascade firing of the rules, i.e., an operation activated by a rule can trigger another set of rules which in turn can trigger some other rules. Rule execution in K.1 is efficient since rules are pre-compiled and equivalent C++ code is generated. This code will be executed when the rules are triggered. A KBMS provides query processing, rule processing capabilities, and persistent database support during the execution of K.1 programs.

4.3 Active OOKBMS in a Distributed and Parallel Environment

As the complexity of an active KBMS increases, efficiency becomes a major concern. Research is being carried out in Parallel and Distributed Active OOKBMS. In this project, a parallel query processor and parallel object manager have been implemented [GOR91, GOP92, BHE92]. It uses wavefront algorithms [TSL90, SCL91]. A parallel rule processor for object processing is being built on top of the parallel query processor. Based on the fact that the rules are often semantically related and they have inter-dependencies, semantic rule structures are used instead of isolated rules. Rule structures capture the control flow semantics among rule objects. A parallel rule processing algorithm that executes rule structures without violating the control flow and database consistency is being implemented. This algorithm achieves inter-structure parallelism, intra-structure parallelism and intra-rule parallelism. A nested locking protocol is used to maintain consistency during rule execution.

References

[ANW92] E. Anwar. Supporting complex events and rules in an oodbms: A seamless approach. Master's thesis, CIS Department, University of Florida, November 1992.

[ARR92] Arroyo, J. A. The Design and Implementation of K.1: A Third Generation Database Programming Language, Master's Thesis, CIS Department, University of Florida, 1992.

[ASL89] Alashqur, A. M., Su, S. Y. W., and Lam, H. OQL: A Query Language for Manipulating Object-oriented Databases, Proc. of VLDB Conference, 1989, pp. 433-442.

[ASL90] Alashqur, A. M., Su, S. Y. W., and Lam, H. A Rule-based Language for Deductive Object-oriented databases, Proc. of the 6th Int'l Conf. on Data Engineering, 1990, pp. 58-67.

[BHE92] Bhethanabotla, Shyam S. Design and Implementation of a Distributed Object Manager, Master of Engineering Thesis, Electrical Engineering, University of Florida, 1992.

[C+89] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, XAIT, Cambridge, MA, Aug. 1989.

[CBM91] S. Chakravarthy and R. Blanco-Mora. Supporting very large production systems using active dbms abstraction. Technical Report UF-CIS TR-91-25, CIS Department, University of Florida, Sep. 1991.

[CG91] S. Chakravarthy and S. Garg. Extended relational algebra (era): for optimizing situations in active databases. Technical Report UF-CIS TR-91-24, CIS Department, University of Florida, Nov. 1991.

[CM91] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, CIS Department, University of Florida, Sep. 1991.

[CNTK90] S. Chakravarthy, S. B. Navathe, A. Tanaka, and S. Karlapalem. The cooperative problem solving approach: A database-centered approach. In S. M. Deen, editor, *Cooperative Knowledge Based Systems*, pages 30–52. Springer-Verlag, 1990.

[FOR82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GOP92] Gopalan, A. Transaction Management and Recovery in a Distributed Object-oriented Database System, Master of Science Thesis, Electrical Engineering, University of Florida, 1992.

[GOR91] Gorur, Arun S. Implementation of a Query Processor on a Multiprocessor Network, Master of Science Thesis, Electrical Engineering Department, University of Florida, 1991.

[HAN92] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[HCKW90] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, May 1990.

[LSR92] Lam, H., Su, S. Y. W., Ruhela, V., Pant, S., Ju, S. M., Sharma, M., and Prasad, N. GTOOLS: An Active GUI Toolset for an Object-oriented KBMS, Int'l Journal of Computer System Sciences and Engineering, Vol 7, No. 2, 1992, pp. 69-85.

[MIR87] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of AAAI 87 Conference on Artificial Intelligence*, pages 42–47, August 1987.

[RS88] Raschid, L. and Su, S.Y.W. A Transaction-Oriented Mechanism to Control Processing in a Knowledge Base Management System, Proceedings of the Second Int'l Conf. on Expert Database Systems, 1988, pp. 353-373.

[SA91] Su, S. Y. W., and Alashqur, A. M. A Pattern-based constraint Specification Language for Object-oriented Databases, Proc. of IEEE COMPCON 91, 1991.

[SCL91] Su, Stanley Y.W., Chen, Yaw-Huei, and Lam, Herman. Multiple Wavefront Algorithms for Pattern-based Processing of Object-oriented Databases, Proc. of the Int'l Conf. on PDIS, 1991, pp. 46-55.

[SHY92] Shyy, Y. M. The Design and Implementation of a Knowledge Base Programming Language for Evolutionary Prototyping of Software Systems, Ph.D. Dissertation, CIS Department, University of Florida, May 1992.

[SKL89] Su, S.Y.W., Krishnamurthy, V., Lam, H. An Object-Oriented Semantic Association Model (OSAM*)for Modeling CAD/CAM Databases, Chapter 17 in Artificial Intelligence: Manufacturing Theory and Practice, Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989, pp. 463-494.

[SL90] Su, S. Y. W. and Lam, H. Object-oriented Knowledge Base Management Technology for Improving Productivity and Competitiveness in Manufacturing, Proc. of the 16th NSF Grantees Conference on Design and Manufacturing Systems Research, AZ., Jan. 8-12, 1990, pp. 161-167.

[SLH92] Su, S. Y. W., Lam, H. Hardwick, M., Spooner, D., Goldschmidt, A. and Chida, J. An Integrated Object-oriented Knowledge Base Management System OSAM*.KBMS/ROSE for Supporting Design and Manufacturing, Proc. of the IEEE Second Int. Conf. on Systems Integration, 1992, pp. 152-161.

[SR85] Su, S. Y. W. and Raschid, L. Incorporating Knowledge Rules in a Semantic Data Model: An Approach to Integrated Knowledge Management, A. I. Applications conf., Miami, Dec. 1985.

[SS91] Shyy, Yuh-Ming and Su, Stanley Y.W. K: A High-level Knowledge Base Programming Language for Advanced Database Applications in Proc. of ACM SIGMOD 1991, Denver, CO., May 29-31, 1991, pp. 338-347.

[TSL90] Thakore, A. K., Su, S. Y. W., Lam, H. and Shea, D. G. Asynchronous Parallel Processing of Object Bases using Multiple Wavefronts, Proc. of the Int. Conf. on Parallel Processing. Chicago, IL., Aug. 1990, pp. 127-135.

[WH92] Yu-wang Wang and Eric N. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proc. IEEE Data Eng. Conf.*, February 1992.

[YSL91] Yaseen R., Su, Stanley Y.W. and Lam, Herman. An Extensible Kernel Object Management System, in Proc. of OOPSLA '91, 1991, pp. 247-263.

A DOOD RANCH at ASU: Integrating Active, Deductive and Object-Oriented Databases

Suzanne W. Dietrich, Susan D. Urban*, John V. Harrison†, Anton P. Karadimce
Department of Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-5406
{dietrich | urban}@asuvax.eas.asu.edu

1 Introduction

Over the past ten years, the use of databases in non-traditional database applications has helped to shape the requirements of future database systems. Among those requirements, two features are outstanding: object-orientation and rule processing, where rule processing includes deductive *and* active database capabilities. A clean integration of object-oriented, active, and deductive database concepts, however, has not yet been achieved. The focus of this research is on the integration of these three areas within a project that we appropriately (here in Arizona) refer to as *A DOOD RANCH* (Active, Deductive, Object-Oriented Databases - Relating Action, Negation, Constraints and Horn rules).

The research has so far developed as two separate projects, with our continuing efforts focused on the synergism of the research into a joint project. The first project considers updates, integrity maintenance and rule analysis in a DOOD environment. The second project focuses on efficient condition monitoring in an active deductive database, which must be cognizant of the efficient and complete evaluation techniques of recursive rules. This paper describes each project separately and then concludes with discussions of the research directions for the integration of these projects into an active DOOD environment.

2 Updates, Integrity Maintenance, and Rule Analysis in a DOOD Environment

Recent work on the integration of deductive and object-oriented concepts can be found in [3]. Although current DOOD proposals provide a declarative query interface for object-oriented data, they typically lack the capability for ad-hoc declarative specification of updates to extensional data. This is due to the complexity of including essential features such as integrity checking and integrity maintenance into the declarative update framework. Our approach to the update issue is based on a DOOD model that supports the declarative, rule-based expression of methods, constraints, and derived data (including recursive rules) [8]. The model also provides declarative *update rules* (URs) that are supported by *active integrity methods* (IMs). A UR has a format similar to a deductive rule: the rule body specifies a set of qualifying objects, and the rule head specifies the elementary updates to be applied to each qualifying object. As an example, the following UR will delete the *advisor* property value of *student* objects that have a gpa less than 2.0:

S[advisor: -A] ← Student:S[gpa:G], G < 2.0.

An IM has a format of an event-condition-action (ECA) rule, where the event is an elementary update that triggers the IM and the action part describes the updates to be performed on objects satisfying the condition part. IMs are derived in a systematic, semi-automatic fashion starting from a set of declaratively-stated database constraints, using an approach known as constraint analysis [10]. Classic update methods are then emulated by a controlled, active and user-transparent interaction between the predefined set of elementary updates and the set of IMs designed to maintain database consistency upon violation of structural or general integrity constraints. The execution model to support our update framework is based on an integration of nested transaction and active database concepts [2].

As a simple example, if there is a 1:N *advisor* -- *advisees* inverse property constraint between the classes of *students* and *instructors*, then the above update that deletes a *student* object's *advisor* value will automatically trigger the following IM to remove the *student* object from the *advisees* property of the affected *instructor* object:

<event> Student:S[advisor: -A] :: <condition> Instructor:I[advisees:Y], S ∈ Y → <action> I:[advisees: { -S }].

*This research was partially supported by NSF Grant No. IRI-9109195
†Current address: Department of Computer Science, University of Queensland, Brisbane, Qld 4072 Australia

In general, constraints can involve more complex relationships between objects. Using the above approach of associating integrity maintenance rules with constraints, end-users of the update language are relieved as much as possible from the responsibilities of consistency consideration; update requests are simpler and less error-prone. Furthermore, if the set of IMs is complete and anomaly-free (see below), then arbitrary, ad-hoc update requests can be made without jeopardizing database consistency. More complex updates, e.g. update transactions as sequences of URs, are readily supported as well [8].

IMs essentially execute as production rules, where the execution of one IM may trigger the execution of other IMs. This *cascaded triggering* of IMs raises the problem of potential *anomalies* -- in particular, the cascaded triggering may not terminate, or the final database state may depend on the order of execution of IMs. Verification of non-anomalous behavior of IMs is one of our major research directions.

In [1], several general results are stated pertaining to anomalies in general production rule systems. Using these results as a starting point, we are developing automated tools for more detailed rule analysis in our specific context of a DOOD with IMs. Our approach relies on the use of OODB schema knowledge and the condition parts of URs and IMs. Using a logic encoding of the OODB schema and a satisfiability algorithm to track the impact of conditions to the progress of IM execution, we achieve a more precise rule analysis in the sense that cases labelled as "potentially anomalous" (according to the approach in [1]) are reduced in number. In other words, some potentially anomalous cases are reclassified as non-anomalous, while for other cases, specific database instances under which an anomaly can occur are identified.

We are investigating the refinement of our automated rule analysis process for special syntactic classes of IMs. For instance, it is known that stratified sets of production rules have the termination property, so that no rule analysis beyond simple syntax checking is necessary. On the other hand, the existence of a stable model (extension) for a set of integrity methods cannot generally be checked syntactically. Certain restrictions on the format of IMs, such as the type of allowed updates (e.g., updates of properties or objects, updates involving insert or delete operations, updates that involve no generation of new values, updates that do or do not involve set-valued properties) lead to more precise automated rule analysis. Although the general problem of detecting anomalous rule behavior is an undecidable problem, our preliminary results indicate that useful tools for helping database designers understand active rule behavior can be developed by analyzing the semantics of active rules [7].

3 Condition Monitoring in an Active Deductive Database

The condition monitor of an active database system is responsible for detecting when *conditions*, such as alerters, triggers and integrity constraints, are satisfied. Researchers developing active database systems have recognized the necessity for an efficient condition monitor and have proposed several approaches for condition monitoring in relational databases or relational databases extended with objects, limiting conditions to SPJ (select, project, join) expressions defined over extensional relations. Deductive databases, however, extend relational databases with inherent support for rules that include the power of union, difference (stratified negation) and *recursion*. Deductive databases provide a logic-based language, called *Datalog*, that can be used to declaratively express complex conditions, providing a uniform language for the expression of data, views, queries, and conditions. One key problem that must be addressed, however, is how to efficiently monitor conditions expressed using the Datalog language, which allows conditions expressed over intensional relations that may be defined in terms of select, project, join, union, stratified negation and *recursion*.

A naive condition monitor could detect condition satisfaction by evaluating each condition in both the old and new database state with respect to a set of changes to the extensional database. The result of each evaluation would then be compared to identify changes that may indicate condition satisfaction. With this naive approach, each condition can be considered as a view definition, where the view is materialized in each database state. Comparing the materialized views would indicate the differences caused by the updates and may indicate condition satisfaction.

An *incremental* approach that obtains the differences without incurring the high cost of materialization is preferred. Variations of this incremental approach can be found in many active database systems, starting with HiPAC [9]. An application of the HiPAC incremental change theory to the problem of monitoring Datalog conditions resulted in an approach that could support conditions defined using safe, nonrecursive Datalog without negation [5]. The incremental change theory introduced a pragmatic concern of requiring the materialization of partial joins for the incremental join operator. This consideration is avoided by an update propagation approach to efficient condition monitoring in an active deductive database defined using safe, recursive Datalog with stratified negation [4]. Note that this incremental condition monitoring strategy naturally provides an update propagation approach to the maintenance of materialized views in a deductive database [6].

The update propagation algorithm is known as PF, which refers to its two-phased approach to evaluation: a propagation phase and a filtration phase. The propagation phase propagates the changes to the extensional relations up through the rules and identifies potential changes to the intensional relations. Potential changes are filtered to identify actual changes. For example, a potential addition represents a derivation of a tuple *t*. If *t* is provable in the database state before the updates, then the potential addition is filtered and is *not* reflected as an actual change to the database. Similarly, a potential removal represents the deletion of a derivation for a tuple *t* and if *t* is still provable in the database state after the updates, then the filter phase does not identify the potential removal as an actual removal.

The PF algorithm is built on top of your favorite *efficient and complete* recursive query evaluation strategy. The PF algorithm cleverly calls for the evaluation of subqueries (using bindings from the updates) in each database state to determine when an actual removal or addition has occurred. (Note that modifications are represented by a removal and an addition.) In addition to taking advantage of the wealth of deductive database (recursive) query evaluation strategies, query optimization techniques developed for deductive databases have been incorporated into the prototype of the PF algorithm [4].

This work also includes additional issues on condition representation and reasoning. For example, the difference between the evaluation and reasoning of event-oriented versus state-oriented conditions are explored. Consider a condition expressed in Datalog that monitors an inventory shortage for each part in the database:

inventory_shortage(Part) ← stock_level(Part,InStock,MinStock), InStock < MinStock.

The inventory_shortage condition can be viewed as either event-oriented or state-oriented. We call an event-oriented condition *resatisfiable* and a state-oriented condition *nonresatisfiable*. If inventory_shortage is resatisfiable (event-oriented), then once an inventory shortage for a particular part is determined, additional shortages for that part continue to be propagated. The resatisfiable inventory_shortage condition is satisfied each time a change occurs in the stock level that does not rectify the shortage. If the above condition is nonresatisfiable (state-oriented), then once an inventory shortage for a particular part is determined, additional shortages for that part are not propagated. Only after the inventory shortage for that part is rectified, either by increasing the stock level or decreasing the minimum stock requirements, will a subsequent inventory shortage be recognized for that part. Note that using deactivation of rules does *not*, in general, provide the semantics of a nonresatisfiable condition. The above rule monitors inventory shortages for *all* parts in the database and thus, deactivation would inhibit the monitoring of the condition for other parts.

Other condition representation and reasoning issues include the specification of conditions that may refer to the changes computed for a relation or to the instance of a relation with respect to either the old or new database state. Changes to a relation, as determined by the PF algorithm, are denoted by the prefix Δ and the type of change is indicated by a subscript of *additions* or *removals*. The subscript *NEW* or *OLD* is used to refer to a relation with respect to a specific database state. As an example, the following condition monitors the situation when a new manager is added to the database and the manager is not an employee in the new database instance:

new_manager_not_emp(MGR) ← Δ $\text{manages}_{additions}$(MGR, DEPT), not(employee_{NEW}(MGR, _, ..., _)).

4 Research Directions

The goal of *A DOOD RANCH* is to integrate the above research efforts into the development of an active database environment that uses a deductive, object-oriented model as a formal basis for the declarative specification and efficient execution of active rule processing. In particular, we are investigating the extension of the DOOD model of Section 2 to provide a more complete framework for support of active and deductive database applications, developing a rule language that supports alerters and triggers in addition to IMs. We are also investigating efficient evaluation strategies for the DOOD rule language, deriving techniques from deductive database optimizations and enhancing those techniques with object-oriented considerations. Existing bottom-up and top-down evaluation techniques must be re-evaluated in the context of object-oriented optimization issues, such as object identity, set-valued attributes, and the use of path expressions in the rule language.

We are also extending and formalizing the rule execution model to create an active processing environment that supports deductive rules and active database rules in general. The research is specifically focused on the integration of a nested transaction processing model for active rules with the condition monitoring technique described in Section 3 above. The condition monitoring algorithm must be re-examined in the context of an active DOOD environment.

This investigation may provide additional tools for optimization. For example, the introduction of object identity with the use of object generating functions may provide useful techniques for detecting changes. The final result of our efforts in this aspect of the research will produce a formal execution model for the use of IMs together with general active database rules, supported by efficient techniques for detecting changes in conditions involving extensional and intensional data.

The final aspect of our work involves the development of a supportive environment for the design, testing, and analysis of active database applications. This aspect of the research will redefine, consolidate, and extend our current research base on the analysis of constraints and the detection of anomalous rule behavior to create an environment that can assist in the design and use of rules in the active DOOD framework. The development of such an environment will be important to the acceptance of active databases as a viable technology.

The areas of deductive, object-oriented, and active databases have primarily developed as separate research areas. Although significant results have been achieved in each area individually, there are weaknesses in each type of database system that can only be overcome by combining the strengths of each. The results of our integrated research project will ultimately provide an active, deductive, object-oriented database that efficiently and correctly processes queries, constraints, and active rules that involve extensional and intensional data. The successful integration of these areas has the potential to provide the kind of powerful database processing environment that will be required by database applications of the future.

References

[1] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings of the 1992 ACM SIGMOD Conference*, 1992, pp. 59-68.

[2] C. Beeri and T. Milo. A Model for Active Object-Oriented Database. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991, pp. 337-349.

[3] C. Delobel, M. Kifer, Y. Masunaga (editors). *Deductive and Object-Oriented Databases. Lecture Notes in Computer Science*, vol. 566, Springer-Verlag, 1991.

[4] J. Harrison. *Condition Monitoring in an Active Deductive Database*. Ph.D. Dissertation, Dept. of Computer Science and Engineering, Arizona State University, August 1992.

[5] J. Harrison and S. W. Dietrich. Towards an Incremental Condition Evaluation Strategy for Active Deductive Databases. In *Research and Practical Issues in Databases: Proceedings of the 3rd Australian Database Conference*, World Scientific, February 1992, pp. 81-95.

[6] J. Harrison and S. W. Dietrich. The Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. *Proc. of the Deductive Database Workshop* in conjunction with the Joint International Conference and Symposium on Logic Programming, Washington, D.C., November 1992, pp. 56-65.

[7] A. Karadimce. *An Investigation of Anomalous Rule Behavior in Active, Object-Oriented Database Systems.* Ph.D. Dissertation, Dept. of Computer Science and Engineering, Arizona State University, to be completed Spring 1993.

[8] A. Karadimce and S. Urban. A Framework for Declarative Updates and Constraint Maintenance in Object-Oriented Databases. To appear in *Proc. of the Ninth Int. Conf. on Data Eng.*, Vienna, April 1993.

[9] A. Rosenthal, S. Chakravarthy, B. Blaustein and J. Blakeley. Situation Monitoring for Active Databases. *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989, pp. 455-464.

[10] S. D. Urban and L. M. L. Delcambre. Constraint Analysis: Identifying Design Alternatives for Operations on Complex Objects. *IEEE TKDE*, vol. 2, no. 4, December 1990, pp. 391-400.

REACH: a REal-time, ACtive and Heterogeneous mediator system

A. P. Buchmann, H. Branding, T. Kudrass, J. Zimmermann

Technische Hochschule Darmstadt
Frankfurter Str. 69a, 6100 Darmstadt, Germany

1. Introduction

The goal of the REACH project is to integrate heterogeneous repositories around an active-object mediator system to support complex applications. The active core of REACH is REACT (REal-time ACTive object-system). To achieve the goals of REACH we require full flexibility and expressive power in the REACT rule system to express and enforce complex data dependencies across heterogeneous systems, to provide access control, a flexible transaction mechanism, and timing constraints. At the same time, large volumes of data must be managed within REACT. Rather than reimplementing access control, transaction management, integrity management and other functions with ad-hoc mechanisms, the same rule-based mechanism should be used. The internal use of the ECA-rules will stress the performance of the ECA-rule manager and requires judicious balance between expressive power of the rule language, the set of events and the operations of the event algebra, and the flexibility of the execution model. In a first attempt to balance these requirements we are expressing a complex transaction model with closed and open nesting, a complete access control mechanism, and complex consistency constraints through a common rule structure. In this paper we discuss the rules for the three domains and outline the structure of the event class hierarchy and timing constraints. It is not the goal of this project to develop yet another object model. Instead, we are defining the functionality of the ECA rules and are investigating what restrictions are imposed on them by the characteristics of various object models. Features of the rule system are prototyped as user-level objects in the O2 object model and the ObjectStore class library as two representatives of basically different object managers. We are also investigating necessary tradeoffs and effects of the rule mechanism on system architecture for later implementation at lower system levels.

2. Rules

Rules are typed and can be organized in a rule type-hierarchy. We distinguish among access control rules, consistency rules and transaction rules. Each of these rule-types can have subtypes, for example, access control rules may be either content-independent or content-dependent. This Section describes the three main rule classes that we consider.

2.1 Access Control Rules

The main motivation for using ECA-rules for authorization control is the possibility to respond in a more flexible manner to attempted security violations. This is required in multi-level security (MLS), where the simple denial of service may lead to inferences about the content of the database. It is also an issue when dealing with privacy protection laws for medical records. ECA-rules permit specifying security policies by specifying a variety of responses in the action part of the rule. These may range from random delays before giving an answer, to false stories and noisy data. ECA-rules can further be defined to specify rules on the log with conditions designed to detect suspicious access patterns.

Access control rules can be content-independent or content-dependent. In the former case no access to the actual object is necessary to decide whether a subject is authorized to manipulate an object or not. In the content-dependent case a predicate must be evaluated on one or more attributes of the object, thus requiring a read access before a decision can be made. Content-dependent access control is often handled similar to integrity constraints through query modification techniques in conventional database systems and is usually offered in addition to content-independent access control. MLS typically assumes content-independent access control for mandatory security while content-dependent access control may be used for compartmentalization. The implication for ECA-rule evaluation is that some access-control rules must be triggered before execution of the DML statement, some after its execution. This may be handled either by explicitly differentiating the triggering events and introducing BEFORE and AFTER constructors, or by embedding the semantics in the transaction model as discussed in Section 2.3., where the transaction model is defined in terms of ECA-rules and based on the type of rule triggered, a different transaction graph can be defined.

Access control rules exhibit a peculiar behavior compared to the integrity rules typically modelled by ECA-rules. Independently of the outcome of the condition evaluation a write action to the audit trail may have to be triggered. This leads to the need for triggering either two rules successively or to define an if-then-else format for the rules. This requirement has implications on the execution model of the rules and would require the failure of a condition evaluation to be established as a primitive event in the event hierarchy. We are currently experimenting with both options to determine the semantic implications of an if-then-else format and the performance implications of both approaches.

2.2 Consistency Rules

ECA-rules have been widely proposed for consistency management, therefore we will only touch on some distinctive aspect of consistency rules in REACH. One of the goals of the project is to define new consistency notions among heterogeneous systems. Some possible relaxations of the traditional consistency notion have been described in [SHET91] and can be summarized as relaxation of the time at which consistency must be achieved, e.g., at midnight or by begin of the next working-day, and the extent over which consistency must be achieved, e.g., certain partitions in federated databases. Enforcement of these relaxed consistency notions requires, among other things, the inclusion of time as a constraint in the rule definition. Section 4 addresses some of these issues.

Complex consistency constraints in applications, such as CAD, cannot always be specified by simple predicates. They may require the execution of a program to test consistency. This has an implication on the simple query format generally assumed for the condition part of a rule. To maintain that format it is necessary to divide complex constraints into two rules, one event-action rule that does the consistency test as the action part of the first rule signals a completion event, and a second rule that executes the repair action. The fact that these complex consistency checks are often executed on external nodes as separate processes requires detached consistency evaluation in addition to immediate and deferred couplings for consistency enforcement. Detached consistency evaluation, however, carries an additional price in that data quality must be expressible in the system.

2.3 Transaction Management Rules

Transaction management rules describe the execution structure and correctness criteria of the underlying transaction model, an extension to the DOM transaction model [BUCH91] with some temporal enhancements. The formal description of the model using the ACTA metamodel [CHRY91] is the basis for the definition of the rules on transactions. Transactions are modelled as objects that belong to different classes, such as multitransactions (open nested), top transactions (closed nested) or compensating transactions. The basic transaction management operations BOT, EOT, COMMIT and ABORT serve as events and are realized as method calls. We distinguish between EOT and COMMIT, since triggered transactions that are executed in deferred mode must be inserted between the last operation of the triggering transaction and the COMMIT. To avoid confusion with the interchangeable usage of EOT and COMMIT in commercial systems, we define an event COMPLETE and another COMMIT. The condition part of the ECA-rules contains checks on the transaction state and the action part implements the actual execution.

A transaction tree is derived from the user-specified transactions and the firing of rules. The most flexible approach is a fully interpretative approach in which the system creates dynamically new transaction objects at run-time. To improve performance it is convenient to compile the transaction tree. While this is not possible for the most general case, we are exploring how far information about the transaction classes, the objects that are accessed and the types of rules that are potentially triggered can be exploited in compiling the transaction trees.

3. The event hierarchy

Events are objects and event types can be organized in an event-type hierarchy. Typing the events also adds performance, since event handling can be specialized and the rule-sets that need to respond to an event of a given type are smaller. Events are any database event, including transaction-related events, arbitrary method call events, and temporal events. User-generated external events can be modelled as messages to the appropriate event-class or method calls. Complex events are composed from simple events. The event hierarchy and composition algebra are similar to the one proposed for SNOOP [CHAK91]. However, since we are interested in determining the effect of the

various models on the rule system, our event hierarchy at present is not minimal. For example, in a function-based object model, a function evaluation event is sufficient, while the model used in O2 requires different treatment for attributes and method invocations, particularly when dealing with authorization rules. Similar effects are noticed when dealing with object models that have an explicit delete vs. those that implement persistence through reachability. Fig. 1 shows the event hierarchy.

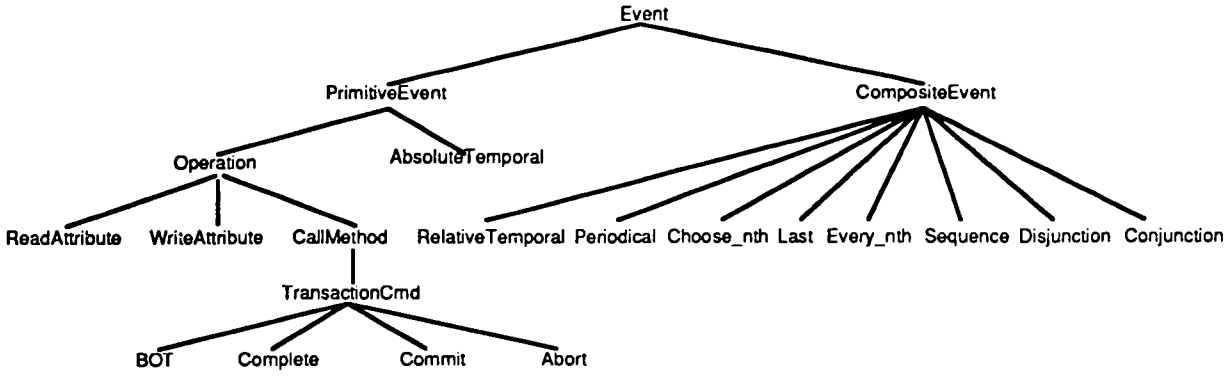

Figure 1: REACT Event hierarchy

Event composition is performed by the various event classes. Objects are activated by associating rules with them. If no rule is active for a given object, no overhead is payed. The association of rules to objects is done by extracting the primitive events that need to be monitored from the rule definition and inserting the corresponding identifier in a structure of the active object. Only tracking of the simple events needs to be performed at the object level. Any event composition is performed by the event handlers. A rule is fired as soon as the event is complete. For this purpose each event object stores the identifiers of the rules that are fired by it.

Temporal events are important in the applications we deal with. Both absolute and relative temporal events are possible. Relative temporal events may be relative to any other event, e.g., a transaction commit or the completion of a machining operation. This is important to model such consistency notions as "twelve hours after an update global consistency must be restored". While absolute temporal events are easy to handle, relative events must be converted. For this purpose, each event has a timestamp. A relative event is then defined with respect to the timestamp of the reference event and treated as an absolute event. The temporal event handler manages a sorted queue and dispenses the appropriate temporal events to the composite event handlers. The temporal event handler also fires the rules with simple temporal events.

Events are considered to be points in time. However, operations have finite length. It is therefore necessary to distinguish between two points in time when defining events. Before and after an operation is executed. This distinction is important for access control rules and is essential when dealing with temporal constraints (Section 4). In the case of content-independent access control rules we specify that the rule should be fired before the action is executed, in the case of consistency checks it must be executed after evaluation. BEFORE and AFTER are constructors that are used in conjunction with the events. The meaning of BEFORE changes when dealing with a temporal event. Many active database projects describe that certain rules must be executed before a given time. In this formulation the triggering point is not defined and cannot be determined in existing database systems by subtraction of the execution time from the deadline, since execution times are not known. Precise specification of the semantics of timing constraints is required.

4. Timing Constraints

Handling of temporal events and timing constraints has many commonalities. Therefore, we are developing an interval algebra for timing constraints in an attempt to standardize the semantics of temporal events and temporal constraints and avoid future conflicts. Full discussion of the modelling of temporal aspects is beyond the scope of this overview. Therefore, we sketch only one topic, the modelling of timing constraints for rule execution.

The temporal behavior of a rule is specified by temporal events and timing constraints. An event is always a point in time. An event defines the triggering-time of a rule and another event is generated upon completion. A timing constraint describes the interval in which the execution is permitted. Intervals are specified by complex expressions of the point algebra using the operators AFTER, BEFORE, and UNTIL. Often, a complex event describes the situation that leads to rule invocation. A subexpression of a complex triggering event describes a situation that occurs before a rule has been triggered. Timing constraints should be able to refer to parts of a complex event. An example that illustrates this is the timely propagation of data which have a constrained validity interval. Two measurements are made, the data are analyzed and the result is propagated to other sites. The timing constraint must refer to the different points of measurement to describe the temporal validity of the analysis result. The derived data will be valid as long as the measurements are valid. The timing constraint should enforce the propagation of valid analysis results.

A value function can be specified on the constraint intervals. Outside its domain it is assumed to be zero. Since events may be part of a constraint-definition and timestamps of events are known after these occur, binding of temporal constraint expressions takes place at run-time, thereby establishing the domain of the value function. Negative values are not permitted. Figure 2 gives an idea of the formalism. A specification of semantics and an extension of the algebra to handle disjunctions of intervals are omitted here.

```
ON      measure1 AND measure2
IF      ok(sensor1) AND ok(sensor2)
DO      do analysis; propagate result;
TIME CONSTRAINT (AFTER trig_event, BEFORE min (ts(measure1) + 6 s, ts(measure2) + 10 s),
                        value(t) = a*(ts (trig_event) + min (ts(measure1) + 6 s, ts(measure2) + 10 s) - t)/t
```

Fig. 2: An example of an ECA-rule with timing constraint

5. Conclusions and Current Status

The REACH project builds on previous research done in the context of the HiPAC [CHAK89, DAYA88] and DOM [MANO92, BUCH90, BUCH91] projects. In a first approach to defining rule semantics and testing them we are implementing an active layer on top of existing object managers. We are comparing the strengths and weaknesses of various object models for supporting active capabilities. The long-term goal is to implement a complete system that also supports the temporal aspects and allows us to implement the performance-critical functions at low system levels. For this purpose we are starting the implementation of a time-constrained DBMS on top of the Chorus [ROZI90] operating system.

References:

[BUCH90] Buchmann, A.; "Modelling Heterogeneous Systems as a Space of Active Objects", Proc.4th Intl. Workshop on Persistent Objects, Martha's Vinyard, Sept. 1990.

[BUCH91] Buchmann. A .; et al. "A Transaction Model for Active Distributed Object Systems", in A. Elmagarmid (Ed.), "Database Transaction Models for Advanced Applics." 1991

[CHAK89] Chakravarthy et.al.; "HiPAC: A Research Project in Active, Time Constrained Database Management", Final Report, Xerox XAIT TR 89-02, Aug. 1989.

[CHAK91] Chakravarthy,S., Mishra,D.; "An Event Specification Language (Snoop) for Active Databases", Univ. Florida, CIS, TR91-23, Sept. 1991.

[CHRY91] Chrysanthis,P; " A Formalism for Extended Transaction Models", Proc.VLDB17, Aug. 1991.

[DAYA88] Dayal et.al. "Rules are Objects Too", Proc. OODBS-2, Bad Muenster, Sept. 1988.

[MANO92] Manola et.al. "Distributed Object Management", IJICIS,1(1) 92:5-42.

[ROZI90] "Overview of the Chorus Distributed Operating System", Chorus Systemes, TR-90-25

[SHET91] Sheth et.al; "Maintaining Consistency of Interdependent Data", CSD-TR-91-16 Purdue Univ. 1991.

Triggers on Database Histories

A. Prasad Sistla Ouri Wolfson

Department of Electrical Engineering and Computer Science
University of Illinois at Chicago, Chicago

1. Introduction

Modern systems, such as traffic control, securities trading and communication networks, are increasingly dependent on real-time software applications for monitor and control. At the center of such applications usually lies an active database, that represents the status of the system. This database is continuously updated by (often remote) sensors, and the software is expected to respond to predefined conditions. Often these conditions refer to the evolution of the database state over time (i.e. the database history). For example, in securities trading, the system may be requested to alert a trader when the value of a particular stock (given as a database attribute A) increases by more than 10% in 15 minutes. We call such conditions temporal triggers, i.e. triggers on the evolution of the database state over time. Furthermore, one may want to specify temporal triggers that also involve external events (such as transaction-begin, transaction-commit, invocation of an object method, etc.) in addition to the database history. The following temporal trigger is one such example— the value of attribute A increases by more than 10% from the time when transaction X commits to the time when transaction Y starts. Existing database management systems, prototypes, and proposed languages, do not provide the capability for specifying temporal triggers. In most of them, the condition part of a rule (in Hipac [1] terminology a rule is an event-condition-action triple) refers to either the current database state, or to the transition from one database state to the next, but not to the complete database history.

2. Temporal Logic Based Languages for Triggers

Temporal Logic (TL) [3] is a formalism for specifying and reasoning about time-varying properties of systems, and therefore it is an appropriate language for specifying temporal triggers. The main feature of TL is that it has special operators that apply exclusively to the time dimension. *Until*, *Since*, *Next-time*, *Last-time*, *Eventually* and *Previously* are some of the widely used temporal operators. In our case, a trigger is a formula of TL, and it is interpreted over a history of system states and a *reference* time. A *system state* is a triple *(database-instance, set-of-external-events, time-stamp)*. Intuitively, a system state defines the database-state and the external events that occur at a particular time. A *history* is a set of system states with distinct time-stamps.

For triggers specified in TL, the history starts at the time when the trigger is entered into the system. The different temporal operators are classified as *past* operators and *future* operators. *Until*, *Next-time* and *Eventually* are future operators, while *Since*, *Last-time* and *Previously* are past operators. The fragment of TL that uses only the past operators is called *Past TL* (PTL) and the fragment the only uses the future operators is called *Future TL* (FTL).

The following are examples of PTL and FTL triggers. Suppose that predicate P on the database state is "A=50"; predicates Q and R (which refer to external events) are "transaction T begins" and "transaction T commits", respectively. Then, "R and (P *Since* Q)" is a PTL trigger. Intuitively, this trigger will fire at the time when T commits, provided that the value of A is 50 continuously from the time when T begins. The same trigger is specified in FTL as "Q and (P *Until* R)"[1].

It is to be noted that when interpreting PTL formulas as triggers, the reference time is taken as the latest time-stamp in the history. In other words, PTL triggers refer to the system states that have a time-stamp smaller than the latest time in the history. In contrast, when interpreting FTL formulas as triggers, the reference time is taken as the time when the formula is entered as a trigger.

PTL and FTL have the same expressive power, but in some cases one is easier and more natural to use than the other. For example, consider the following trigger specification in FTL where P1, P2, Q1 and Q2 are predicates on the database state:

(P1 *Until* Q1) and (P2 *Until* Q2).

For instance, the above trigger becomes reasonable for a stock trader under the following semantics. P1 states that "the IBM stock is less than 90", and P2 states that "the GM stock is less than 38", and Q1 states that "the DJ industrial average is more than 3000", and Q2 states that "the S&P average is more than 400".

The above trigger fires when the latter of the Q1 and Q2 is satisfied (provided that P1 and P2 hold respectively).

Assuming that the trigger is entered at 2pm, in past logic this trigger is expressed as follows.

[Q2 and (P2 *Since* (P1 and P2 *Since* 2pm))] or [Q1 and (P1 *Since* (P1 and P2 *Since* 2pm))]

Clearly the past logic formula is more complex. Actually, if there are k conjuncts (rather than 2) in the future logic formula, then the size of the past logic formula will be exponential in k. Intuitively, the reason for this is that one does not know the order in which the Q's occur, therefore the past logic formula has to account for all possible orders.

One can also use temporal formulas that combine both past and future operators to specify triggers.

In addition to the temporal operators, TL allows the use of global variables that enable comparison of database values derived at different system states. For example, consider the trigger mentioned in the introduction, the value of attribute A increases by more than 10% in 15 minutes. In FTL it can be expressed as follows.

if $x \leftarrow A$ *then*
eventually within 15
$A \geq 1.1x$

In the above formula x is a global variable, and eventually-within-15 is a composite FTL operator that is defined in terms of *Until* and a global variable for time. The above formula should be read as follows: if x is the value of attribute A at the reference time (i.e. when the trigger is entered), then within 15 minutes A has a value grater than or equal to $1.1x$. By adding the *Eventually* operator in front of the above formula we obtain a trigger that fires whenever the value of A increases by 10% in any period of 15 minutes, not just in the first 15 minutes after entering the trigger.

[1]Strictly speaking, we need to have an additional *Eventually* operator in front to allow the beginning of transaction T any time after entering the trigger

3. Comparison to other Specification Formalisms

Query languages based on First Order Logic (FOL), such as SQL, can also be used to specify temporal triggers. In this case each relation, or type, must be augmented with the time attribute and there must be separate relations containing a representation of the current external events. TL is more intuitive since time is an attribute with special properties, e.g., it is monotonically increasing, and TL has special operators for dealing with time naturally (i.e. the way it is used in natural language). Additionally, the triggers specified in TL allow us to identify the least amount of information that needs to be saved over time in order to monitor the trigger. Whereas the FOL approach does not enable the easy identification of such information from the syntax of the specification. The following example illustrates the above point.

Example 1. Consider the simple TL formula (P *Until* Q). In *First Order Logic* (FOL) this is expressed as: $\exists t[t \geq starttime \wedge Q(t) \wedge \forall t'\{starttime \leq t' < t \Rightarrow P(t)\}]$ where *starttime* is the time when the trigger is entered. It is easy to see that the FOL formula is more complicated. It is also difficult to determine the past-database information that has to be saved in order to determine satisfaction of the trigger, and how to minimize this information. On the other hand, in [5] we proposed an incremental algorithm that evaluates the TL formula as follows. When the trigger is entered the algorithm checks whether the current database state satisfies Q. If so, the trigger is satisfied. Otherwise, the algorithm checks whether P is satisfied. If not, the formula is false (will never be satisfied). If P is satisfied, the above procedure is repeated after the next database update which changes either P or Q. Therefore, clearly from this description, no database information has to be saved from one database state to the next. In [5] this processing methodology is generalized to handle an arbitrary FTL formula. Of course, for the general case there may be information that needs to be saved from one state to the next, but the algorithm in [5] minimizes the amount of such information. []

Extensions of SQL to deal with temporal databases [4] were not designed for the specification of triggers. Therefore they also do not enable efficient processing in real time. Additionally, it is hard to reason about the semantics and the expressive power of such extensions, whereas the expressive power of TL has been studied extensively.

Event expressions (EE) is another formalism for specifying triggers [2]. Event expressions are based on regular expressions. They consider the basic events to be the letters of the alphabet, and the expression defines the order in which these basic events occur. For example, the trigger "A,B,C" will fire if A occurs, followed by B, followed by C. The TL formalism allows one to specify time varying properties of external events and database predicates in a unified manner, whereas event expressions are mainly appropriate for specifying the order of external events. Furthermore, real-time properties, i.e. properties on the time of occurrence of different events, cannot be elegantly specified in event expressions. The following property is not easily expressible using event expressions: Events A, B, and C occur in this order, within 60 minutes. This property is easily expressed by the following FTL formula.

A and *if* $T \leftarrow$ *currenttime then*
Eventually
(B and *Eventually*
(C and $currenttime \leq T + 60$))

In the above formula T is a global variable and *currenttime* is a database variable that gives the value of the time. The above formula states that if A is satisfied in the current state and T has the

value of *currenttime*, then eventually B occurs, followed by C; additionally, at the occurrence of C the value of *currenttime* is less than or equal to $T + 60$.

An event expression is usually processed by constructing a finite-state automaton. Another drawback of the EE approach is the high complexity of the automaton-size. It can be superexponential in the size of the event-expression. When global variables are not used, the trigger processing algorithm given in [5] for FTL reduces to a finite state automaton run on the history. The size of the automaton in this algorithm does not suffer from the same high complexity as in the case of the EE approach.

At the level at which we discuss FOL, TL, and EE, in this paper, they are incomparable with respect to expressive power. Specifically, there are variants of TL that are more expressive than variants of EE and FOL, and vice versa. Particularly, it can be shown that a variant of TL without global variables is equivalent in expressive power to EE.

4. Conclusion and Systems Issues

In this short paper we considered the problem of specifying triggers on the evolution of database over time. We have proposed languages based on temporal logics for this purpose. We discussed two types of temporal logic, one based on future temporal operators and the other based on past temporal operators. Finally, we compared this approach with other formalisms, such as Event Expressions and First Order Logic based languages.

Regardless of the formalism in which triggers are expressed, there has to be an algorithm that is invoked at different transition points to detect the satisfaction of the trigger. A *transition point* is either a change to the system state (occurrence of an external event, or update to the database), or a commit of a transaction (when all its updates are considered to have occurred atomically), or a periodic time-instance (say every 10 seconds). How should this invocation be incorporated into the transaction processing mechanism, if at all? Should this invocation be a separate transaction that runs concurrently and serializably with other transactions, some of which may be invocations of the algorithm for other transition points? In case the invocation is due to the commit of a transaction, should it part of the transaction? We have investigated some of these issues, and the results will be presented in a forthcoming paper.

References

[1] S. Chakravarthy et. al., *HiPAC: A Research Project in Active, Time-Constrained Database Management*, TR XAIT-89-02, Xerox Advanced Information Technology.

[2] N. H. Gehani et. al., *Composite Event Specification in ACtive Databases: Model& Implementation*, Proceedings of the 18th International Conference on Very Large Databases, Vancouver, Canada, August, 1992.

[3] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems—Specification*, Springer-Verlag 1992.

[4] R. Snodgrass, *The Temporal Query Language TQuel*, ACM Trans. on Database Systems, 12(2), June 1987.

[5] A. P. Sistla and O. Wolfson, *Temporal Triggers in Active Databases*, Technical Report, Department of Electrical Engineering and Computer Science, Univ. of Illinois at Chicago, 1992.

Active Databases for Approximate Consistency Maintenance

Leonard J. Seligman
The MITRE Corporation
McLean, Virginia

Larry Kerschberg
George Mason University
Fairfax, Virginia

1 Introduction

As collections of distributed, heterogeneous knowledge- and data-based systems enter into federations, there is a real need to define architectures and mechanisms for these systems to communicate and to exchange information relevant to their problem-solving tasks. In such federations, it is often necessary for an application or a component database or knowledge-base to cache data from another federation component. This paper describes a novel approach to supporting approximate consistency maintenance between primary and secondary copies of data in such an environment. The approach relies on an intelligent interface to active databases called a Mediator for Approximate Consistency (MAC). The MAC has several unique features: (1) it permits applications and federation components to specify their consistency requirements declaratively, using a simple extension of a frame-based representation language, (2) it automatically generates the interfaces, rules, and other database objects necessary to enforce those consistency requirements, shielding the application developer from the implementation details of consistency maintenance, and (3) it provides an explicit representation of consistency constraints in the database, which allows them to be queried and reasoned about.

Other researchers have noted the need for approximate consistency maintenance [Alonso90, Rusinkiewicz91]. As they note, it is sometimes inappropriate to use traditional distributed transaction management techniques to enforce consistency between the primary and secondary copies of cached data for the following reasons: first, these techniques do not always effectively support local system autonomy; second, they do not effectively support long transactions; and third, the high degree of consistency which they guarantee may be entirely unnecessary for the application.

Because of the need to maintain various kinds and degrees of consistency, which may be short of 100% consistency, a hardware model of cache consistency is not appropriate. What is required is a *quasi-cache*, as defined in [Alonso90]. Quasi-caches contain *quasi-copies*, which are cached copies whose values are allowed to deviate in controlled ways from the primary copies of those objects. While both [Alonso90] and [Rusinkiewicz91] describe techniques for specifying inter-component consistency constraints, neither describes a general technique for enforcing them. Instead, they rely on custom-coded procedures for refreshing the quasi-copies when the specified consistency constraints are violated. Our work is the first we are aware of to automatically generate the database objects necessary to enforce the consistency constraints specified in a declarative form.

Quasi-caches and their consistency requirements are specified in our approach using a simple extension to a frame-based representation language, as shown in Figure 1. This figure shows the definition of a new derived class, TankUnit, which is a specialization of Unit. The selection-conditions slot uses a query expressed in a logic-based query language to indicate that instances of TankUnit are to be created in the quasi-cache whenever there are instances of Unit and UnitAssets in database Db_1 such that Unit.type is "tank", UnitAssets.asset is "T-72", and Unit.name equals UnitAssets.uname. The retraction-conditions slot indicates that instances are to be purged from the cache only when Unit.type is changed to something other than "tank". There are two consistency-conditions shown in this example. First, a knowledge-base instance should be refreshed whenever it is

more than three versions out of date. Second, it should be refreshed whenever the strength attribute changes by more than 30 percent from the currently cached value. Finally, the msg-priority slot indicates the priority of update messages from the active databases to the cache for instances of TankUnit.

```
(Define-Derived-Class TankUnit ((:superclasses Unit) (:database Db_1))
   (selection-conditions ; Maps result of query into the local slots name,
                         ; type, echelon, and strength
      ((ans _name _type _echelon _strength) <-
               (Unit _name _type _echelon _strength)
               (UnitAssets _uname _asset _number)
               (= _type "tank")
               (= _name _uname)
               (= _asset "T-72")))
   (retraction-conditions (≠ _type "tank"))
   (consistency-conditions
      ((version 3)
       (percent strength 30)))
   (msg-priority 5)))
```

Figure 1: An example quasi-cache definition[1]

2 An Architecture for Approximate Consistency Maintenance

Our approach for managing inter-component consistency relies on the use of an intelligent interface that we call a Mediator for Approximate Consistency (MAC). The term "mediator" comes from [Wiederhold92] and refers to software that presents data at a higher level of abstraction. The MAC abstracts away most changes to the underlying databases and only reports those updates that the quasi-cache specification has defined as being significant.

Figure 2 illustrates the operation of the MAC in its interactions with a single active DBMS.[2] The MAC is composed of two major submodules: the *translator*, which handles communication from the application to the active database, and the *mapper/message handler*, which handles communication from the active database to the application.

The translator accepts the declarative specification of a given class' consistency requirements as it appears in a quasi-cache definition and translates it into the following: queries to be executed immediately, rules for monitoring the future state of the database, and data definition language commands which result in the creation of and updates to consistency constraint objects in the database. The queries which are to be executed immediately are used to populate the quasi-cache with those instances of the newly defined class for which the selection-conditions are satisfied at quasi-cache initialization time. The rules are of three types: *selection-rules*, which are used to monitor the database for future occurrences of the selection-conditions, *retraction-rules*, which are used to monitor the database for the retraction-conditions, and *consistency-rules*, which are used to monitor the database for conditions which require refreshing of quasi-copies. Constraint objects are used to represent constraints explicitly in the database, instead of burying the constraint representation in the where clauses of active database rules. This results in much less rule base maintenance and has the additional advantage that it allows the constraints to themselves be queried

[1] "Ans" refers to the answer relation. Tuples that are returned into the answer relation are mapped into the specified slots (i.e., name, type, echelon, and strength). Variables are preceded by an underscore character.

[2] For a discussion of how the MAC might be adapted to a heterogeneous multidatabase environment, see [Seligman93].

and reasoned about [Shepherd86]. See [Seligman92] for more detail on the constraint representation used and for an example showing automatically generated constraint objects and consistency rules.

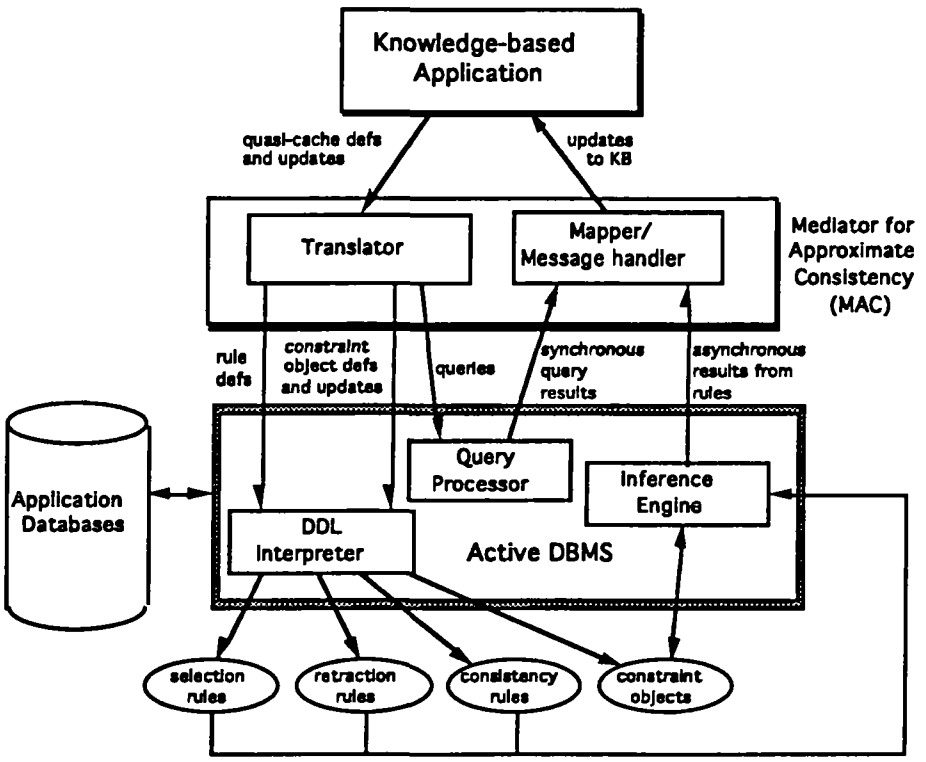

Figure 2: Operation of the MAC

The mapper/message handler receives notification of relevant database updates from the active database and maps them into the representation of the application or component system. The mapper/message handler must accept two kinds of messages: synchronous query results, which are immediate responses to queries forwarded to the database by the translator, and asynchronous quasi-cache update messages, which result from the firing of selection, retraction, and consistency rules in the active database. Asynchronous messages are managed by a priority queue, to ensure that higher priority updates are processed before lower priority ones.

3 Conclusions

This paper has presented a brief overview of a new approach to approximate consistency maintenance using an intelligent interface to active databases. More detailed discussions appear in [Seligman92] and [Seligman93].

We are currently developing a prototype implementation of our approach. We are implementing the MAC in the Common Lisp Object System (CLOS) and are providing an interface from the MAC to POSTGRES [Stonebraker88], a prototype active database. We do not anticipate that providing interfaces to other active databases would be a major difficulty, although we may have to change some of our underlying implementation assumptions (e.g., that every database instance has a unique object identifier). In addition, while we are using an extended relational database in our prototyping, there is nothing in our approach which would prevent us from using an object-oriented database, assuming it had adequate rule processing capabilities. That is why we have been careful to use the generic terms *class* and *instance* instead of their relational counterparts, *relation* and *tuple*.

Following the completion of our prototype implementation, we will use it to integrate an AI planning application with a POSTGRES database. During the course of developing the application, we expect to identify new requirements that will help us refine our design. Also planned is the construction of a simulation model that will allow us to assess when these techniques are more efficient than alternate ones, such as using alerter rules to propagate all changes to the cache and periodic polling of the database to detect critical changes.

4 References

[Alonso90] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System", ACM Trans. on Database Systems, Vol. 15, No. 3, September, 1990.

[Rusinkiewicz91] M. Rusinkiewicz, A. Sheth, and G. Karabatis, "Specifying Interdatabase Dependencies in a Multidatabase Environment", Computer, Vol. 24, No. 12, December 1991.

[Seligman91] L. Seligman and L. Kerschberg, "Active Federation: A New Architecture for Integrating AI and Database Systems," Proc. of Workshop on Integrating AI and Databases, IJCAI-91, Sydney, Australia, August, 1991. Also to appear in L. Delcambre and F. Petry, eds., The Emerging Landscape of Database and Information Systems, JAI Press, 1993.

[Seligman92] L. Seligman and L. Kerschberg, "Approximate Knowledge-base/Database Consistency: An Active Database Approach", in Proc. of First Int. Conf. on Information and Knowledge Management, Baltimore, Maryland, November 1992.

[Seligman93] L. Seligman and L. Kerschberg, "Knowledge-base/Database Consistency in a Federated Multidatabase Environment", Proc. of Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS '93), Vienna, Austria, March, 1993 (to appear).

[Shepherd86] A. Shepherd and L. Kerschberg, "Constraint Management in Expert Database Systems", in L. Kerschberg, ed., Expert Database Systems: Proc. from the First International Workshop, Benjamin Cummings, Menlo Park, CA, 1986.

[Stonebraker88] M. Stonebraker, E. Hanson, and S. Potamianos, "The POSTGRES Rule Manager", IEEE Trans. on Software Engineering, 14(7), July, 1988.

[Wiederhold92] G. Wiederhold, "The Roles of Artificial Intelligence in Information Systems", Journal of Intelligent Information Systems, Vol. 1, No. 1, Kluwer Academic Publishers, August 1992.

Acknowledgements

This research is partly supported by MITRE Sponsored Research (MSR) and DARPA grant number 8987, administered by the Office of Naval Research.

Events and Events rules in Active Databases

Toni Urpí Antoni Olivé
Universitat Politècnica de Catalunya
Pau Gargallo,5
E 08028 Barcelona - Catalonia

Abstract

Change definition and computation is an essential component in several capabilities of an active database, such as integrity constraints checking, materialized view maintenance and condition monitoring. We describe a method for change definition and computation, which is general and flexible. The method can be implemented easily in most active databases systems.

1 Introduction

Active databases provide the capabilities of:

a) Defining one or more changes to be monitored.
b) Computing the changes induced by a database update.
c) Executing some action when some of the defined changes has been induced.

These capabilities are essential in a number of applications, including integrity constraints checking, materialized view maintenance and condition monitoring.

We have developed a method that can be useful for change definition and computation in the above applications. The method is general in the sense that it is logic-based and can be easily implemented in most active database systems.

The method derives, in a systematic way, a set of rules that describe the changes induced by a generic database update. Evaluation of these rules, at execution time, by any query answering method produces the set of changes induced by a particular update. The method can be used in active, deductive databases and, thus, it is also applicable to active relational databases.

2 Deductive Databases

A deductive database D consists of three finite sets: a set F of facts, a set R of deductive rules, and a set I of integrity constraints. A fact is a ground atom. The set of facts is called the Extensional Database (EDB), and the set of deductive rules is called the Intensional Database (IDB).

We assume that database predicates are either base or derived. A base predicate appears only in the extensional database and (eventually) in the body of the deductive rules. A derived predicate appears only in the intensional database. Every database can be defined in this form.

We also assume that each database predicate (base or derived) has a non-null vector of arguments, **k**, that form a key for the predicate. We have then two types of predicates: those, P(**k**,**x**), with key and non-key arguments and those, P(**k**), with only key arguments, where both **k** and **x** are vectors.

2.1 Deductive Rules

A deductive rule is a formula of the form: $A \leftarrow L_1 \wedge \ldots \wedge L_n$ with $n \geq 1$, where A is an atom denoting the conclusion, and $L_1, \ldots, L_n$ are literals representing conditions. Each L_i is either an atom or a negated atom. Any variables in A, $L_1, \ldots, L_n$ are assumed to be universally quantified over the whole formula. We also assume that the terms in the conclusion are distinct variables, and the terms in the conditions are variables or constants.

Condition predicates may be ordinary or evaluable. The former are base or derived predicates, while the latter are predicates, such as the comparison or arithmetic predicates, that can be evaluated without accessing the database.

As usual, we require that the database before and after any update is *allowed*, that is, any variable that occurs in a deductive rule has an occurrence in a positive condition of an ordinary predicate.

2.2 Integrity Constraints

An integrity constraint is a closed first-order formula that the database is required to satisfy. We deal with constraints that have the form of a denial: $\leftarrow L_1 \wedge \ldots \wedge L_n$ with $n \geq 1$, where the L_i are literals, and variables are assumed to be universally quantified over the whole formula. More general constraints can be transformed into this form. For the sake of

uniformity, we associate to each integrity constraint an inconsistency predicate Icn and thus it has the same form as the deductive rule. We call them integrity rules.

To enforce the concept of key we assume that associated to each P(k,x) there is a key integrity constraint that we define as: $Icn(\mathbf{k}) \leftarrow P(\mathbf{k},\mathbf{x}) \wedge P(\mathbf{k},\mathbf{x}') \wedge \mathbf{x} \neq \mathbf{x}'$.

For example, if the EDB has predicate PM(project,manager), the key integrity rule stating that project is a key for the predicate would be:

$Ic(\underline{project}) \leftarrow PM(\underline{project},manager) \wedge PM(\underline{project},manager') \wedge manager \neq manager'$

Note that, for clarity, we underline the key arguments of each predicate.

3 Events and Change Definition

In this section we define the events, a key concept in our method. We also discuss the use of internal events for change definition in the applications mentioned in the introduction.

3.1 Events

Let D° be a database, U an update and D^n the updated database. We say that U induces a transition from D° (the old state) to D^n (the new state). We assume for the moment that U consists of an unspecified set of base facts that have been inserted, deleted and/or modified.

Due to the deductive rules, U may induce other updates on some derived predicates. Let P be a derived predicate, and let P° and P^n denote the evaluation of P in D° and D^n, respectively. Assuming that $P^\circ(\mathbf{K},\mathbf{X})$ holds in D°, where K and X are vectors of constants, three cases are possible:

a.1 $P^n(\mathbf{K},\mathbf{X})$ also holds in D^n
a.2 $\neg\exists y$ such that $P^n(\mathbf{K},y)$ holds in D^n
a.3 $\exists \mathbf{x}'$, such as $\mathbf{X}'$, for which $P^n(\mathbf{K},\mathbf{X}')$ and $\mathbf{X}\neq\mathbf{X}'$ holds in D^n

and assuming that $P^n(\mathbf{K},\mathbf{X})$ holds in D^n, three cases are also possible:

b.1 $P^\circ(\mathbf{K},\mathbf{X})$ also holds in D°
b.2 $\neg\exists y$ such that $P^\circ(\mathbf{K},y)$ holds in D°
b.3 $\exists \mathbf{x}'$, such as $\mathbf{X}'$, for which $P^\circ(\mathbf{K},\mathbf{X}')$ and $\mathbf{X}\neq\mathbf{X}'$ holds in D°

In case a.2 we say that a deletion internal event occurs in the transition, and we denote it by $\delta P(\mathbf{K},\mathbf{X})$. In case b.2 we say that an insertion internal events occurs in the transition, and we denote it by $\iota P(\mathbf{K},\mathbf{X})$. In cases a.3 and b.3 we say that a modification internal event occurs in the transition, and we denote it by $\mu P(\mathbf{K},\mathbf{X},\mathbf{X}')$ and $\mu P(\mathbf{K},\mathbf{X}',\mathbf{X})$, respectively.

Formally, we associate to each derived predicate P an insertion and a deletion internal event predicate defined as:

(1) $\forall \mathbf{k},\mathbf{x}\ (\iota P(\mathbf{k},\mathbf{x}) \leftrightarrow P^n(\mathbf{k},\mathbf{x}) \wedge \neg \exists y P^\circ(\mathbf{k},y))$
(2) $\forall \mathbf{k},\mathbf{x}\ (\delta P(\mathbf{k},\mathbf{x}) \leftrightarrow P^\circ(\mathbf{k},\mathbf{x}) \wedge \neg \exists y P^n(\mathbf{k},y))$

where k and x are vectors of variables (x may be empty).

Furthermore, we associate to each derived predicate P with non-key arguments, a modification internal event predicate defined as:

(3) $\forall \mathbf{k},\mathbf{x},\mathbf{x}'\ (\mu P(\mathbf{k},\mathbf{x},\mathbf{x}') \leftrightarrow P^\circ(\mathbf{k},\mathbf{x}) \wedge P^n(\mathbf{k},\mathbf{x}') \wedge \mathbf{x}\neq\mathbf{x}')$

We handle the modification of a key as a deletion $\delta P(\mathbf{k},\mathbf{x})$ and an insertion $\iota P(\mathbf{k}',\mathbf{x})$.

From the above rules, we then have the following transition axioms:

(4) $\forall \mathbf{k},\mathbf{x}\ (P^\circ(\mathbf{k},\mathbf{x}) \leftrightarrow (P^n(\mathbf{k},\mathbf{x}) \wedge \neg \iota P(\mathbf{k},\mathbf{x}) \wedge \neg \mu P(\mathbf{k},\mathbf{x}',\mathbf{x})) \vee \delta P(\mathbf{k},\mathbf{x}) \vee \mu P(\mathbf{k},\mathbf{x},\mathbf{x}'))$
(5) $\forall \mathbf{k},\mathbf{x}\ (\neg P^\circ(\mathbf{k},\mathbf{x}) \leftrightarrow (\neg P^n(\mathbf{k},\mathbf{x}) \wedge \neg \delta P(\mathbf{k},\mathbf{x}) \wedge \neg \mu P(\mathbf{k},\mathbf{x},\mathbf{x}')) \vee \iota P(\mathbf{k},\mathbf{x}) \vee \mu P(\mathbf{k},\mathbf{x}',\mathbf{x}))$

which relate the old state with the new state and the internal events induced in the transition, and

(6) $\forall \mathbf{k},\mathbf{x}\ (P^n(\mathbf{k},\mathbf{x}) \leftrightarrow (P^\circ(\mathbf{k},\mathbf{x}) \wedge \neg \delta P(\mathbf{k},\mathbf{x}) \wedge \neg \mu P(\mathbf{k},\mathbf{x},\mathbf{x}')) \vee \iota P(\mathbf{k},\mathbf{x}) \vee \mu P(\mathbf{k},\mathbf{x}',\mathbf{x}))$
(7) $\forall \mathbf{k},\mathbf{x}\ (\neg P^n(\mathbf{k},\mathbf{x}) \leftrightarrow (\neg P^\circ(\mathbf{k},\mathbf{x}) \wedge \neg \iota P(\mathbf{k},\mathbf{x}) \wedge \neg \mu P(\mathbf{k},\mathbf{x}',\mathbf{x})) \vee \delta P(\mathbf{k},\mathbf{x}) \vee \mu P(\mathbf{k},\mathbf{x},\mathbf{x}'))$

which relate the new state with the old state and the internal events induced in the transition.

We also use definitions (1), (2) and (3) above for base predicates. In this case, ιP, δP and μP facts represent the external events (given by the update) corresponding to insertion, deletion and modifications of base facts, respectively. Therefore, we assume from now on that U consists of an unspecified set of insertion and/or deletion and/or modification external events. Notice that by (1), (2) and (3) we require:

(8) $\forall \mathbf{k},\mathbf{x}\ (\iota P(\mathbf{k},\mathbf{x}) \rightarrow \neg \exists y P^\circ(\mathbf{k},y))$ and
(9) $\forall \mathbf{k},\mathbf{x}\ (\delta P(\mathbf{k},\mathbf{x}) \rightarrow P^\circ(\mathbf{k},\mathbf{x}))$ and

(10) $\forall k,x,x' (\mu P(k,x,x') \rightarrow P^{\circ}(k,x) \wedge x \neq x')$

also to hold for base predicates. Due to this similar definition, we use the term "event" to denote either an internal or external event.

Example 1

Consider the following database, where predicates EP, PM, EPM, P and C stand for Employee-project, Project-manager, Employee-project-manager, Project and Critical project, respectively.

Base Facts
EP(Peter,2), EP(Tom,2), EP(John,1), PM(2,Ann), PM(1,Mary)
Deductive rules
$EPM(e,p,m) \leftarrow EP(e,p) \wedge PM(p,m)$
$P(p) \leftarrow PM(p,m)$
$C(p) \leftarrow PM(p,m) \wedge p > 10$
Integrity constraints
$Ic1(e,p) \leftarrow EP(e,p) \wedge \neg P(p)$

Let the update be the set of external events U= {ιEP(Roger,12), δEP(Peter,2), ιPM(12,Karen), μPM(1,Mary,Sue)}. The internal events induced by U on EPM are: ιEPM(Roger,12,Karen), δEPM(Peter,2,Ann) and μEPM(John,1,Mary,Sue); the internal events induced on P and C are: ιP(12) and ιC(12), respectively; and no internal events are induced on Ic1.

3.2 Change Definition using the Internal Events

We now explain how to define changes using the internal events. Consider inconsistency predicate Ic1, defined in example 1, meaning that employees must work in projects. Then, insertion internal events ιIc1 will represent violations of the corresponding integrity constraint. If an update to base predicates induces some ιIc fact then the update must be rejected. Deletion and modification internal events are not defined for inconsistency predicates, since we assume that the database is consistent before the update and, therefore, any fact $Ic1^{\circ}(e,p)$ is false.

Now, assume that the EPM derived predicate, defined in example 1, corresponds to a materialized view. In this case, internal events ιEPM, δEPM and μEPM correspond to the insertion, deletion or modification of facts in the extension of EPM. Thus, for instance, if the update induces an ιEPM(E,P,M) fact, then EPM(E,P,M) must be inserted into the extension of the materialized view EPM.

General conditions can also be represented as insertion, deletion or modification internal events of a derived predicate. Assume, for example, that we want to monitor insertions of critical projects, that is, in example 1, insertions on predicate C. Then, ιC(p) can be used to define a change meaning that p is a critical project after the update, but not before. Appropriate actions could be associated to each, or some, of the above changes.

Thus, we see that the single concept of internal event may serve for defining relevant changes in a variety of applications in active deductive databases.

4. Transition Rules

An important aspect to take into account in change computation is the moment in which changes are computed. Changes can be computed either before the application of the update or after it.

Let P be a derived predicate. If change computation is done before the application of the update, P° (old state) may be computed from the database using the definition of P, and P^{n} will be computed using a new rule, called *transition rule*, which defines predicate P^{n} (new state) in terms of old state predicates and events.

On the other hand, if change computation is done after the application of the update, P^{n} (new state) may be computed from the database using the definition of P, and P° will be computed using a new *transition rule*, which defines predicate P° (old state) in terms of new state predicates and events.

The transition rule when change computation is done after the application of the update, is formally defined in [UrO92]. The transition rule in the other case can also be obtained in a similar way.

5 Internal Events Rules

Assume that change computation is done before the application of the update. Replacing P^{n} in (1), (2) and (3) by its corresponding transition rule, and after applying some transformations, we get what we call insertion, deletion and modification internal events rules. They allow us to deduce which ιP, δP and μP facts (induced insertions, deletions and modifications) happen in a transition.

There are several simplifications that can be applied to the deletion, insertion and modification internal events rules. We

can often simplify and even remove some of these rules. Applying these simplifications, we obtain a set of rules semantically equivalent to the former but with a smaller evaluation cost. In fact, the application of our simplifications produces expressions that are more optimized that those obtained by other methods. In this paper, we only give the result of applying them.

Example 2

The internal events rules corresponding to example 1 are:

$$\iota EPM(e,p,m) \leftarrow EP^{o}(e,p) \wedge \neg \delta EP(e,p) \wedge \iota PM(p,m)$$
$$\iota EPM(e,p,m) \leftarrow \iota EP(e,p) \wedge PM^{o}(p,m) \wedge \neg \delta PM(p,m) \wedge \neg \mu PM(p,m,m')$$
$$\iota EPM(e,p,m) \leftarrow \iota EP(e,p) \wedge \iota PM(p,m)$$
$$\iota EPM(e,p,m) \leftarrow \iota EP(e,p) \wedge \mu PM(p,m',m)$$
$$\delta EPM(e,p,m) \leftarrow \delta EP(e,p) \wedge PM^{o}(p,m)$$
$$\delta EPM(e,p,m) \leftarrow EP^{o}(e,p) \wedge \delta PM(p,m)$$
$$\mu EPM(e,p,m,m') \leftarrow EP^{o}(e,p) \wedge \neg \delta EP(e,p) \wedge \mu PM(p,m,m')$$

$$\iota P(p) \leftarrow \iota PM(p,m)$$
$$\delta P(p) \leftarrow \delta PM(p,m)$$

$$\iota C(p) \leftarrow \iota PM(p,m) \wedge p > 10$$
$$\delta Cp) \leftarrow \delta PM(p,m) \wedge p > 10$$

$$\iota Ic1(e,p) \leftarrow EP^{o}(e,p) \wedge \neg \delta EP(e,p) \wedge \delta P(p)$$
$$\iota Ic1(e,p) \leftarrow \iota EP(e,p) \wedge \neg P^{o}(p) \wedge \neg \iota P(p)$$
$$\iota Ic1(e,p) \leftarrow \iota EP(e,p) \wedge \delta P(p)$$

The process for obtaining the internal events rules in the other case, that is, when change computation is done after the applications of the update, is formally defined in [UrO92].

Furthermore, a number of optimization techniques can be naturally incorporated into our method. The most important is the partial evaluation [LlS91] of the transition rules, internal events rules and a given transaction with respect to the relevant internal events. Partial evaluation produces, at compilation time, a set of equivalent rules which which can be evaluated more efficiently at execution time.

We can also take into account some details of a given application of change computation. Thus, in view materialization we have available the old state of the view. In such case, we can easily adapt our rules to take advantage of this knowledge.

6 Change Computation using the Internal Events Rules

The above rules can be directly used to compute the changes induced by an update. We only need to query the database, extended with the above rules and the update, for relevant induced events. Thus, for example,

a) Ic1 will be violated if the query ? ιIc1(e,p) returns a non-empty set.
b) Answer to queries ? ιEPM(e,p,m), ? δEPM(e,d,m), ? μEPM(e,d,m,m') give the updates to the EPM materialized view.
c) Answer to query ? ιC(p) gives the set of critical projects inserted in the update.

Any query-answering system (bottom-up, top-down) can be used.

Acknowledgements

We would like to thank D. Costal, E. Mayol, J.A. Pastor, C. Quer, M.R. Sancho, J.Sistac and E. Teniente for many useful comments. This work has been partially supported by the CICYT PRONTIC program project TIC 680.

References

[LlS91] Lloyd,J.W.;Shepherdson,J.C."Partial evaluation in logic programmming", Journal of Logic programming, 1991, n° 11, pp 217-247.

[Oli91] Olivé,A. " Integrity constraints checking in deductives databases", Proc. of the 17th. VLDB Conf., Barcelona, 1991, pp. 513-523.

[UrO92] Urpí,T.; Olivé, A. " A method for change computation in deductive databases", Proc. of the 18th. VLDB Conf., Vancouver, 1992, pp. 225-237.

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