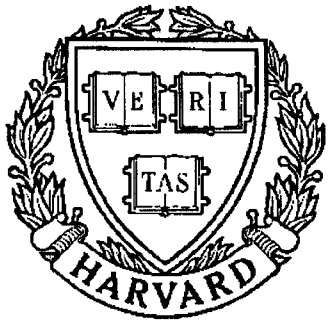
THESIS REPORT
Ph.D.

SYSTEMS
RESEARCH
CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Issues in Integrating Active Rules into Database Systems

by R. J. Cochrane
Advisor: L. Mark

Ph.D. 92-1

Abstract

Title of Dissertation: Issues in Integrating Active Rules Into Database Systems

Roberta Jo Cochrane, Doctor of Philosophy, 1991

Dissertation directed by: Assistant Professor Leo Mark
Department of Computer Science

An essential feature of next-generation database systems is the ability to define and process rules that respond to database events. We address several issues involved in fully integrating such active rules into multi-user database systems. We do this by investigating two very different database rule systems: the Update Dependency Language, which uses a tentative goal-oriented search strategy, and the Starburst Rule System, which uses a forward-chaining irrevocable control strategy.

For the Update Dependency Language, we formally define the language and define safety requirements for its conditions and procedures. We analyze the locking requirements for two different execution strategies: one that uses a depth-first search and one that uses a concurrent search. We show that it is incorrect to release shared locks on failed subpaths before a successful path is found. However, we show that two-phase locking can be relaxed to allow the early release of exclusive locks along failed subpaths.

For the Starburst Rule System, we describe the components that handle recovery in the presence of system-generated and user-requested rollbacks. We investigate the problem of maintaining rule priorities in the Starburst Rule System and others, describing the requirements and implementation of a priority system that combines user-defined priorities and system-generated default priorities. To support an environment in which users can modify rules during normal database operations, we define consistency requirements for rule definition operations and present a solution based on hierarchical locking that maintains these consistency requirements in a multi-user environment.

Issues in Integrating Active Rules Into Database Systems

by

Roberta Jo Cochrane

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1991

Advisory Committee:

Assistant Professor Leo Mark, Chairman/Advisor
Professor John Gannon
Professor Nick Roussopoulos
Associate Professor Christos Faloutsos
Associate Professor George Harhalakis
Assistant Professor Timos Sellis
Dr. Jennifer Widom

Dedication

To my sister Amy and my parents Bernadine and Joe for all their love and support.

Acknowledgements

I would like to express my utmost gratitude to my advisor, Leo Mark, whose encouragement, guidance and friendship have made my time at Maryland an invaluable learning experience, both technically and personally. He coaxed me into taking the Ph.D. comprehensive exams and encouraged me to realize my potential by pursuing a Ph.D. – a decision that has positively influenced my career and my life.

I am grateful for the unique opportunity to work with Jennifer Widom and the Exploratory Database Group at IBM Almaden Research Center. Jennifer has been a great inspiration and role model – living proof that there is a way to balance music, life, and a computer science research career. I would also like to express my appreciation to all members of the Starburst Team who contributed to this invaluable experience, especially Rakesh Agrawal who provided encouragement and guidance for the work presented in Chapter 6, and Bruce Lindsay who provided invaluable stimulus and advice for the implementation of the Starburst Rule System.

I would like to recognize Timos Sellis and Louiqa Raschid who contributed to my background in the area of database production rules through seminars and personal conversation. In addition, I have learned much from interactions with the other members of my preliminary examination and dissertation committees: John Gannon, Nick Roussopoulos, Christos Faloutsos, George Harhalakis, and Bill Pugh.

There are three groups of people at Maryland that I would like to mention. (1) Database Systems Group. I will especially remember: Raymond Ng, Christian Jensen, and Alex Delis for several stimulating dinner conversations; Yi Rong for her strength and friendship; Chih-Chen (Chris) Lin for his good humor and group spirit; Helen Papadopoulos whose miracle works with travel arrangements are the least of the talents and pleasure she contributes to the database group. (2) Jack Minker and the Prism group, who adopted me as an adjunct Prismite. In addition to the technical knowledge gained from attending their seminars, I have developed many close friendships: Jorge Lobo and Terry Gaasterland, my traveling companions and dear friends; José Alberto Fernández and Yuan Liu, who had all the solutions to my Gremlin and Tex needs. (3) Administrative Support Staff, especially Nancy Lindley for her friendship, care, encouragement, and ability to maneuver within and around the system.

I am very grateful to James Madison University for providing a stimulating environment from which I obtained a strong foundation, enthusiasm and appreciation for the field of computer science. I would also like to mention a few special friends

whose encouragement and understanding have been invaluable: Jana Mortensen, Gina Thomas, Ann Sullivan, Susan Imber, Amy Bair, Steve Davis, Alan Spangler, and Lynn Apseloff.

Finally, I would like to thank the Systems Research Center and TRW for their financial support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Database Management Systems

A *database management system (DBMS)* [Dat86] is a software package that supports the definition, storage, and manipulation of large amounts of data shared among multiple users. The goal in the development of such a system is to support correct and efficient interactions between users and stored data. There are several features inherent in all DBMSs. Each system supports a *data model* that defines the way the users and application programs interact with the DBMS. Users define the structure of their data, referred to as the *schema*, using the model's data definition language (DDL). They retrieve (or *query*) and modify data using the model's data manipulation language (DML). The data model also typically includes a *view* mechanism for data abstraction so that different users can access shared data according to their individual needs.

Most DBMSs support concurrent access of data by multiple users and, therefore, provide certain basic mechanisms for ensuring data security and integrity. Users interact with these systems through *transactions* [Gra78], which allow the users to group together database operations that should be executed as atomic units. Each transaction should appear to execute in isolation and should execute entirely or not at all. To ensure the correct execution of transactions, concurrency control and recovery mechanisms are provided. The concurrency control mechanism prevents transactions from interfering with each other; it guarantees that errors will not be introduced by the fact that the transactions are running concurrently. The recovery mechanism maintains data during normal system operation that allows, to the extent possible, the restoration of the database to a state that is known to be correct after some failure has occurred.

Although DBMSs are powerful and widely used, there are several areas in which additional functionality is needed. In particular, several researchers have proposed extending conventional DBMSs with more powerful mechanisms for specifying and enforcing integrity constraints, enriching the modification[1] language of the data models, providing additional support for view modification, and providing the ability to react to specific conditions. We discuss these issues in turn.

Integrity refers to the accuracy and consistency of the data in the database. The problem of maintaining integrity is one of guarding the database against invalid modifications. The DBMS should support and enforce *constraints* that govern the acceptable values for the stored data. For example, a *domain constraint* might restrict the value that represents the total number of hours worked by an employee in one week to be less than 100. When data is read and modified by several users, it is crucial that the integrity of the data is preserved at transaction boundaries. Otherwise, the erroneous behavior of one user may propagate throughout the database.

The integrity of the database is usually supported through the specification of integrity constraints that describe the acceptable database states. For example, if employees' weekly hours are recorded in the field `E.hoursWorked`, then the integrity constraint:

$$0 <= \texttt{E.hoursWorked} < 100$$

represents the acceptable values for this field. An integrity constraint facility enforces integrity constraints by checking them at the end of each transaction. Although integrity constraint facilities have been extensively investigated by the database research community, they are typically very inefficient. Hence, most commercial systems provide only rudimentary validation of individual transactions [Dat83]. For example, relational systems [Cod70] typically support only specific types of constraints (such as uniqueness of keys and referential integrity) with corrective actions that are limited to a fixed set of actions. Any additional integrity constraints must unfortunately be maintained manually by each user and embedded in each application program. Such duplication of code is difficult to maintain; furthermore, there is no guarantee that each user or each program properly enforces the integrity constraints.

[1]Throughout this dissertation, we use the term "modification" to refer collectively to the terms "insert", "delete" and "update".

Another approach that has been proposed for increasing the functionality of DBMSs is to enrich their modification languages. As previously noted, conventional modification languages require that each user updating the database is responsible for preserving its integrity. In addition, a user must fully specify all information required for each modification. Proposals for richer modification languages include support for specifying procedures that are associated with the data and are executed whenever the data is modified. In this way, the logic of database modifications, which typically was replicated in each application program, can now be maintained centrally by the DBMS. These procedures perform tasks that help users achieve their desired modifications, including providing default values for unspecified fields and invoking additional modifications to maintain the integrity of the database. For example, suppose in a bank database that the date and time of last access are associated with each account. The date and time need not be supplied by the user, but can be obtained by the system. This logic can be encoded in a procedure that processes the user's request.

The modification language of DBMS should also provide the ability to define view modification policies. Views are a convenient and powerful tool for sharing data. They provide different levels of abstraction to different users according to their individual needs. However, not all views are updatable in conventional systems. Any user who modifies the database will likely need to access the base data from which the views are derived. This is an unnecessary restriction since, even when the view is not theoretically updatable [FC85, BS81], the database designer often knows how a view should be modified. For example, suppose an insurance database in a relational system consists of the base relations `accounts(ins-no, payer)` and `insured(ins-no, patient)`, and the view `dep(payer, patient)` in which `dep.patient` is covered by an insurance policy paid by `dep.payer`. It is not theoretically possible to determine how to delete a tuple from the view `dep`; this tuple can be deleted either by deleting the `payer` from the `accounts` relation or by deleting the `patient` from the `insured` relation. However, there could be an application-specific policy that deleting a tuple from `dep` should be achieved by deleting the `patient` from the `insured` relation. DBMSs need a facility that allows users to define application-specific modification policies.

Conventional database systems are passive repositories of data. The systems only manipulate data in response to explicit requests from users and applications. New database applications, such as inventory control, process control, and factory automation, need systems that are *active*. An active system is one that can automatically respond, without user intervention, to events that are either internal or external to the system itself. Typical events

might include database operations such as an update to a particular value, or temporal events such as the occurrence of a given moment in time. Consider a parts-inventory database that records the quantity of each part in stock. Suppose the stock is maintained by ordering 100 more parts whenever the quantity of a part drops below 10. One solution to this application in conventional systems requires periodic querying of the database (*polling*) at intervals that ensure a timely response. One problem with this solution is that frequent polling is usually required in order to ensure that all items remain in stock. This is a waste of the database resources and often leads to thrashing. Alternatively, this logic could be embedded in each application program (or performed by each interactive user) that decreases the number of parts in the database. However, as mentioned with respect to modifications, such duplication of logic is prone to error and difficult to maintain. If the database has the ability to respond to the occurrence of conditions in the database then this logic could be performed whenever the DBMS detects that the quantity of a part is being modified.

Over the past 15 years, several special-purpose mechanisms have been proposed for enhancing conventional DBMSs with one or more of the above facilities. All of these mechanisms, to some degree, attempt to raise the level of activity of the database. In response to this observation, an area of database research that is currently receiving considerable attention is *active database systems*, which is reflected in the recent collections [Sel89], surveys [HW92, ZB90] and tutorials [DD91, Wid91]. Several of the special-purpose mechanisms include sublanguages in which rules are used to define activity. This is not surprising since each of these facilities requires the ability to define application-specific rules. Rules have been identified as a unifying paradigm for providing a broad range of database facilities [BBB+90] and their incorporation into DBMSs is the main focus of ongoing research in active database systems.

1.2 Active Database Systems

An *active database system* has the ability to recognize certain events and, without user intervention, automatically execute certain operations. The advantages of a DBMS that is capable of reacting to events have been realized and explored for some time. The ancestry of active database systems can be traced back to the ON-conditions of CODASYL [COD73]. Triggered procedures whose events were restricted to modifications on base tables were suggested for System R (but never implemented) [Esw76]. Also, both [Mor83] and [SJR82]

propose active rules for expressing relationships between data items.

The goal of recent efforts has been to increase the capabilities of DBMSs by extending them with powerful, efficient general-purpose rules facilities. By having such a facility, the DBMS can provide centralized control and management of solutions to problems (such as data integrity) that are common to all application programs. Such an environment lends itself to the development of more modular systems and has the added advantage that application programs need not be recompiled when these solutions are modified. It also permits the optimization of rule evaluation. Furthermore, the structure of rules is more amenable to analysis [Ras90] and querying facilities than general-purpose procedures. It is becoming increasingly evident that this feature is important.

Recent work has demonstrated that a general-purpose database rules facility can be used to remedy many deficiencies of traditional database systems. The POSTGRES Rule System II was used to demonstrate that rules can be used for defining views and for specifying the special semantics needed for resolving ambiguous view modifications [SJGP90]. Similarly, the Update Dependency Language was used to demonstrate that view modifications can be specified in the same rule formalism as integrity constraints [MRC91]. Taking this a step further, a general compile-time facility has been designed which partially automates the process of deriving production rules from integrity constraints [CW90]. In addition to increasing the functionality of the databases, rules can also be used to support internal database processing. For example, rules can be used to implement virtual views [SJGP90]. They can also be used to maintain materialized views: modifications to the base tables trigger rules that modify the views. Facilities have been defined for generating such rules from view definitions [CW91].

1.3 Integrating Active Rules into Database Systems

There are several important issues that must be addressed in order to fully integrate active rules into database systems.

- **Rule definition:** What language is used to define rules in the database system? When are rules defined to the database system? How do dynamic changes to the set of rules defined to the system affect concurrently running transactions?

- **Rule activation:** How can rule activation be efficiently supported in a database system? What auxiliary structures are required and how is normal database processing affected?

- **Rule execution:** How does rule execution affect and interact with transaction processing? Are there optimizations to current transaction mechanisms that can be employed to reflect the semantics of rule execution? Are there existing concepts that can be utilized to support the execution of rules?

- **Conflict Resolution:** How should rules that are simultaneously triggered be executed?

The integration of rules into database systems has recently received considerable attention, especially with respect to language constructs, execution models and efficiency issues. However, very few efforts have addressed the problems of defining and executing rules in a multi-user environment. We investigate these issues in the context of two different rule paradigms: the Update Dependency Language [MRC91], which is tuple-oriented and uses a tentative control strategy (i.e. backtracking) for rule execution, and the Starburst Rule System [WF89a, WF90], which is set-oriented and uses a forward-chaining, irrevocable control strategy.

We investigate concurrency control requirements for each paradigm. For both paradigms, rule execution works correctly using existing concurrency control and recovery mechanisms. However, extra provisions must be taken with any auxiliary structures used for rule triggering. Furthermore, since the Update Dependency Language uses a tentative control strategy, existing mechanisms can be relaxed, increasing the concurrency for the execution of such a strategy. We also describe aspects of the Starburst Rule System implementation that were influenced by the need to support concurrency control, recovery and partial rollback.

Rule definition for database applications is a constantly evolving process. We investigate two issues related to the definition of rule sets. We consider the problem of maintaining rule priorities and we also provide consistency requirements and concurrency control for rule definition.

Since the rule system can be used to solve many different problems, several rules may be defined for the same event. When the event occurs, all of the rules defined for the event trigger simultaneously. A rule priority system determines the order in which simultaneously

triggered rules execute. We present a priority system in which the system provides default priorities between all rules while allowing users to control the ordering between specific rules. We define an algorithm for determining a resulting order for rule execution that is *repeatable* and *adherent*. Repeatability guarantees that, for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions is executed twice on the same initial database state. Adherence provides a resulting order in which the pairwise ordering of rules is the same as that implied by the default order unless this order conflicts with the user-defined priorities.

We assume a rule definition facility that allows users to define rules during normal transaction processing. To insure correct behavior of rule execution in the presence of concurrently executing transactions that modify the rule set, we define consistency requirements and provide solutions for enforcing these requirements.

1.4 Dissertation Overview

In Chapter 2, we present background terminology and techniques in both databases and rule systems that are assumed for the remainder of the dissertation. We give a survey of active database rule languages in Chapter 3.

In Chapter 4, we consider the Update Dependency Language and issues involved in integrating it into a DBMS. We begin with a detailed description of the language, in which rules are grouped together in procedures, and we define requirements for *safe* conditions and procedures. We describe two execution strategies for the purpose of giving an execution semantics of the modification procedures, and also for analyzing the locking requirements for backtracking control strategies. The first execution strategy uses a sequential depth-first search and the second uses a fully parallel breadth-first search. We prove that X-locks along failed paths can be released and how all locks obtained on the non-solution paths of a successful execution can be released.

In Chapter 5, we consider the aspects of the Starburst Rule System implementation that were influenced by the need to support concurrency control, recovery and partial rollback. We begin with an overview of the language and the implementation. We then describe the auxiliary structures used for determining rule triggering and discuss optimizations for reducing the amount of information maintained by these structures. Finally, we describe how the system provides for and reacts to complete and partial rollback.

In Chapter 6, we first explain why it is important for rule execution to exhibit deterministic behavior in a multi-user system. We then describe a semantics for this behavior that exhibits repeatability and adherence. We give an algorithm for maintaining this semantics and a proof that the algorithm is correct. We provide an outline for implementing the algorithm and describe incremental algorithms for maintaining the structures used by the algorithm.

In Chapter 7 we consider the problem of dynamically modifying rule sets in the presence of concurrently executing transactions. We give a definition for consistency and describe a mechanism for insuring it. We present our conclusions and open issues in each of these areas in Chapter 8.

Chapter 2

Background

2.1 Database System Components

Figure 2.1 gives an abstract model of a DBMS, illustrating the components that are relevant to this dissertation. A typical interaction with the database system is through a *transaction* [Gra78] that groups together a sequence of database operations. The *Transaction Manager* is the interface between the transactions and the DBMS. It preprocesses all operations issued from the transaction, possibly appending the transaction identifier to the operations, and then forwards them to the appropriate component.

Data definition commands are processed by the *Data Definition Module*, which initializes structures for storing the defined data. The Data Definition Module also stores the definitions in a special database table called the *Schema Catalog*. Access to and modification of data is performed by issuing appropriate commands to the *Query Execution Module*. This module translates queries, issued by both users and internal components, into database requests. It parses and optimizes these queries into a *query plan* that is submitted to the *Data Manager*. To do this, it must access the Schema Catalog.

All access to the database is performed through the *Data Manager*, which performs several functions. Its main purpose is to translate conceptual commands into read and write operations on stored files. In addition to receiving and processing query plans, it also receives and processes transaction commands (*begin transaction, commit, rollback*), issued by the Transaction Manager, and data structure initialization commands, issued by the Data Definition Module. The Data Manager also maintains the information needed to perform authorization, concurrency control, and recovery. We will discuss the particular mechanisms used to perform concurrency control and recovery in Section 2.3.

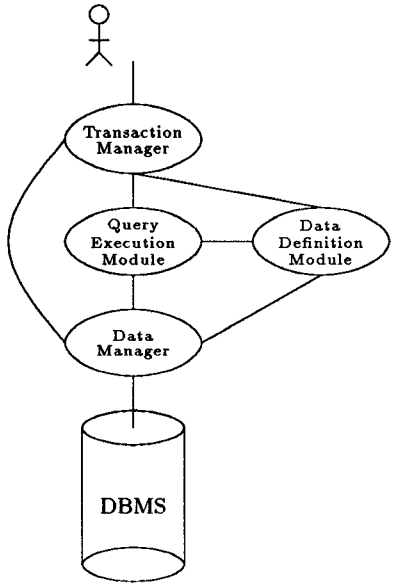


Figure 2.1: Database System Model

To integrate an active rule system into a DBMS, two components must be added to this model: a *Rule Definition Module* and a *Rule Execution Module.* Functionality also must be added to existing components to monitor events and to invoke the Rule Execution Module when rules are triggered.

2.2 Relational Model

The two rule paradigms investigated in this dissertation assume a *Relational Database Management System (RDBMS)*, which supports the relational data model [Cod70]. The relational model is based on named *relations* and *views.* Each relation has a *schema* which includes its name and the name and domain of each of its *attributes.* Each data element in a relation or view is a *tuple* (or row). Several database systems allow duplication and hence support tables rather than relations. We make no assumptions with respect to this matter. Users query and modify relational databases through data manipulation languages. The standard languages, such as SQL and QUEL, perform set-oriented operations, while other paradigms, such as Prolog, manipulate the database with a tuple-at-a-time semantics.

2.3 Transactions

Transactions [Gra78, BHG87] are used to group database operations and are seen by the end-user as the unit of atomicity in database processing. The beginning of a transaction is marked by a *begin* operation, which is usually issued by the Transaction Manager on behalf of the transaction. The user can request that the system commits the work processed by the transaction by issuing a *commit work* operation. This point is known as the *prepare-to-commit* point, and should not be confused with the actual *commit* point. The commit point occurs only after the database performs several functions that insure that the effects of the transaction will persist. At any time before the commit point, a transaction's operations may be rolled back due to a system failure, a deadlock, or an explicitly requested *rollback* from the user. These rollbacks can undo the entire transaction or, if supplied with a previous point in time, only undo a portion of the transaction. The latter class is referred to as *partial rollbacks* [MHL+91].

To guarantee atomicity in the presence of system failures, the DBMS treats the transaction as the unit of recovery of the database. If a transaction fails, the database behaves as if the transaction was never tried. If the transaction succeeds, its effects persist even in the presence of failures. Transactions should be executed exactly once; they should not be lost, partially executed, or executed more than once.

The transaction is also regarded as the unit of consistency. A DBMS must ensure that concurrently executing transactions do not interfere with each other. This is usually done by using concurrency control techniques that ensure *serializability.* That is, any concurrent execution of a set of transactions is equivalent to some serial execution of the transactions. In systems that support integrity constraint checking, integrity constraints are normally checked at the prepare-to-commit point of the transaction.

There are several well-known mechanisms proposed and used for supporting transactions in traditional database systems. In this section, we describe several of these techniques that we use and extend in this dissertation.

2.3.1 Two-Phase Locking

In basic two-phase locking [EGLT76], locks are obtained on data items that are read or written. Shared locks (S-locks) must be obtained by a transaction before it reads a data item. Exclusive locks (X-locks) must be obtained by a transaction before it modifies a data

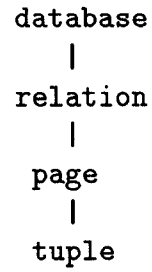


Figure 2.2: Hierarchy of Database Objects

item. An X-lock on a data item can be obtained only if no other transaction holds an S-lock or an X-lock on that item. An S-lock on a data item can be obtained only if no other transaction holds an X-lock on that data item. In other words, S-locks are compatible only with S-locks, and X-locks are not compatible with S-locks or X-locks. The "two-phase" part of this protocol requires that all locks are obtained in a first phase of each transaction, and all locks are released in a second phase. That is, once a lock has been released, no other locks can be obtained. Two-phase locking guarantees serializability[EGLT76, BHG87].

2.3.2 Hierarchical Locking

Hierarchical locking [GLPT76] is an extension of two-phase locking that allows simultaneous locking at varying levels of granularity. The Data Manager chooses the appropriate locking granularity for an operation based on the way it accesses the database. As an example, consider a database query that scans an entire relation. If the tuple is the only level of data item that can be locked, then the Data Manager incurs severe overhead if it must lock each tuple in the relation. However, with hierarchical locking, the Data Manager need only obtain a lock on the relation, which effectively locks the entire relation. Conversely, if the relation is the only locking granularity, then any operation that accesses only one tuple unnecessarily locks the entire relation, reducing the allowable concurrency in the system.

Here we present a simplified adaptation of hierarchical locking to relational databases. A general presentation of hierarchical locking is found in [GLPT76]. The database can be organized into the hierarchy given in Figure 2.2. Each relation is a sub-object of the entire database, each page is a sub-object of the relation that it belongs to, and each tuple is a sub-object of the page on which it is stored.

	S	X	IS	IX	SIX
S	y		y		
X					
IS	y		y	y	y
IX			y	y	
SIX			y		

Figure 2.3: Lock Compatibility Matrix

Locks can be obtained at any level in the hierarchy. There are five different lock modes: *Shared* (S), *Exclusive* (X), *Intention Shared* (IS), *Intention Exclusive* (IX), and *Shared Intention Exclusive* (SIX). As in two-phase locking, S-locks grant permission to read and X-locks grant permission to write. The difference is that an S-lock or an X-lock on a higher level object implicitly applies to all of its descendents. The other lock modes were introduced to enforce serializability when locks are obtained for different granularities within the same hierarchy. The compatibility of the different lock modes is given in Figure 2.3. Separate transactions can simultaneously hold a lock on the same object only if there is a "y" in the matrix corresponding to the lock modes. Permission to request a lock at a node must be obtained according to the following protocol:

- To request an S-lock or an IS-lock on a node N, a transaction must first hold an IX-lock or an IS-lock on all of N's ancestors.
- To request an X-lock, SIX-lock, or an IX-lock on a node N, a transaction must first hold an SIX- or an IX-lock for all of N's ancestors.

Two-phase hierarchical locking guarantees serializability since it is equivalent to conventional two-phase locking [GLPT76].

2.3.3 Nested Transactions

The *nested transaction* model [Mos85] provides a structuring facility for transactions that supports internal sub-transaction parallelism and recovery. In addition to primitive database operations, a transaction may contain any number of sub-transactions with arbitrarily deep nesting. The execution of a nested transaction is modeled by a tree in which transactions and operations are represented as nodes and parent transactions invoke their children. The root of the tree represents a transaction, referred to as the top-level transaction, that is

created by a user or an application program; the leaves represent all the primitive database operations issued on behalf of the top-level transaction.

Top-level transactions and nested transactions differ in that the effects of a nested transaction are not permanent until the top-level transaction commits. However, all sibling transactions can be executed concurrently and the resulting execution is serializable. Each transaction commits or aborts independently of its sibling transactions. However, the parent of a sub-transaction is notified when one of its subtransactions commits or aborts and can react accordingly. It may decide to unilaterally abort or, alternatively, to execute another sub-transaction.

We only present here the locking requirements that apply to nested transactions [RM89]. Concurrency control and recovery mechanisms for nested transactions have also been extensively investigated and can be found in the literature [HR87, Mos87, Mos85, RM89].

- When a sub-transaction commits, all of the locks that it holds and retains are inherited by its parent, which then retains the locks. When a top-level transaction commits, all the locks that it holds and retains are released.

- A transaction T can obtain a lock if no other transaction holds the lock and the only other transactions that retain the lock are T's ancestors.

- When a transaction aborts it releases all the locks that it holds or retains.

The nested transaction model has been used to provide an execution model for database rule systems [MD89, BM91]. We investigate the use of nested transactions for the execution of Update Dependency procedures in Chapter 4.6.

2.3.4 Applicability to Dissertation

We use and extend the techniques described above in various sections of the dissertation. We assume that *two-phase locking* [EGLT76] is the protocol used to ensure serializability. We show how this technique can be relaxed in the execution of Update Dependency procedures in Chapter 4.7. *Hierarchical locking* [GLPT76], an extension of two-phase locking, is used in Chapter 7 to permit transactions that contain rule definition statements to run concurrently with other database transactions. A variation of *nested transactions* [Mos85] is described as an alternative for ensuring mutual exclusion for the parallel execution of Update Dependency procedures in Chapter 4.6. In Chapter 5.6, we use a strategy for handling

rollback in the implementation of the Starburst Rule System that is similar in philosophy to ARIES [MHL+91], the recovery mechanism of Starburst.

2.4 Rule-Based Systems

The three main components of a traditional rule-based system are the *global database*, the *rule set*, and the *control strategy* [Nil80]. The global database, sometimes referred to as the working memory, is the central data structure that is accessible by all rules in the rule set.

The rule set contains a set of rules of the following form:

if *antecedent*
then *consequent*

The *antecedent* is a condition that is evaluated over the global database. If this condition is satisfied, then the rule is applied by executing the *consequent.* Execution of the consequent typically may assert a proposition, modify the data, modify the rules, or execute some external event.

The control strategy, or rule interpreter, determines when rule conditions are evaluated, selects applicable rules, applies them, and ceases computation when a prespecified condition for termination is satisfied. We characterize rule-based systems on two dimensions. The first dimension is the type of event that activates rule execution, and the second is how rules are applied.

In a *forward-chaining* control strategy, rule execution is activated by changes to the data. The initial contact with the data is through the antecedent. In contrast, a *backward-chaining* control strategy is goal-driven, so rule execution is activated by a request to prove a given consequent. As an analogy in database systems, a system that keeps materialized views that are updated when modifications are made to base relations is a forward-chaining system; a system that evaluates views when they are requested is a backward-chaining system.

The process by which a control strategy applies rules can be either *irrevocable* or *tentative.* When an irrevocable control strategy selects and applies a rule, it does not make provisions for returning to the selection point and trying another rule. However, when a tentative control strategy selects and applies a rule, it saves the state of the global database immediately before the rule is applied and has the option to return to this state at some future point in the execution.

```
recognize-act()
begin
  match(conflict-set);
  while conflict-set not empty
     choose R from conflict-set;
     apply-action(R);
     match(conflict-set);
  endwhile
end
```

```
match(conflict-set)
begin
  conflict-set = empty-set;
  for each rule R
    if check-condition(R) then
      add R to conflict-set;
    endif
  endfor
end
```

Figure 2.4: Recognize-Act Cycle

The *recognize-act cycle* is a popular irrevocable control strategy that characterizes a special class of rule-based systems called *production systems* [For81]. Using this control strategy, rules are processed according to the algorithm in Figure 2.4. At each iteration, all the rules are evaluated against the global database. In the *match* phase, each rule whose condition is satisfied by the global database is entered into the *conflict set.* The control strategy then uses some policy to determine which satisfied rule to *fire* (apply). This policy is the *conflict resolution* strategy. The chosen rule is then applied, which may change the database. This process is continued until none of the rules are applicable (i.e. their conditions are not satisfied by the global database). Some modifications of this control strategy add a *selection* phase before each *match* phase that selects the set of rules from the rule base to be matched against the database. This selection phase is often just a filter that eliminates the evaluation of rules whose conditions could not possibly be satisfied.

Sometimes the application of a rule may cause the database to enter a state from which the termination condition cannot be reached by successive applications of rules. In these cases, irrevocable control strategies cannot reach the termination condition and successfully terminate – a tentative control strategy is more suitable. An interpreter that uses a tentative control strategy does not commit to the choice of a selected rule. Instead, it makes provisions for undoing the effects of an applied rule if, at a later stage in the execution, it reaches a state in which no rules are applicable and the termination condition is still not satisfied.

There are two types of *tentative* control strategies. A *backtracking* control strategy establishes a backtrack point whenever a rule is selected. Should the selected rule lead to

an unsuccessful execution, the state of the computation is restored to the state immediately before the rule was applied (the backtrack point). Then, other applicable rules can be tried. *Graph-search* control strategies track the effects of several rules simultaneously and are convenient models for parallel execution of independent rules [Nil80].

In general, rule-based systems are characterized by the fact that they distinguish between data, rules, and control. This separation provides a good environment for the evolutionary development of solutions to problems, since changes to the data, rules, and control strategy can be made independently.

Chapter 3

Survey of Active Database Systems

3.1 Introduction

An active database system has the ability to recognize and respond to certain events or database conditions. Most research in active database systems proposes integrating a general-purpose rules facility into the DBMS. Work in this area has concentrated on solving two problems: (1) the design of appropriate rule languages for database environments and (2) the development of techniques for efficient rule execution. This section surveys the design of a number of database rule systems that have been proposed.

Database rule systems typically have adapted rule constructs from the artificial intelligence (AI) research community [HR85] for database applications and environments. With some exceptions, these languages use a variation of production rules, first proposed for HiPAC, called event-condition-action (ECA) rules. ECA rules have the following form:

when *event* ,
[**if** *condition* ,]
then *action*

The rule is *triggered* whenever the *event* occurs. When a rule is triggered, its *condition* is evaluated against the database and, if satisfied, the *action* is performed. The *action* may cause other *events* to occur that may, in turn, trigger other rules. There are many different adaptations of this style of production rules and other constructs from rule-based languages to databases. The major differences in these adaptations are:

- the types of events and actions allowed,
- the evaluation times of the events, conditions and actions,

- the way simultaneously triggered rules interact, and
- set-oriented versus instance-oriented processing.

In this survey of proposed and existing systems, we highlight the various extensions to the rule paradigm that distinguish each one.

3.2 HiPAC

We begin our survey with the HiPAC project [MD89] because it is perhaps the most general and thorough database rule system yet proposed. Although HiPAC was designed for an object-oriented database system, many of its concepts are not specific to the object-oriented paradigm and can be easily adapted to relational systems. The main benefit derived from the object-oriented framework is that rules are defined as first-class objects and are subject to the same operations as user-defined objects. This is discussed further in Section 7.5.

HiPAC proposed the ECA style of rules, described above, which allow the explicit specification of the triggering event of the rule. The triggering event language used for HiPAC is very expressive. Rules can be triggered by primitive events such as database modifications (*delete, insert, update*). They can also be triggered by temporal events specified by an absolute time, a time relative to some baseline, or a periodic time. These events can be combined using disjunction, conjunction, sequence, and negation. Rule processing in HiPAC is essentially *instance-oriented*: each occurrence of a rule's event triggers an instance of the rule.

An important aspect of rules triggered by database operations is whether they can refer to the *transition values* of the event; for example, when a database modification triggers a rule it is often convenient for the condition and action of the rule to have access to the NEW and OLD values of the triggering modification. Dynamic integrity constraints are not specifiable without access to these values. In HiPAC, parameters can be passed from the event to the condition and from the event and condition to the action. In this way transition values are made accessible to the rules.

A unique aspect of the HiPAC rule system is its use of nested transactions in rule condition evaluation and action execution. This is related to our parallel execution of Update Dependency procedures described in Chapter 4.6. When a rule is triggered, a new transaction is created to evaluate the rule's condition. If the condition *succeeds* (i.e. evaluates to

true), a new transaction is created to execute the action.

In addition to supporting fully contained subtransactions, HiPAC extends the nested transaction model to allow transactions to create other top-level transactions. The success and commitment of the child and parent transactions can be completely independent (*decoupled*). Alternatively, the child transaction's commit point can be forced to be serialized after the parent transaction's commit point (*causally-dependent decoupled*).

This extension of the nested transaction model supports the specification of different *coupling modes* for both event-condition coupling and condition-action coupling. This provides a flexible semantics for rule execution with respect to the database transaction boundaries. There are three different coupling modes [DD91]:

immediate: An immediate event-condition coupling specifies that the condition is evaluated immediately after and in the same transaction as the event that triggered the rule. Similarly, an immediate condition-action coupling specifies that, if the condition is satisfied, the action is executed in the condition's transaction immediately after the condition is evaluated.

deferred: A deferred event-condition coupling specifies that the condition is evaluated at the prepare-to-commit point of the event's transaction. Similarly, a deferred condition-action coupling specifies that the action is evaluated at the prepare-to-commit point of the condition's transaction. This coupling mode is useful for checking integrity constraints and for maintaining replicated data.

decoupled: A decoupled event-condition coupling specifies that, when an event occurs, the condition should be evaluated in a separate transaction. Similarly, a decoupled condition-action coupling specifies that, if the condition evaluates to true, the action should be evaluated in a separate transaction. This coupling mode is useful for executing actions on which the triggering event's transaction should not depend, such as auditing. Decoupled coupling modes can be specified as *causally dependent* or *independent.* A causally-dependent coupling mode specifies that the new transaction must be serialized after the originating transaction, and if the originating transaction is aborted or *fails* (i.e. the user or program initiates the abort based on some invalid computation), then the new transaction is also aborted. This coupling mode is useful for chaining together steps of a long activity, allowing the transaction in which the event occurred to commit

independently of the success of the transaction that applies the rule.

Not all combinations of coupling modes are allowed. It is not possible to have a rule with a deferred event-condition coupling mode together with an immediate condition-action coupling mode. Similarly, a rule cannot have both decoupled event-condition and deferred condition-action coupling modes.

HiPAC rules are processed as follows. When an event occurs, it triggers a set of rules *RS*. For each rule in *RS*, a transaction that executes the rule is scheduled according to the rule's event-condition coupling mode. This transaction evaluates the condition, and, if the condition is true, schedules the transaction that executes the rule's action according to the rule's condition-action coupling mode. Nested transactions (as opposed to conflict resolution strategies) are used to concurrently execute all simultaneously triggered rules. These rules are serialized by the transaction scheduler but will not have deterministic behavior. However, priorities can also be introduced to influence the serialization order of concurrently executing siblings. This implies that the resulting order of rule execution is determined at run-time by the scheduler. Users control which rules can be fired through a mechanism for enabling and disabling rules.

The HiPAC design also provides for failure recovery. When the execution of a rule fails, the rule's subtransaction is aborted. This failure is reported to the parent transaction, which can take the appropriate action. If there is a system failure, then all events that occurred during the execution of committed transactions are recovered as part of normal transaction recovery, and all uncommitted transactions that are triggered by the recovered events are restarted.

3.3 The POSTGRES Rule Systems

The proposals for the POSTGRES Rule System (PRSI and PRSII) adapt the production rule paradigm to a relational DBMS supporting QUEL [H+75]. In PRSI [SHP88], rules are defined by tagging QUEL commands with the keywords `always`, `refuse`, or `one-time`. A command tagged with `always` appears to have always just been executed; a command tagged with `refuse` is never allowed to be executed; and a command tagged with `one-time` is executed exactly once the first time it is triggered. For example, the following rule has the effect of making any employee with a Ph.D. always have a "senior" employee class.

```
replace always emp(class = ''senior'')
where emp.education = ''Ph.D.''
```

The triggering events for these rules are not explicitly defined and depend on the rule itself. The above rule is triggered whenever (1) the employee class for an employee with a Ph.D. is updated, (2) an employee's education is changed to Ph.D., or (3) a new employee is inserted. *Alerters*, which are rules that simply send messages when an event occurs, can be written by applying the **always** keyword to QUEL retrievals. For example, the following rule will retrieve Mike's salary whenever it is updated.

```
retrieve always (emp.salary)
where emp.name = ''Mike''
```

As with HiPAC, PRSI rules are instance-oriented. There are two alternative control strategies for rule processing (which sometimes give different semantics). The first is forward-chaining and is equivalent to combining HiPAC's immediate event-condition and immediate condition-action coupling modes. Rule conditions are checked and actions are executed when the triggering event occurs. The second strategy is backward-chaining – the rule is executed only when the data that the rule writes is queried. The user has the option to choose a control strategy, but if none is specified, the optimizer chooses the strategy based on the access patterns of the database.

Using PRSI's forward-chaining control strategy, rules are processed as follows. When an external tuple-level operation occurs, it may cause a new combination of tuples to satisfy the conditions of several rules. These rules are executed in sequence according to the conflict resolution semantics, which will be discussed later. The execution of a rule's action may recursively trigger other rules causing a nested sequence of rule invocation which is executed within the same transaction.

The backward-chaining control strategy processes rules when users issue queries that reference the data written by the rules. The query processor detects the occurrence of such a query and uses query rewrite techniques to produce a query that incorporates the rule (i.e. the given query is replaced with a new query that utilizes the rule). Not all rules can be processed using this strategy, as described in [SEH87].

One of design criteria for PRSI was to support conflicts or exceptions, such as allowing the general rule that "all birds fly" with the exception that "penguins do not fly". There

must be some semantics for dealing with simultaneously triggered rules, which results in a conflict resolution strategy. Several different semantics are proposed: *priority*, *random* and *union*. Consider the following two rules.

```
replace always EMP(salary=E.salary) using E in EMP
where E.name = "Fred" and EMP.name = "Mike"

replace always EMP(salary=E.salary) using E in EMP
where E.name = "Bill" and EMP.name = "Mike".
```

If random semantics are applied to these two rules,then the value of Mike's salary will oscillate between Fred's salary and Bill's salary. If union semantics are used, then the values of both salaries are returned when the value of Mike's salary is requested. Users can also assign absolute priority values to rules (with a default of 0). If several rules trigger simultaneously, then the rule with the highest priority value is chosen first. Additionally, rules for which priorities are not specified may also execute. If none of the triggered rules have a priority at least one of the rules must execute.

PRSI rules are syntactically a simple extension of the query language and are therefore easy for users to learn. However, because of this simplicity, they were not powerful enough to support many of the basic database tasks. For example, rules cannot refer to the transition values of their events. Furthermore, the inability to make events explicit makes it impossible to have fine control over independent events that trigger the same rule. Such control is needed to support the specification of view modification policies and to provide the logic needed to support different policies for maintaining referential integrity.

A second rule system PRSII [SHP89] was proposed for POSTGRES. The structure of rules in this new language is similar to the ECA rules of HiPAC. The events are explicit and can be any database operation, including retrieval. The rules are still instance-oriented, but the action clauses can refer to the transition values of the event through NEW and OLD keywords.

Support for user control of exception handling (i.e. conflict resolution) is provided by the use of the `exception` keyword. Users specify an exception hierarchy during rule definition by defining that a rule is an exception to some other rule. This exception hierarchy defines a partial ordering among the rules.

In addition to a modified rule structure, PRSII also proposes the use of *rule classes.* Rule classes are used to organize rules logically. The rule classes themselves can also be organized into a hierarchy. Rule classes can be created and deleted, and rules can be added and removed to the existing rule classes. Furthermore, rule classes can be activated and deactivated. The activation (deactivation) of a rule class collectively activates (deactivates) every rule in the rule class and in all descendent rule classes. If a rule is deactivated, then it will not be triggered when its triggering event occurs until the rule is subsequently activated. Rule set definition commands also allow the specification of procedures for execution whenever the rule class is deactivated or activated.

PRSII still supports both a forward-chaining and a backward-chaining control strategy as described previously for PRSI. Forward-chaining is implemented using tuple locking and backward-chaining is simulated using query modification [SJGP90].

3.4 Ariel

Ariel [Han89] is another system that integrates production rules into a database system whose query language is a subset of POSTQUEL. It was developed using the EXODUS database tool kit [CDF+86] as a foundation. The structure of Ariel rules is similar to the ECA rules proposed by HiPAC. Ariel events include temporal (time-based) events, database modifications, and database retrievals.

The most significant difference between Ariel rules and other ECA rules is that the event clause is optional. If a rule has an event clause, the rule is triggered whenever the event occurs. In this case the condition is optional; if it exists, it is evaluated over the entire database state. For example, the following rule is an alerter that executes once to remind Sales department employees of a 2:00 PM staff meeting.

```
define rule staff-mtg-reminder
on time = 14:00
if emp.dept = "Sales"
then execute reminder("Staff Meeting at 2", emp.name);
```

If a rule does not have an event clause, then the condition clause must exist and defines an implicit triggering event or events. The rule is triggered by any database modification that causes a new combination of tuples to satisfy the condition. For example, the following rule executes whenever the salary of an employee is increased by more than 10%.

```
define rule raise-limit
if emp.salary > 1.1 * previous emp.salary
then abort
```

This example also demonstrates that Ariel provides the keyword **previous**, allowing the rule condition to reference the OLD value of an updated or deleted tuple. Without the keyword, the tuple variable **emp** refers to the current value of the modified tuple.

The control strategy in Ariel is forward-chaining. It is a variation of the recognize-act cycle (described in Section 2.4) adding the detection of events as well as condition evaluation in the match phase. The rules are set-oriented: they are triggered by and process sets of database changes rather than single tuple-level changes. This takes advantage of the high performance set-oriented query processor of the DBMS. It uses a deferred event-condition coupling, in the sense that rule triggering is evaluated at the end of each *top-level* command. A top-level command may be a set-oriented modification, retrieval, or a series of such operations grouped together in *compound commands* as in the actions of rules. Rules are executed in the same transaction as the operation that triggers them. All rules that are triggered are added to the conflict set. Conflict resolution, described in detail below, is performed to select a rule to execute. The rule then executes over the entire set of relevant changes that triggered the rule and satisfy the rule's condition. At the end of executing the rule action, any rules that are triggered based on the operations performed by the action are added to the conflict set. This process is continued until the conflict set is empty.

Common tuple variable names are used in the condition and action to specify *implicit linkage* between values in the condition and their use in the action. This linkage can decrease the execution time of rule actions and avoid reexecution (and reiteration) of queries in the action needed to select the tuples that satisfy the condition. A more thorough discussion on the advantages of implicit linkage can be found in [HW92].

Ariel's conflict resolution strategy is a variation of the OPS5 [For81, BFKM85] strategy. Users are allowed to specify absolute priorities for chosen rules. These absolute priorities impose a partial ordering on all the rules since several rules may be assigned the same priority, and rules that do not receive a priority assume the value 0. The following strategy is used to select a rule from the conflict set, C.

C_1 = highest priority rules from C
C_2 = most recently triggered rules from C_1
C_3 = rules with the most selective condition[1] from C_2

if C_3 contains more than one rule, choose one arbitrarily

A rule with a time-based event is scheduled as soon as possible after the event occurs, and is executed in its own transaction. However, there is no guarantee on how soon, after the event occurs, the rule will be executed . One of the main contributions of the Ariel system is the development and evaluation of main-memory structures for very efficient detection of rule triggering based on condition-triggered rules [HCKW90, HW92].

3.5 RDL1

The goal of RDL1 is to provide efficient support for deductive database queries using an underlying rule triggering system. This system uses the if-then style of production rules from AI (i.e. ECA rules without the events). Like Ariel, these rules are set-oriented. The condition of the rule is a relational calculus expression and the action is a sequence of tuple-level modification commands. RDL1 rules are compiled into graph-based internal structures called *Production Compilation Networks (PCNs)* [dMS88], on which optimizations such as logical rewriting can be performed.

RDL1 proposes a control strategy based on fixed-point semantics. Rules are grouped into rule modules that have a set of input relations and a set of target (output) relations. A rule module is activated by a query reference to the module name (i.e. a deductive query). When a rule module is activated, the PCN for the module is executed. The conditions of all the rules in the module are then evaluated over the database state. All rules whose condition evaluation contains at least one satisfying tuple are said to be *relevant.* A rule is triggered only if it is relevant and its action modifies the database state. Note that this differs from the semantics of rule triggering described above for Ariel in which a rule was triggered only by modifications that cause a new combination of tuples to satisfy the condition. Rule processing terminates when either no more rules are relevant or none of the relevant rules changes the database.

For efficiency reasons, a partial order is imposed on the set of rules in a rule module based on the read and write sets of the rules. This partial order, which is equivalent to *stratification* for logic programs, is used by the conflict resolution strategy to choose one rule from the conflict set. A control string can also be included with a rule module allowing the user to impose an explicit ordering between selected rules. Like Ariel, implicit binding between values selected by the condition and acted upon in the actions is supported through

the use of common tuple variables. The condition and the condition-action variable bindings must adhere to safety constraints formally defined in [dMS88]. The architecture of RDL1 is described in [KdMS90].

3.6 RPL

The goal of RPL [DE89] is the development of an OPS5 style production language that is set-oriented, relationally complete, and at least as expressive as OPS5 [DE89]. Its rule language is identical to RDL1, with the exception that the conditions and actions in the original proposal of RPL are based on SQL.

Like RDL1 and Ariel, condition matching is performed for sets of tuples and the action of the rule processes all the tuples that satisfy the condition. This allows the rule execution to take advantage of the set-oriented processing inherent in relational database systems.

However, the control strategy described for RPL is slightly different from both RDL1 and Ariel. It is based on the recognize-act cycle, adapting this control strategy for set-oriented firing. All tuples that simultaneously trigger the selected rule are processed in the same iteration of the recognize-act cycle. The conflict set is reevaluated for each iteration based only on unprocessed changes. This is in contrast to the RDL1 execution module in which a rule's condition is evaluated over the entire database for each iteration. However, RDL1 only triggers rules that add new tuples to the database, assuming that the database is relational and does not contain duplicates. Hence, the resulting semantics of these execution models is essentially the same.

RPL uses view materialization techniques for the detection of rule triggering. The conditions of rules are represented as views on the database. As modifications are made to the database, *Pattern Match Reduction (PMR)* formulas [DWE89] are used to incrementally detect the differences in the views (i.e. rule conditions) that cause rule triggering.

Both RPL and RDL1 develop relational storage structures and retrieval mechanisms for rules [DE88, CdM89]. However, these systems only consider the execution of one rule program in a single-user environment. An open research problem is how these systems behave with concurrent users simultaneously activating rules.

3.7 The Starburst Rule System

The Starburst Rule System [WF90] incorporates set-oriented production rules into a relational database system based on SQL. The implementation of this system [WCL91a] fully integrates rule definition and execution with database processing, providing for features such as concurrency control and rollback. Here we only highlight unique aspects of this system since it is described in detail in Chapter 5.

The rules in this system have the ECA structure proposed by the HiPAC project. However, all the components of the rules are set-oriented and their semantics are defined in terms of sequences of database modifications, as opposed to single database modifications. The events that trigger the rules are database modifications. An event clause of a rule may contain more than one event, indicating that the rule is triggered if at least one of the events occurs.

Rule processing begins at the prepare-to-commit point of any transaction that contains at least one event that triggers a rule, and all rules execute in the same transaction as the event that triggers them. The triggering of an event is evaluated against the net effect of all modifications that occurred during this transaction. Consider the following rule which does not allow employees' salaries to be raised by more than 10% in a given transaction.

```
create rule raise-limit on emp
when (updated(salary)),
if 'exists (select * from nu as (new_updated()),
                          ou as (old_updated())
                    where nu.salary > 1.1 * ou.salary
                    and   nu.empno = ou.empno)'
then ('rollback work');
```

This rule will only be triggered at the prepare-to-commit point of a transaction T if T updates the `salary` field of a tuple t_1 in the `emp` table and, as a consequence of considering only the net effect, t_1 was not previously inserted by T. This example also demonstrates the extension to SQL that permits conditions (and actions) to refer to transition values. The transition table `new_updated()`, which is bound to the variable `nu` in the example, refers to the current values of all updated tuples; the transition table `old_updated()`, bound to `ou` in the example, refers to the values of all updated tuples prior to the beginning of the transaction. Similar transition tables exist for `inserted()` and `deleted()`.

Like Ariel, the control strategy of the Starburst Rule System is an adaptation of the recognize-act cycle that includes event triggering in the match phase. In fact, the match phase only checks for event triggering: the condition of a rule is not evaluated until the rule is selected for execution by the conflict resolution strategy. If the condition evaluates to true, then the rule action is executed and the conflict set is reevaluated; otherwise, the conflict resolution strategy is used to choose the next rule for consideration. Notice that the conflict set is reevaluated whenever a rule action is performed. This, in conjunction with triggering evaluation based on net effect, can result in previously triggered rules no longer being triggered.

Like Ariel and POSTGRES, the conflict resolution strategy for Starburst rules allows the user to impose a partial order on the rules. However, this partial order is specified by the user through explicit relative ordering between rules: a rule may specify that certain rules `follow` or `precede` it as part of its creation. Furthermore, the rule priority scheme described in Chapter 6 resulted from the desire to combine the user-defined partial ordering of the rules with repeatable behavior of the rule system. Users can control which rules are can be fired through a mechanism for enabling and disabling rules.

3.8 The Update Dependency Language

The Update Dependency Language (UDL) [MRC91] is a database rule language quite different from the previously described production-style database rule languages. Its goal is to support integrity constraints, view modification, and application-specific modification policies through a general-purpose modification language based on rules. It uses a goal-oriented tentative control strategy. We give only a brief description here since this language is defined in detail in Chapter 4.

Like RDL1, rules are grouped into procedures, but the procedures are specifically modification procedures – they are activated whenever a tuple-level modification is requested. The semantics of the language requires that exactly one of the rules is applied for any *successful* execution of a procedure activation. If none of the rules can be applied, then the requested modification is *denied.*

The syntax of the rule conditions is based on *domain relational calculus* (DRC), and incorporates the ability to test the instantiation of input parameters of the triggering modification request. The actions of the rules either submit the requested modification to the

database or request subsequent modifications, activating other modification procedures. The semantics of these rules provide explicit binding between the triggering request, the condition, and the actions. The rules are instance-oriented. A rule succeeds if all of its actions can be successfully applied for at least one instance of values that satisfies the rule's condition. If no such instance can be found then the rule fails. If the rule succeeds, exactly one of the instances is chosen as the successful instance.

3.9 Summary

We have surveyed several active database rule systems that have been proposed and/or prototyped by the research community. We focused our description of each system on the various adaptations of the AI rule paradigm to the database environment.

The chart shown in Figures 3.1 and 3.2 summarizes each of the rule systems with respect to various techniques used and features supported. A "$\star$" in an entry indicates that the feature (technique) is supported (used) by the corresponding rule system. For comparison, we include a column for OPS5. Note that many of the systems restrict the features that they support, such as event types, for pragmatic reasons.

The categories for a control strategy were previously described in Chapter 2.4. Each rule system is either forward-chaining or backward-chaining. If backtracking can occur then the control strategy is tentative, otherwise it is irrevocable.

The types of events that can trigger rules include modifications to base data, requests for operations (i.e. the rules are applied or executed before the physical operation is actually performed), retrieval of data, occurrence of an instance in time, and requests for view updates. HiPAC has the most extensive event language, including all event types except requests for operations (which is inherently backward-chaining). Requests for view update are supported by application defined events. The triggering events for a majority of PRSI forward-chaining rules, including the `always retrieve` style rule, are modifications, however it appears that retrieval events can be used for data protection. The event language for PRSII is much more extensive and explicitly supports view update. All of the backward-chaining systems are triggered by requests for operations since, by definition, they are goal oriented.

AI production rule systems, such as OPS5, consist of conditions and actions in which the action is executed whenever the condition is satisfied by the global database. These rules are

Feature/technique	Language	OPS5	HiPAC	PRSI		PRSII	
				forward	backward	forward	backward
Control Strategy	forward-chaining	★	★	★		★	
	backward-chaining				★		★
	irrevocable	★	★	★		★	
	tentative						
Event Types	modification	★	★	★		★	
	operation request				★		★
	retrieval		★	★		★	
	time-based		★				
	view update		★			★	
Event Specification	implicit	★		★			
	explicit		★			★	
Processing Granularity	set-oriented				★		★
	instance-oriented	★	★	★		★	
Condition Scope	entire database		★	★		★	
	transition values	★					
Termination Criteria	empty conflict set	★	★	★	n/a	★	n/a
	fixed-point				n/a		n/a
E-C Binding	immediate		vari-	★		★	
	deferred	★	able		★		★
	decoupled				★		★
C-A Binding	immediate	★	vari-	★		★	
	deferred		able				
	decoupled						
Transition Value Reference	none			★			
	parameterized	★	★				
	keywords					★	
Net effect (triggering)		★		★	n/a	★	n/a
Conflict Resolution	User control		serialization order	exception hierarchy		exception hierarchy	
	Non-deterministic	★	★	★		★	
	Deterministic						
	Recency of data	★					
	Selectivity of Condition	★					
	Stratification						
Programming Support	Deactivate/Activate		★	one-time		★	
	Rule Classes					★	
	Rule Procedures						

Figure 3.1: Summary of Database Rule Systems

Feature/technique	Language	Ariel		RDL1	RPL	Starburst	Update Dependencies
		explicit	implicit				
Control Strategy	forward-chaining	★		★	★	★	
	backward-chaining						★
	irrevocable	★		★	★	★	
	tentative						★
Event Types	modification	★	★	★	★	★	
	operation request						★
	retrieval	★					
	time-based	★					
	view update						★
Event Specification	implicit		★	★	★		
	explicit	★				★	★
Processing Granularity	set-oriented	★		★	★	★	
	instance-oriented						★
Condition Scope	entire database	★		★		★	★
	transition values		★		★		
Termination Criteria	empty conflict set	★			★	★	n/a
	fixed-point			★			n/a
E-C Binding	immediate						★
	deferred	★		★	★	★	
	decoupled						
C-A Binding	immediate	★		★	★	★	★
	deferred						
	decoupled						
Transition Value Reference	none						
	parameterized			★	★		★
	keywords	★				★	
Net effect (triggering)		★		★	★	★	n/a
Conflict Resolution	User control	execution order		execution order		execution order	
	Non-deterministic	★			★		★
	Deterministic					★	
	Recency of data	★					
	Selectivity of Condition	★					
	Stratification			★			
Programming Support	Deactivate/Activate					★	
	Rule Classes						
	Rule Procedures			★			★

Figure 3.2: Summary of Database Rule Systems (cont.)

triggered by implicit events that modify the global database in such a way that causes the condition to become true. This style of rule was adapted by several systems such as RDL1, RPL, PRSI, and one style of Ariel rules. However, the ECA style of rules (proposed by HiPAC and also adopted by Starburst, PRSII, and Ariel) requires the explicit specification of events and allows finer control over the independent events that trigger the same rule.

Databases are inherently set-oriented and, therefore, an obvious alteration to AI production rules is to simultaneously process sets of events that trigger the same rule. This approach has been taken by Ariel, RDL1, RPL, and Starburst. The syntax of the rules for Ariel, RDL1, and RPL is very similar to both OPS5 and to the relational calculus, all of which are instance-oriented. A rule expresses the actions that should be applied to a single instance of the rule. The difference between these systems and OPS5 is that the processing of instances that trigger the same rule is done simultaneously. In contrast, the syntax of Starburst rules is set-oriented. The actions of the rules clearly manipulate the entire set of instances that trigger the rule (see Chapter 5.2). For example, in Starburst, it is possible to perform aggregate functions over the entire set of modifications that caused a rule to fire.

Another difference among the rule systems is the scope of the condition. All of the rule systems with explicit events evaluate the condition over the entire database. But the condition evaluation for several of the rule systems that adapt implicit events from AI production rules is based only on the modifications that cause a new combination of tuples to satisfy the condition. This category of condition scope is referred to as *transition values* in the chart.

All of the forward-chaining rule systems except RDL1 adapt a form of the recognize-act semantics in which rule processing terminates when there are no more triggered rules in the conflict set. In contrast, RDL1 rule processing terminates when a fixed-point of the database state is obtained. That is, the actions of the rules do not modify the database, assuming that the database consists of relations and not tables (i.e. duplication is not allowed).

The event-condition (E-C) and condition-action (C-A) bindings considered in the chart are the same as those described for HiPAC (Section 3.2), with the exception that we consider various units of deferment for the deferred modes in addition to the commit point of transactions. HiPAC is the only system that allows the rule programmer to specify the binding modes. All of the other rule systems described in this chapter have immediate C-A bindings. [2] The forward-chaining PRSs and Update Dependencies also have immediate E-C

[2]The object-oriented rule system for ODE [GJ91] also supports user-defined bindings.

bindings. This together with the fact that the rule actions in both systems can contain multiple commands, causes a nested execution of rules within the same transaction. All of the other systems have a deferred E-C binding, but the unit of deferment varies. In the PRS backward-chaining rule systems, the rules are not evaluated until users issue queries to data affected by the rules. Hence the E-C binding is either deferred or decoupled depending on when the queries are issued. The unit of deferment for OPS5 is the action, since the entire action of a rule, which may contain several commands, is processed before reevaluating the rule conditions. Ariel, RDL1 and RPL simultaneously process the actions of a single rule to the entire set of changes in the recognize-act cycle, and hence the unit of deferment is one pass through this cycle. The unit of deferment for Starburst rules varies. If the rule has not been executed during a transaction, then this unit is the entire transaction, which includes at least the externally generated transaction since rules are first processed at the commit point of the transaction. Otherwise, this unit is the time since the rule's most recent execution.

The "transition value reference" feature indicates if a rule system supports references to transition values and if so, by which mechanism. One of the recognized weaknesses of PRSI is the inability to reference transition values, and this feature has been included in all subsequent rule languages. There are two mechanisms used to provide this feature. The "parameterized" row in the table indicates that the rule system supports this feature through parameterized rules; variables are used to bind the values of the event that triggers the rule to the rule's action. Ariel, Starburst, and PRSII rules support transition value references through the use of keywords (e.g. NEW, OLD) that can appear in the conditions and actions of the rules. In Ariel and PRSII these keywords refer to the NEW or OLD values of the tuple that triggered the rule. However, in Starburst, the keywords refer to relations containing the NEW or OLD values of the set of tuples that triggered the rule.

When an event occurs in HiPAC, a transaction is scheduled for each rule that is triggered by the event, and therefore the net effect of modifications is not considered in deciding which rules are triggered. In the other forward-chaining systems, even the ones in which the E-C binding is immediate, exactly one rule that is triggered by an event is chosen to run. Hence, there may be several modifications to the same tuple before a triggered rule is selected for execution. Normally, when this happens, the net effect of these modifications is used to determine whether a rule's event actually occurred. For example, if a tuple is inserted and then subsequently deleted, then any unexecuted rule triggered by the inserted tuple is no

longer triggered. Note that, although HiPAC does not support net effect for computing rule triggering, the conditions of HiPAC rules can be used to test if the triggering event is still valid.

Several systems have proposed different features to support conflict resolution. All of these proposals indicate that there needs to be some degree of user control while supporting some degree of system-determined behavior. The PRS systems allow the user to specify an exception hierarchy such that, when rules are simultaneously triggered, the rule that is highest in this hierarchy is the only one executed. In PRSI this hierarchy is specified by attaching absolute priority values to the rules. This hierarchy is defined in PRSII through the `exception to` clause in rule definition commands. Ariel and Starburst support user-defined rule priorities that are used to determine the execution order of simultaneously triggered rules. These priorities define a partial-ordering of the rules that affects the semantics of the rules since the execution of one rule can invalidate the triggering event of other rules. Like PRSI, this ordering is specified in Ariel using absolute priority values. In Starburst, it is specified using the `precedes` and `follows` clauses in rule definition commands (see Section 5.2). HiPAC allows users to specify priorities between rules which affects the serialization order of the transactions that are executing the rules. This may appear to have the same effect as specifying the execution order. However, it is different since the execution of a rule's action cannot prevent a simultaneously triggered rule from executing. In Chapter 6 we present a priority system that combines user-specified priorities with a default system-determined behavior that was developed for the Starburst rule system, but can be adapted to provide deterministic behavior for any of the other conflict resolution strategies.

Finally, as with any programming paradigm, once the languages are developed, there is a need for providing support for large-scale development of rule programs. To a degree, some support for structuring rules is inherent in languages such as RDL1 and Update Dependencies. However, constructs for structuring and manipulating rules are just starting to appear in the literature for the production rule languages. These include the ability to group rules into classes and the ability to activate and deactivate rules during normal database processing. PRSI allowed a command to be tagged with the keyword `one-time` indicating that the command should execute exactly once the first time it is triggered. This type of rule effectively deactivated itself after it ran the first time. Starburst, HiPAC and PRSII provide explicit commands for deactivating rules that can be executed by the user or in the action of the rules. PRSII also proposes the use of rule classes to logically organize rules.

3.10 Other Research in Database Rules

Other work in database rule systems has considered the pure AI rule paradigm, concentrating on developing techniques for efficient rule execution. Specifically, these proposals have developed efficient algorithms for detecting triggered rules using both disk-based structures [SJGP90, SLR88], main-memory structures [HCKW90, HW92, KdMS90], and relational structures intended for main-memory databases [DWE89]. The Data-Intensive Production Systems (DIPS) project [SLR88, Lin81] directly adopts the OPS5 paradigm, which has if-then rules and uses the instance-oriented recognize-act cycle previously described. A set-oriented mechanism is used for trigger detection. All database changes are recorded in specially designed auxiliary relations which are stored in the database and used to support rule matching. [RSD91] investigates the use of database transactions for concurrently executing simultaneously triggered rule instances.

In addition to active rule systems for relational systems, there have been numerous other proposals for rules in databases. One prominent area is *deductive databases*, which provides a theoretical framework to study database query languages [Min87]. There have recently been several proposals for active object-oriented database systems [BM91, GJ91, DPG91], in addition to HiPAC.

Active database rule systems currently exist in commercial database systems in limited forms. Rules that are triggered by conditions based on events or single table references are supported in commercial INGRES (V6.0+) [ING89]. Sybase supports forward chaining rules, but these can only be triggered by modification commands and only one triggered event is allowed per modification [How86]. Also, support for the enforcement of referential integrity constraints is required by the ANSI SQL2/SQL3 standard.

Most of the work in active databases thus far has concentrated on defining appropriate languages and developing efficient mechanisms for executing these languages. Equally important is the correct behavior of such rule systems in the presence of concurrently executing transactions. With the exception of HiPAC (whose rule semantics define the behavior of rules in a multi-user environment), none of the proposed systems have thoroughly investigated these issues. This is one main contribution of this dissertation.

Chapter 4

Integrating Update Dependencies into a DBMS

4.1 Introduction

The *Update Dependency Language*, first introduced in [Mar85], is a general-purpose database modification language based on rules. It supports integrity constraints, view modification, and application-specific modification policies. The database designer specifies modification procedures that accompany data definition and are activated when users issue modification commands. A modification procedure consists of a collection of rules and has the notion of success or failure. Each rule defines a transition which, if applied, transforms a valid database state satisfying the condition of the rule into another valid database state. The execution of a modification procedure entails searching for an applicable rule, where the applicability of a rule depends on the current database state and the current variable substitution. Tentative, goal-oriented control strategies are used to find such a rule. If no such rule can be found the procedure fails. Otherwise, the procedure succeeds. In contrast with production systems, the semantics of the language require that exactly one of these rules be applied to any given procedure activation.

User-issued database modification commands are transformed into modification *requests*. A modification request activates a corresponding procedure. If the procedure fails, the modification is rejected and the database remains unchanged; if the procedure succeeds, the modification is granted and the database is modified by the procedure (according to one of the successful rules) to contain the modification. The procedures are tuple-oriented, i.e. they process modifications one tuple at a time. Therefore, for languages with set-oriented commands, several modification requests may be generated for one user modification

command. For example, the SQL statement

```
update R  set A1 = f1, ..., An = fn where P;
```

is translated into one update request for each tuple that satisfies the selection P. We also make the assumption that if no procedure has been specified for a request, the modification will be processed normally by the database system.

Since a modification procedure is activated by a user operation, all modifications the procedure makes will be either committed or aborted atomically with the transaction that issued the operation (the *top-level* transaction). Furthermore, the transaction can test the result of the procedure activation and react accordingly. These tests can be explicitly specified by the user or inherent in the semantics of the top-level transaction, such as in the execution of modification procedures to be discussed in the next section.

We begin this chapter with a detailed description of the syntax and semantics of the Update Dependency Language. The syntax of this language is given in Appendix A. We describe the general format of procedures and define safeness criteria for them. In Section 4.3 we present two different tentative strategies for executing the procedures of this language, adapting both depth-first sequential search and concurrent breadth-first search to the execution of procedures. We describe mechanisms for providing mutual exclusion for the concurrent execution strategy in Section 4.6. In Section 4.7 we discuss multi-user concurrency requirements for the execution strategies. We present an original algorithm for relaxing two-phase locking when a tentative control strategy is applied, and we prove that the original locking semantics for nested transactions does not support consistent executions of modification procedures. Finally, Section 4.8 gives the summary and directions for future research. In Appendix B several examples are given that demonstrate the use of the Update Dependency Language to encode several application-specific solutions of well-known problems.

4.2 Language Definition

Each relation and each view has three modification procedures associated with it – one for each type of modification (**insert, delete, or update**). Each procedure groups together a set of candidate rules that might apply to a modification that activates the procedure. A rule has a condition which tests the database state, followed by a sequence of actions that are

tried if the condition is satisfied. These actions either request other modifications, perform external i/o which can be used to get information that is needed to complete the modification, or actually make physical modifications to the database. Note that the activation of one procedure may activate other procedures including itself. Each modification procedure has a structure represented by the following template:

$$\begin{aligned} &\textit{op-type}\ \textit{rel-name}\ (a_1{=}v_1,\ \ldots,\ a_n{=}v_n[;\ b_1{=}w_1,\ \ldots,\ b_m{=}w_m]) \\ &\qquad \texttt{->}\ \textit{cond}_1,\ \textit{act}_{1,1},\ \ldots,\ \textit{act}_{1,n_1}. \\ &\qquad \texttt{->}\ \ldots \\ &\qquad \texttt{->}\ \textit{cond}_q,\ \textit{act}_{q,1},\ \ldots,\ \textit{act}_{q,n_q}. \end{aligned}$$

The first line of this template is the *procedure head* and each line following an "`->`" represents one *candidate rule.*

Example 4.1 The following example is a modification procedure that controls and guides updates to employees. If the employee's salary is specified by the update, then the update is only allowed if the new salary is less than 7% more than the old salary. If the salary is not specified by the update, then the employee gets a standard 3% raise.

```
update emp(id=E; salary=NEW)
-> nonvar(E) and nonvar(NEW) and emp(id=E, salary=OLD) and
   NEW < 1.07*OLD,
   upd emp(id=E; salary=NEW).
-> nonvar(E) and var(NEW) and emp(id=E, salary=OLD),
   upd emp(id=E; salary=1.03*NEW).
```

■

The procedure head in this example is: `update emp(id=E; salary=NEW)`. All procedure heads contain the operation type *op-type* (i.e. `update`) and the relation name *rel-name* (i.e. `emp`) of the procedure. There is at most one modification procedure for each combination of the *op-type* and *rel-name*. The *op-type* is either **insert**, **delete**, or **update** and the relation name *rel-name* identifies the base relation or view of the modification.

The procedure head also contains either one or two attribute-parameter lists which provide a mechanism for passing values between a request and its corresponding procedure activation. A variable that occurs in an attribute-parameter list is *instantiated* for a procedure activation if the corresponding attribute value is supplied by the activating request. The first attribute-parameter list, which occurs in all procedures, instantiates the variables

v_i with the values of the request's *selection attributes* a_i, $1 \leq i \leq n$. For procedures with operation type **insert**, these values are the attribute values of the inserted tuple. For procedures with operation type **update** or **delete**, these values are the existing attribute values of the tuple that was selected for update or deletion. The second attribute-parameter list, which only occurs in procedures of type **update**, instantiates the variables w_i with the values of the request's *update attributes* b_i, $1 \leq i \leq m$. These values are the new attribute values for the updated tuple.

Example 4.2 The modification procedure in Example 4.1 is an `update` procedure and therefore it has two attribute-parameter lists: (1) `id=E` (2) `salary=NEW`. The only selection attribute is `id`, and the only update attribute is `salary`. Suppose the user issues the following update, which sets the salaries for all employees in the Toy Department to 50,000.

```
update emp
set salary = 50,000
where dept = ''Toy'';
```

Then the modification procedure in example 4.1 will be activated once for each employee in the Toy Department, binding `E` to the employee's `id` and `NEW` to 50,000. ∎

The attribute-parameter lists do not need to contain a variable mapping for each attribute in the procedure's relation. (The procedure in Example 4.1 does not include the employee's department in either attribute-parameter list.) Furthermore, a modification request that activates a procedure does not need to provide values for all attributes represented in the parameter lists. If the latter occurs, then there are variables in the parameter lists which are not instantiated at the beginning of the procedure activation. However, these variables are guaranteed, by procedure safety (see Section 4.2.5), to be instantiated as a result of a successful activation. Hence, this provides a mechanism for passing values back to the activating request which is commonly used by requests that are made from within modification procedures (see Section 4.2.2).

The procedure head is followed by a set of candidate rules. Each *candidate rule* is preceded by an "`->`" and consists of a condition $cond_i$ followed by a sequence of actions $act_{i,j}, 1 \leq j \leq n_i$. Each application of a rule maintains a *substitution* which provides a unique mapping of variables to values. The conditions are queries on the database state, written in a language similar to domain relational calculus. The actions perform physical

modifications to the database, activate other procedures by requesting further modifications, or perform external i/o.

4.2.1 Conditions

Conditions are queries on the database state that are written in a language that is similar to domain relational calculus (DRC) [Ull88]. DRC provides a convenient notation for indicating which parameter values in the procedure head are substituted for variables in the condition, and which attribute values of tuples that satisfy the condition map to variables used in the actions of the rule. *Instantiation tests*, which are meta-logical predicates (à la Prolog [LS86]), are included for testing if a parameter is supplied in a given activation of a procedure.

As with DRC, our conditions are defined recursively. The *atomic formulas* in this condition sub-language are defined as follows:

literal: *rel-name*$(a_1{=}v_1, \ldots, a_k{=}v_k)$

Evaluates to true if there exists at least one tuple in relation (or view) *rel-name* such that, for every instantiated variable v_i, the value of attribute a_i is equal to the value of v_i. Every uninstantiated variable v_j will be instantiated as a result of the evaluation. The instantiated variables act as selection values and the uninstantiated variables act as either join or return-value variables.

comparison: $X\ \theta\ Y,\ \theta \in \{<, \leq, =, \neq, \geq, >\}$

Evaluates to true if the algebraic relation θ holds between symbols X and Y. X and Y are either constants or variables. If the $=$ operator has one variable operand and one constant operand, then the condition evaluates to true with the side effect that the variable gets bound to the constant.

empty condition: Always evaluates to true. This condition is useful for a rule that is always possible for a given procedure, independent of the database state.

negative-instantiation test: **var**(v_i)

Evaluates to true if the variable v_i, introduced in the procedure head, is not supplied by the modification request that activated the current procedure.

positive-instantiation test: **nonvar**(v_i)

Evaluates to true if the variable v_i, introduced in the procedure head, is supplied by

the modification request that activated the current procedure.

Conditions can be combined, negated, and quantified to form other conditions as follows:

negation: **not** C

Evaluates to true if the sub-condition C, which cannot contain any instantiation tests, does not evaluate to true.

conjunction: C_1 **and** ... **and** C_n

Evaluates to true if every condition C_i, $1 \leq i \leq n$ evaluates to true.

disjunction: C_1 **or** ... **or** C_n

Evaluates to true if at least one condition C_i, $1 \leq i \leq n$ evaluates to true.

existential quantification: **exists** v_1 ... v_n C

Evaluates to true if there is at least one mapping of values to variables v_i, $1 \leq i \leq n$ that satisfies the sub-condition C. C cannot contain any instantiation tests, and there must be at least one occurrence of each variable v_i that is free in C.

nesting: (C_i)

Evaluates to true if C_i evaluates to true. Parenthesis are used to alter the default precedence of operators. The unary operators, negation and existential quantification, have the highest precedence and are applied right to left when they occur consecutively. As usual, conjunction has a higher precedence than disjunction.

4.2.2 Actions

There are three types of actions: doit-actions, req-actions, and i/o-actions. Doit-actions perform the physical modifications for the request that activated the procedure, while req-actions request other modifications from within the procedures. I/o actions perform i/o operations that are external to the database and can be used to get information that is needed to complete the modification or to notify the user of the actions taken by the database.

Doit-actions have one of the following forms:

ins *rel-name* $(a_1 = e_1, \ldots, a_n = e_n)$,
del *rel-name* $(a_1 = e_1, \ldots, a_n = e_n)$,
upd *rel-name* $(a_1 = e_1, \ldots, a_n = e_n;\ b_1 = f_1, \ldots, b_m = f_m)$.

The expressions $e_i, 1 \leq i \leq n$, and $f_i, 1 \leq i \leq m$, are evaluated using the current substitution of the action's rule. All variables that occur in these expressions must be instantiated. As with modification procedures, there are two types of attribute-parameter lists. However, these lists serve to provide values from the variables to the attributes.

The first attribute-parameter list occurs in every doit-action and the second one only occurs in doit-actions of type **upd**. Not all attributes of relation *rel-name* must be present in each attribute-parameter list. However, for actions of type **ins**, the value for any attribute not specified in the attribute-parameter list is initialized to `NULL`. The set of attributes $a_1, \ldots, a_n$ in the first attribute-parameter list must contain the attributes in the key of the relation *rel-name* for all actions. For **upd** actions, the second attribute-parameter list indicates the update values for the attributes $b_1, \ldots, b_m$.

Req-actions have a form that is similar to the procedure heads:

insert *rel-name* $(a_1 = e_1, \ldots, a_n = e_n)$,
delete *rel-name* $(a_1 = e_1, \ldots, a_n = e_n)$,
update *rel-name* $(a_1 = e_1, \ldots, a_n = e_n;\ b_1 = f_1, \ldots, b_m = f_m)$.

The expressions $e_i, 1 \leq i \leq n$, and $f_i, 1 \leq i \leq m$, are either uninstantiated variables or arithmetic expressions whose variables are instantiated. Those expressions that are arithmetic expressions (or instantiated variables) supply values to the request's activation. If the expression e_i (f_j) is an uninstantiated variable, then it will be instantiated to the value of the variable that corresponds to a_i (b_j) in the procedure that is activated by the request; the resulting instantiation is added to the substitution of this action's rule. Note that modification procedures may be recursive since an action may request a modification that activates the action's modification procedure.

The *i/o-actions* **read** and **write** provide interaction between procedures and external sources and have the form:

read(x **)**
write(x **)**

A **read** action has the effect of instantiating x with a value from the input stream. A **write** action has the effect of writing the instantiation of x to the output stream.

4.2.3 Procedural Semantics

The execution of a modification procedure results in a success or failure. It entails searching for an applicable rule, where the applicability of a rule depends on the current database state and the current variable substitution. Additionally, all variables in the procedure head that are not instantiated by the activating request are instantiated as a side-effect of the procedure execution, guaranteed by condition safety as described in Section 4.2.5. In this subsection we define the semantics of successful procedures, rules and actions. In Section 4.3, we describe two execution strategies which support the procedural semantics given here.

A modification procedure succeeds if any one of its rules succeeds with the variable substitution provided by the activating request. Exactly one of the successful rules is elected as the *solution rule* and only the effects of this rule persist. If no rules satisfy these requirements, the modification procedure fails and the database state remains unchanged. The choice of the solution rule is not explicitly stated by the language, and the order in which candidate rules appear in a procedure is irrelevant. If the designer wants control over which rule is chosen, then the conditions of the candidate rules should be mutually exclusive.

A rule succeeds if there is at least one tuple that both satisfies the rule's condition and results in at least one successful sequential execution of the rule's actions. Exactly one tuple and corresponding execution of the actions is chosen from all the successful executions of the rule. A rule fails if no tuple satisfies its condition or if none of the tuples that satisfy the condition result in a successful execution of the rule's action.

Actions also have the notion of success or failure. Since req-actions request other modifications, they succeed only if the request is granted. Doit-actions and write i/o-actions always succeed.[1] However, since read actions depend on receiving some input from either an interactive terminal or a file, they may not always succeed. If the request is being processed by an interactive process and the user does not respond within some fixed amount of time the read will fail. The read also fails if it is being processed by a non-interactive process unless a non-empty input file was specified when the process was initiated.

When an action is performed, its effects are only visible to actions that follow it in the candidate rule unless its candidate rule succeeds and is chosen as the solution rule. If this is the case, the effects of all the actions of the rule become the effects of the request

[1]Note that these actions may fail if there is a database or system failure, however, we do not consider these kinds of failures here.

and are visible only where the effects of the request are visible. If the request is from a user modification, then visibility of the request's effects is governed by the user's top-level transaction. If the request is from another modification procedure, then this visibility is governed by the requesting modification procedure.

Each rule maintains a *substitution* which provides a unique mapping of variables to values for all variables in the scope of the rule and its procedure. A variable is *instantiated* if there is a mapping for it in the rule's substitution. This substitution is initialized, through the attribute-parameter lists of the rule's procedure, to the values supplied by the activating request. It is modified by the evaluation of the rule's condition and the execution of its actions. Each tuple that satisfies a condition defines a mapping of values to the condition's variables. A rule's substitution is appended with the mapping defined by its chosen tuple. Note that the chosen tuple's mapping cannot conflict with the rule's substitution (i.e have a different value for any variable that is already mapped to a value in the rule's current substitution). The substitution must also reflect the mappings that are supplied through external i/o and those that result from successful activations of modification procedures (Section 4.2.2). The values returned by a modification procedure are the values represented by the substitution of its solution rule.

Example 4.3 Example 4.1 contains two candidate rules. The first one is applicable when the new salary is specified by the activation and the second one is applicable when the new salary is not specified. The second rule will always fail for every activation submitted as a result of the Toy Department query since the salary is specified. The first rule will only succeed for an activation if 50,000 is not more than 7% larger than the employee's current salary. ∎

4.2.4 Variable Scope and Binding

The variables that occur in modification procedures are analogous to logical variables [LS86] and behave differently from conventional variables (such as those found in C and Fortran). If a variable has a value, the value cannot be overwritten. However, if the action that mapped a value to a variable is undone, then the mapping is also undone and the variable returns to a state of being *uninstantiated.*

The scope of all variables that occur in the procedure head is the entire procedure, and the scope of a variable that only occurs within a candidate rule is limited to the rule in which

it occurs. If a variable is passed a value by the activating request, then the variable acts as a constant in the procedure activation; its value cannot be overwritten. On the other hand, any variables that occur only within the scope of a rule or occur in the procedure head but are not passed a value from the activating request may have different values within the scope of each rule; there is no correlation between common variables in different rules. Since only one rule is chosen as the solution rule, each variable in the head of the procedure will have a unique value at the end of the procedure activation.

4.2.5 Safety

There are constraints that must be imposed on the modification procedures to insure that any given execution is *domain-independent.* Domain independence must be guaranteed for both conditions and procedures. A condition is domain-independent if, for any given instantiation of the variables from the procedure head, the relation generated by the condition depends only on the constants in the condition and the domains of the relations named in the condition. A procedure is domain-independent if it guarantees that each variable in the procedure head will be instantiated for any successful execution of the procedure.

We define the concept of "safe" procedures that guarantees domain-independence by adapting the constraints for safe rules and safe domain-relational-calculus formulas as described in [Ull88] to modification procedures. First we define condition safety and then conclude this subsection with the definition of procedure safety.

Condition Safety

Occurrences of variables in conditions are distinguished as either *free* or *bound.* All occurrences of variables in an atomic formula C are free in C. An occurrence of a variable in a condition C that is a negation, conjunction, disjunction or nesting is free or bound in C if it is free or bound in the subcondition in which it occurs. All free occurrences of variables v_i, $1 \leq i \leq n$ in C are bound in the existential quantification **exists** $v_1 \ \ldots \ v_n \ C$; all other occurrences of variables in existential quantifications are free or bound if they are free or bound in the subconditions in which they occur.

In DRC, variables that occur free in a condition define the relation that corresponds to the condition. In our language, only the instantiated free variables define the relation that corresponds to the condition. A free variable is not an instantiated free variable if it only

occurs in negative instantiation tests in the condition.

A condition of a rule is safe if all instantiated free variables in the condition are *limited*, and any variable that occurs bound in an existential quantification **exists** $v_1 \ \ldots \ v_n \ C$ is limited in the sub-condition C.[2] Each condition can be considered as a conjunction of one or more sub-conditions. For the following discussion, let $C = C_1$ **and** $\ldots$ **and** C_n. A variable v_i is limited in C if one of the sub-conditions in which it occurs is of the following type:

1. positive instantiation test, **nonvar**(v_i). If a variable satisfies this test, the value of the variable is supplied by the originating request and is therefore a constant with respect to the condition.

2. literal, *rel-name*($a_1 = x_1, \ \ldots, \ a_k = x_k$) where v_i is one of the x_i. Since the underlying database is finite, there are only a finite number of possible values for the variable.

3. equality comparison, $v_i{=}X$ or $X{=}v_i$ where X is a constant or limited.

4. a disjunction K_1 **or** $\ldots$ **or** K_q in which v_i is limited in all K_j. Hence, disjunctions are only allowed when they specify either a union or a selection condition for a given set of variables. Any variable that occurs in a disjunction that is not otherwise limited by the enclosing condition must be limited in all sub-conditions of the disjunction.

5. an existential quantification **exists** $v_1 \ \ldots \ v_n \ K$, in which $v_i \neq v_j$, $1 \leq j \leq n$ and v_i is limited in K.

Any instantiated free variable that appears in either a negation test or a negative-instantiation test, which are sources of domain-dependence, must also occur in one of the above types of sub-conditions that will limit the domain of the variable. The safeness of variables that are not instantiated free is guaranteed by procedure safety.

Note that universal quantification is not included in our condition sublanguage. This does not reduce the expressiveness of the conditions since universal quantification can be expressed with negation and existential quantification. Domain-independence is guaranteed by insuring that conditions are *safe*.

[2]The discussion of safe conditions here is similar to that in [Ull88] with a few minor exceptions. We allow the meta-logical type predicates to limit variables. We also relax the safety conditions of disjunctions and incorporate these conditions into the definition of limited.

Procedure Safety

A safe procedure is one in which each candidate rule is safe with respect to the procedure head and all conditions of the candidate rules are safe. Notice, for the purpose of defining safety, a procedure with head h and candidate rules $b_i, 1 \leq i \leq n$, is similar to a set of Datalog [Ull88] rules $d_i, 1 \leq i \leq n$, such that h is the head of each d_i and b_i is the body of d_i. The safe rule requirements apply to each candidate rule with respect to the procedure head. Every variable in the procedure head must be limited in each candidate rule. A variable is limited in a rule if it is limited in the rule's condition or it occurs either in a **req** or a **read** action. Furthermore, any variable in a doit-action $act_{i,j}$ must be limited by the condition or by some preceding action $act_{i,k}, k < j$.

4.2.6 Additional Restrictions

A write must follow an operation that substitutes a value for the write's variable. Hence, the variable must occur as either[3] (1) a limited variable in the condition $cond_i$, or (2) a parameter in a read or req-action $act_{i,k}, k < j$.

On the other hand, a read cannot follow any operation that substitutes a value for the read's variable. Hence, the variable of the read action $act_{i,j}$ cannot occur as (1) a limited variable in the condition $cond_i$, (2) a parameter in any read or req-action $act_{i,k}, k < j$, or (3) a parameter in the procedure head, unless it occurs as an unlimited variable in a negative instantiation test.

Since doit-actions make the physical modifications to the database, they are only allowed from a modification procedure that has the same operation type and relation as the doit-action. Furthermore, they are not allowed in modification procedures for views. Modification requests for a view are performed by requesting modifications on the base relations from which the view is derived. These restrictions encapsulate all modification access to a relation in the relation's three update procedures. Figure 4.1 shows that the the doit-action **ins** $R(....)$, which is defined only if R is a base relation, is only allowed in the procedure **insert** $R(...)$, but procedure **insert** $R(...)$ can be activated by a req-action from another procedure.

[3]If the variable occurs in the procedure head, its value must be supplied by the user or obtained from the database. In either case it will be limited by the condition.

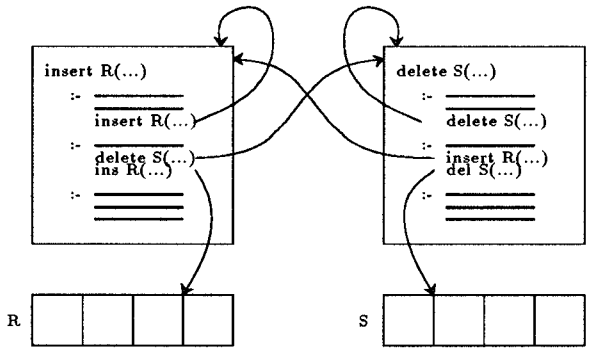


Figure 4.1: Encapsulation of Modifications

4.2.7 Example

We illustrate the Update Dependency Language with one example; for numerous additional examples see Appendix B.

Example 4.4 View Modification - Application Dependent
Assume an insurance database consisting of base relations: `insured(ins-no, patient)` and `accounts(ins-no, payer)` and view: `dependent(payer, patient)`, where dependent patients are those persons whose insurance is covered by some other person. The only restrictions that exist for this application are:

- a patient can only be inserted as a dependent of a payer that has exactly one policy. (This restriction can be lifted by accepting a dependent to be inserted under any one of the insurances paid by the payer, or by adding information to the base relations that indicates the primary insurance policy for a given person.)
- when a dependent tuple is deleted, then the insured patient is no longer covered by the payer's policy. The `account` relation is not affected by modifications to the `dependent` view.

The following procedures support the modification of the `dependent` view.

```
delete dependent(payer=p, patient=q)
/* delete q as a dependent of p  */
-> nonvar(p) and nonvar(q) and p<>q and
   accounts(ins-no=s, payer=p) and insured(ins-no=s, patient=q),
```

```
   delete insured(ins-no=s, patient=q),
   delete dependent(payer=p, patient=q).
-> nonvar(p) and nonvar(q) and
   not exists s (accounts(ins-no=s, payer=p) and
   insured(ins-no=s, patient=q)
   and p<>q).
/* delete q as a dependent of any p */
-> var(p) and nonvar(q) and
   insured(ins-no=s, patient=q) and account(ins-no=s, payer=p) and
   p<>q,
   delete insured(ins-no=s, patient=q),
   delete dependent(patient=q).
-> var(p) and nonvar(q) and
   not exists s, p1 (insured(ins-no=s, patient=q) and
   account(ins-no=s, payer=p1) and p1<>q) and p=NULL.

insert dependent(payer=p, patient=q)
/* q is inserted as a dependent of p on p's one and only account */
-> nonvar(p) and nonvar(q) and p<>q and
   account(ins-no=s1, payer=p) and
   not exists s2 (account(ins-no=s2, payer=p) and s1<>s2) and
   not insured(ins-no=s1, patient=q),
   insert insured(ins-no=s1, patient=q).

update dependent(payer=p1, patient=q; payer=p2)
/* q is removed as a dependent from (all) p1's account(s) and
   made a dependent on p2's one and only account */
-> nonvar(p1) and nonvar(p2) and nonvar(q) and account(payer=p2),
   delete dependent(payer=p1, patient=q),
   insert dependent(payer=p2, patient=q).

/* q is removed as a dependent from all accounts and
   made a dependent on p2's one and only account */
-> var(p1) and nonvar(p2) and nonvar(q) and account(payer=p2),
   delete dependent(patient=q)
   insert dependent(payer=p2, patient=q).
```

■

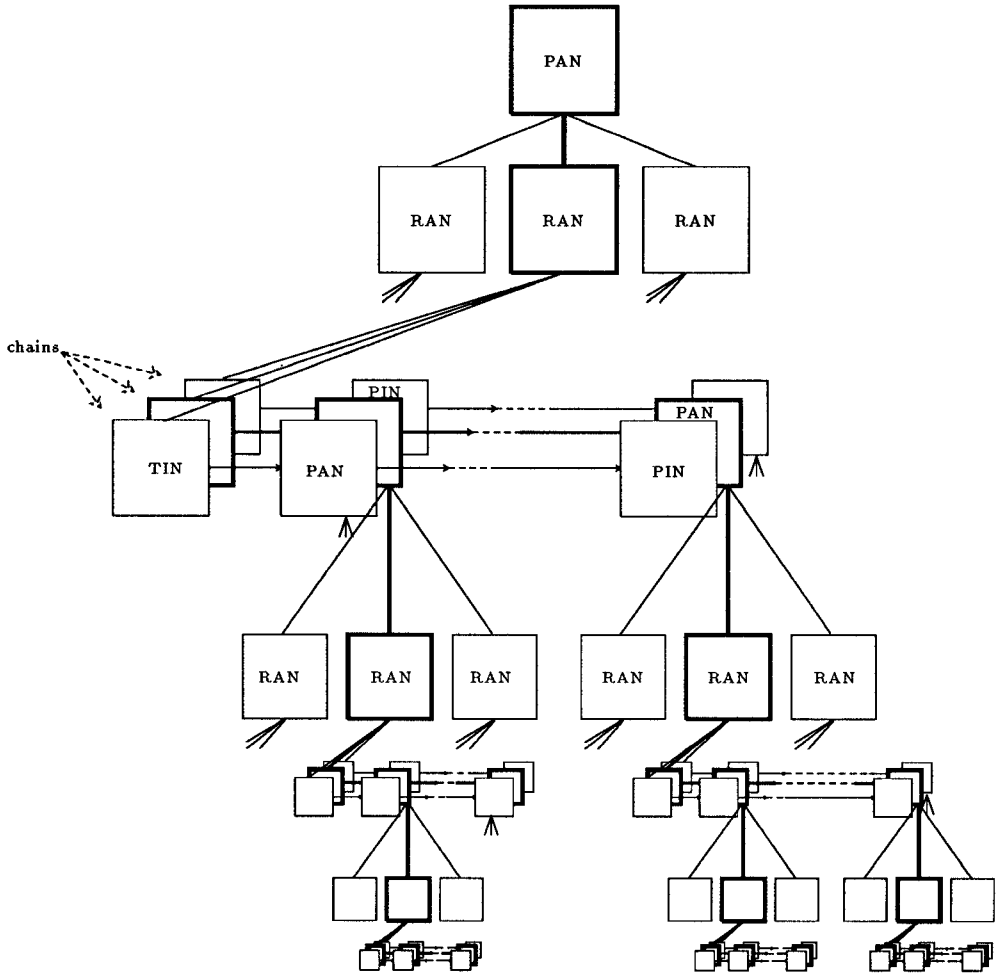


Figure 4.2: Execution Pyramid

4.3 Execution Model

In this section we first define *the update dependency execution pyramid* (Figure 4.2) which is a model of execution for modification procedures. We use this model to describe two control strategies for executing the procedures. Pyramids are related to AND/OR trees [Nil80]. They contain nodes that are linked together in parent-child hierarchies and ordered-lists called *chains*. A parent-child hierarchy represents a disjunction between siblings; the parent is analogous to an OR-node. Chains represent a conjunction between the nodes in the chain with the constraint that the nodes are evaluated in a pre-defined sequential order; the first node in the chain is similar to an AND-node. It is this sequential execution of the nodes

in a chain that distinguishes pyramids from AND/OR trees. The pictorial representation of the chains in the parent-child hierarchy abstractly looks like a pyramid. A pyramid contains four types of nodes: *procedure activation nodes (PANs)* represent activations of modification procedures from user transactions or from rule applications; *rule application nodes (RANs)* represent applications of candidate rules from procedures; *tuple instance nodes (TINs)* represent tuples in the database that satisfy conditions; *primitive nodes (PINs)* represent doit-actions and i/o-actions.

The root of a pyramid is a PAN, representing the procedure activated for the modification requested by a user transaction. Each PAN has one or more children RAN nodes, representing the disjunction between the candidate rules of the PAN. All non-root PAN nodes occur in *chains* and represent req-actions.

A RAN is the condition choice point for a rule application. It has one child TIN for each tuple that satisfies the rule's condition. If the condition of a rule cannot be satisfied, then the RAN will not have any children. If the condition of a rule is the empty condition, the RAN will have exactly one TIN child which represents the empty tuple. All the TIN nodes of a RAN are in a disjunction, representing the tuple-oriented semantics of the rules.

A chain links a sequence of PANs and PINs to a TIN node. The first node of a chain is always a TIN node, and TIN nodes only occur as the first node of a chain. The chain's remaining nodes represent instances of the rule's sequence of actions.

A leaf in a pyramid is a node that has no children. TINs and PINs are always leaves, even though they may be linked in a chain. RANs whose condition cannot be satisfied by the database are also leaves. Since modification procedures must always have at least one candidate rule, a PAN will always have at least one child and will never be a leaf.

The execution pyramid for an activation may be infinite since modification procedures can be recursive. At first it might seem that the search space could be reduced without affecting the semantics by preventing multiple requests of the same modification with the same set of parameters. This is true only if the database state did not change between two identical requests. But since the database state may change between requests, and the success of a candidate rule depends on this state, we cannot perform such optimizations.

Execution Semantics

The execution of a modification procedure transforms some start state $X_s = \langle S_s, D_s \rangle$, where S_s is a substitution of values for variables and D_s is a database state, into some final state $X_f = \langle S_f, D_f \rangle$ where S_f is a superset of S_s, and D_f is obtained by performing some sequence of modifications to D_s. This sequence of modifications is determined by the execution of a root-node PAN for the start state according to the following semantics.

PAN: An execution of a PAN A_i for start execution state $X_s = \langle S_s, D_s \rangle$ is successful if there is a successful execution of one of its RANs for X_s. The resulting state X_f is the final state of exactly one of the successful RANs.

RAN: An execution of a RAN is successful for start execution state X_s if the RAN's condition evaluates to true and there is a successful execution of a chain of one of the qualifying tuples. The resulting state X_f is the final state of exactly one of the RAN's chains.

Chain: An execution of a chain $\langle TIN, A_1, ..., A_n \rangle$ for start state $X_s = \langle S_s, D_s \rangle$ is successful if each action A_i can be successfully executed for start states $\{X_{s_1} = \langle S_s \cup TIN, D_s \rangle;\ X_{s_i} = X_{f_{i-1}}\ for\ i > 1\}$ where $X_{f_{i-1}}$ is the final state of action A_{i-1}. The resulting state X_f is the final state of A_n.

Doit-action PIN: An execution of a doit-action PIN A_i for start execution state $X_s = \langle S_s, D_s \rangle$ is always successful and results in the final execution state $X_f = \langle S_s, D_f \rangle$ where $D_f = A_i$ applied to D_s.

Read i/o-action PIN: An execution of a read i/o-action PIN for start execution state $X_s = \langle S_s, D_s \rangle$ is always successful and results in the final execution state $X_f = \langle S_f, D_s \rangle$ where $S_f = S_s \cup$ *result of read*.

Write i/o-action PIN: An execution of a write i/o-action PIN for start execution state X_s is always successful and results in the final execution state X_s.

Unsuccessful executions of RANs, PANs and chains do not modify the database.

The successful executions of an activation for an initial database state D can be visualized as sub-pyramids of the activation's corresponding execution pyramid, called *solution pyramids*. Figure 4.2 shows a solution pyramid for the top PAN. Solution pyramids have the following properties:

1. each PAN node has exactly one child RAN,

2. each RAN node has exactly one child TIN,

3. all leaf nodes of the pyramid are either PINs or TINs,

4. any RAN in the traversal of a solution pyramid sees only the modifications to the initial database state made by the doit-action PINs preceding the RAN in the solution pyramid. That is, if (i) v is a tuple for any TIN in the subpyramid, (ii) $\langle s \rangle$ is the sequence of modifications for the doit-action PINs that precede the TIN in a left-to-right traversal of the subpyramid, and (iii) C is the condition of the rule of the TIN's parent RAN, then v must satisfy C evaluated on the database state D' that results from applying the modifications in $\langle s \rangle$ to the initial database state D.

5. the variable substitutions used at each node in the solution pyramid are consistent. That is,

   - the start substitution of a PAN represents the values specified by the activating request or the final substitution of the previous node if the PAN is in a chain,
   - the final substitution of a PAN augments its start substitution with the return values mapped from the final substitution of the PAN's chosen RAN,
   - the start substitution of a RAN maps the start substitution of the RAN's parent PAN to the variables used in the procedure,
   - the RAN's final substitution is just the final substitution of the last node in the chain of the RAN's chosen TIN,
   - each TIN augments its RAN's start substitution with the variable mappings that result from the selection of the TIN's tuple,
   - each action node in a chain has an start substitution equal to the final substitution of the node that precedes it in the chain, and
   - doit-action and write-action PIN's do not update the substitution, and a read-action PIN augments its start substitution with the values it reads.

A *sub-solution pyramid* for a non-root node N satisfies the above requirements for all children of N.

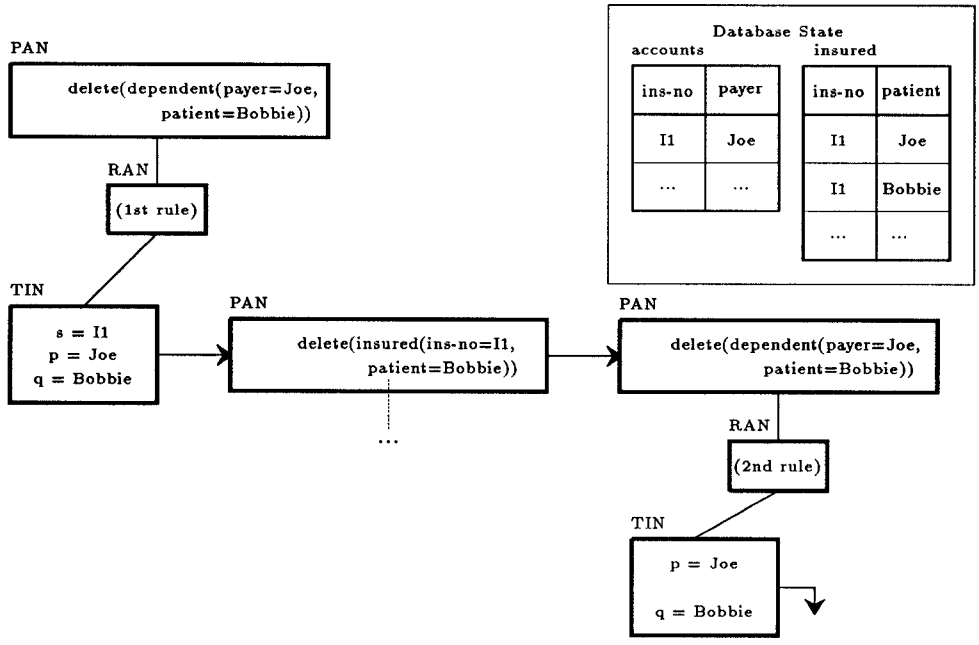


Figure 4.3: Sample Solution Pyramid

A successful execution of a modification procedure activation for an initial state D with start values V is equivalent to executing exactly one of the activation's solution pyramids given start state $\langle D, V \rangle$. Note that there may be several solution pyramids for a given activation of a modification procedure. Exactly one of these is chosen by the execution strategy as the solution pyramid. The resulting database state is the database state that results from applying modifications for the doit-action PINs to D according to the left-to-right sequence of the PINs in the chosen solution pyramid

Example 4.5 Assuming the modification procedure from example 4.4, Figure 4.3 shows a subset of a database and the corresponding solution pyramid for the database request `delete(dependent(payer=''Joe'', patient=''Bobbie''))`. Note that the two TIN nodes are leaves, and that the modification procedure
`delete(dependent(...))` is activated from both a user transaction and from within a candidate rule. ∎

In the remainder of this section, two tentative control strategies for executing modification procedures described in terms of the execution pyramid are explored. Both strategies involve

dynamically building solution pyramids, sometimes executing paths in the pyramid that must later be undone. The first strategy performs a depth-first traversal using a single process and backtracks when it discovers that it is building a non-solution pyramid. The second strategy concurrently executes possible solution pyramids, exploiting parallelism among all alternative child links.

In Appendix C, we show the correctness of each execution strategy by showing that, for a given activation A in some initial database state X, using execution strategy E

1. if there is a solution pyramid for A in X, then the execution of A using execution strategy E succeeds, and the resulting database state is the application of the modifications for the doit-action PINs to X according to the left-to-right traversal of exactly one of the solution pyramids.

2. if there is not a solution pyramid for A in X, then the execution of A using E fails, and the resulting database state stays the same.

4.4 A Depth-First Execution Strategy

Like all depth-first search techniques, the depth-first execution of a pyramid (PDFE) must maintain a stack of choice-points and return control to the most recent choice-point whenever a node fails. There are two types of choice-points for pyramids: the selection of a RAN for a PAN and the selection of a TIN and its associated chain for a RAN. The selection of a RAN represents the selection of a rule to apply for a given activation of a procedure. The selection of a TIN and its associated chain represents the selection of a substitution for the variables in the rule's condition.

What makes PDFE unique is the existence of chains in the pyramid. A PDFE of a chain must maintain the sequential order between the nodes in the chain. Since the variable substitutions for nodes in the same chain must be consistent in a solution pyramid, a PDFE must also maintain the left-to-right dependency of nodes in the chain on the current variable substitution and the database state. When a PDFE backtracks, it must undo any variable mappings and database modifications that occurred after the backtrack point.

PDFE begins at the root PAN. At a PAN, PDFE chooses a RAN to investigate, initializing the RAN's variable substitution according to the attribute values specified by the activating request.

A PDFE of a RAN involves choosing one of the RAN's TINs. TINs represent mappings of values to variables. Only a subset of a RAN's TINs are applicable for a given instance of the rule application since several of the TINs' mappings may not be consistent with the current substitution and the current database state. So, PDFE chooses one TIN that represents a mapping which is consistent with both the RAN's current substitution and the database state. The RAN's substitution is updated with any new variable instantiations introduced by the chosen TIN.

A PDFE must then try to successfully execute the TIN's chain. To successfully execute a chain, the sub-pyramids rooted at each node in the chain are executed in the order in which they occur. After each execution of a node, the RAN's substitution S is updated to reflect any variable mappings obtained by the node. If the node is an i/o-action PIN, all external reads must be reflected in the RAN's substitution. If the node is a doit-action PIN, then the modification is performed on the database. If the node is a PAN, then all uninstantiated variables in activating request will have a value upon the successful completion of the PAN, guaranteed by procedure safety (Section 4.2.5).

If PDFE successfully reaches the end of a chain it returns success to the RAN that is the parent of the chain's TIN. The RAN, in turn, returns success to its parent PAN, passing along appropriate return values for the variables in the PAN's modification request. If this PAN is the member of a chain, PDFE updates the substitution of the chain's RAN with the appropriate return values, and continues to process the next node in the chain. If the PAN is the root, then the search terminates successfully.

Since pyramids can potentially be infinite structures, PDFE may get stuck searching an infinite alternative. This can be prevented by imposing a bound for the maximum height of the search pyramid. Whenever this height is reached, PDFE assumes that the current node fails and backtracks to the most recent choice-point.

A PAN fails when all of its RANs fail, reflecting that a procedure activation fails if none of the procedure's rules can be successfully applied given the input attribute values and the database state. A RAN fails when it is impossible to successfully complete at least one of the TINs' chains, reflecting that there are no tuples that satisfy the rule's condition, are consistent with the rule's substitution, and for which a successful execution of the rule's actions can be found. A chain cannot be successfully completed when there is not a successful PDFE of the chain under the rule's substitution appended with the substitution implied by

the chain's TIN. Since PINs never fail[4], a chain fails only when one of its PANs fails.

Failures are always first detected at a RAN for which no TINs are consistent with the current variable substitution and database state. When this failure happens, the search must find another substitution by chosing a different TIN for a previously searched RAN or a different RAN for a previously searched PAN. In a PDFE strategy, this is done by backtracking to the most recent such choice-point. When this occurs, all database modifications and mappings of values to variables that occurred after the choice-point must be undone.

To enable backtracking, a state for all choice-points is maintained in a stack. Each state is a quadruplet $\langle N, S, D, T \rangle$ defined as follows:

N the node in the pyramid where the execution should resume.

S the substitution of values for variables before N is applied. This substitution is affected by the external activation, TINs, external reads, and the activation of other modification procedures. We assume the proper handling of scope as is typical in all procedure oriented languages.

D the depth of the node in the pyramid.

T the timestamp assigned by the database when the state enters the stack.

The stack is initialized with the state $\langle$`FAIL, INPUT, 0, STARTTIME`$\rangle$ where `INPUT` is the input substitution from the user and `STARTTIME` is the timestamp at the time the user request occurs. Assume that `SUB` is the current substitution, `TS` is the current timestamp assigned by the database, and `D` is the depth of the node in the current solution pyramid. Whenever a PAN is visited, the search pushes the state $\langle$`RAN, SUB, D, TS`$\rangle$ onto this stack for each of the PAN's RANs (except the chosen RAN). Similarly, whenever a RAN is visited, the search pushes the state $\langle$`TIN, SUB, D, TS`$\rangle$ onto the stack for each of the RAN's TINs (except the chosen TIN) that is consistent with `SUB` and with the database state at time TS.

When the search encounters a RAN that does not have any TINs that are consistent with the current substitution and database state (i.e. when it detects a failure), it pops the top node $\langle N, S, D, T \rangle$ from the stack. It resets the current substitution to S, the current node to N, the current depth to D, and rolls the database back to the timestamp T. A RAN fails if (1) there are no TINs that satisfy its condition on the database state and substitution or

[4]Ignoring, for the purpose of this description, timed-out reads.

(2) all the TINs that it pushed on the stack have been popped and searched, and the chain for the last TIN it popped fails. A PAN fails when all the RANs that it pushed on the stack have been popped and investigated, and the last RAN it popped fails. A chain fails when its first PAN fails. Concurrency control for PDFE is discussed in Section 4.7.

Theorem 4.6 (Correctness of PDFE) PDFE correctly executes a procedure activation A in any initial database state X, if there is a solution pyramid with height less than the maximum height bound. The proof is given in Appendix C.

4.5 Concurrent Execution Strategies

In this section we describe a concurrent execution strategy for pyramids, PCE. Several research efforts in logic programming utilize concurrent execution to improve the response time of a query [Sha87, KKM83]. Other efforts [Wol88, WS88, CW89, Don89, WO90, GST90] study Datalog [Ull88, Chapter 4] parallelization, which involves bottom-up evaluation of Datalog queries. However, the paradigms investigated in these efforts do not consider the behavior of modifications in the concurrently executed substructures. Our main interest in describing a concurrent execution strategy for pyramids is to investigate the issues involved in supporting concurrent execution in a multi-user database system. In such systems, concurrency control must be provided for any set of transactions that are simultaneously executing. Concurrency control issues are discussed later in Section 4.7.

Concurrent execution can be applied only to a restricted set of modification procedures. I/o-actions implicitly interface with a sequential medium (either a file or a terminal) and cannot be executed in parallel without confusion. Hence, concurrent execution strategies can only be applied to procedures that do not activate i/o-actions. Such procedures cannot contain i/o-actions in their rules, and they can only contain modifications that do not activate i/o-actions.

An execution strategy can use concurrency at any node whose children represent independent parts of the structure. In pyramids, every parent-child hierarchy represents a disjunction in which the siblings are not dependent on each other. Hence, there are two dimensions of concurrency corresponding to the two types of parent-child relationships:

rule concurrency exploits the disjunctive relationship between the sibling RANs of a PAN, indicating that an execution strategy can simultaneously (but independently) try to

apply all the rules of an modification procedure to a given activation.

tuple concurrency exploits the disjunctive relationship between the sibling TINs of a RAN, indicating that, for a given application of a rule, an execution strategy can simultaneously try to apply the actions of the rule to each tuple that satisfies the rule's condition.

It is no coincidence that these dimensions of concurrency are identical to the choice-points for PDFE since a concurrent execution that exploits both forms of concurrency is equivalent to a breadth-first execution that uses multiple processes.

As with PDFE, a concurrent execution strategy for pyramids has some unique requirements. Since pyramids contain actions that modify the database (which is shared by all processes), a concurrent execution strategy must ensure that the database modifications performed by concurrent processes are mutually exclusive. For the remainder of this section, we assume that processes that are concurrently executing subpyramids of disjunctive sibling nodes do not see each other's modifications and a parent only sees the modifications of its chosen successful child. Mechanisms for obtaining this mutual exclusion are discussed later in Section 4.6.

A concurrent execution strategy must also maintain the left-to-right dependencies between the nodes in a chain that arise when nodes modify either the database or the current variable instantiation. These dependencies are maintained if the nodes in a chain are executed sequentially, and the current variable instantiation is passed into and updated by each node of the chain as it executes.

Since we are only interested in finding one solution pyramid, the concurrent execution strategy for pyramids described in this section is optimistic but cautious. Whenever a successful child is found, its parent optimistically assumes that the child is its one and only child in the solution pyramid. Being somewhat cautious, the parent suspends, rather than terminates, all concurrent executions of the siblings of the chosen child. In addition, the successful child suspends itself. If the execution later discovers that the chosen child does not lead to a solution pyramid, the parent can resume the suspended executions. Furthermore, since there may be more than one instance of a node that leads to a solution, the chosen child's execution is also resumed. Because executions are suspended and resumed, our execution is not entirely concurrent and must use some backtracking.

When a concurrent execution of a pyramid encounters a parent-child hierarchy, it spawns

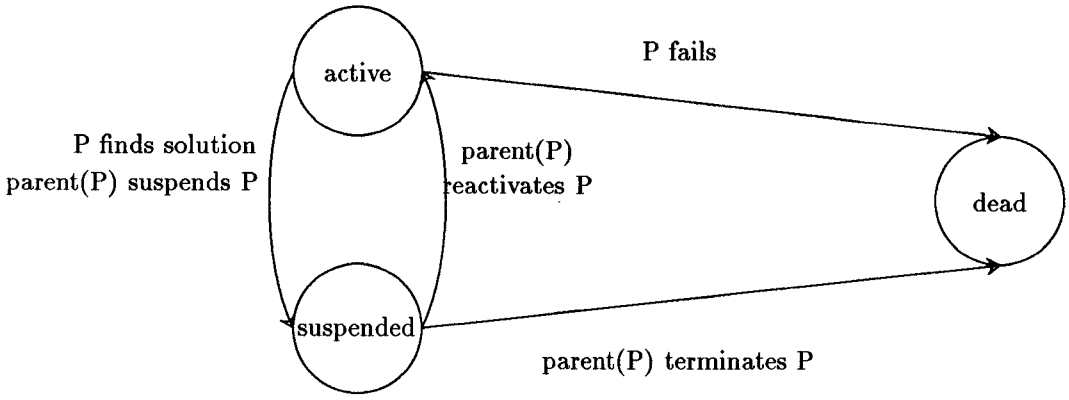


Figure 4.4: Execution States of Process P

a process to independently execute each of the children. There are three types of execution processes used: two corresponding to the two types of concurrency, and one for the root PAN node. *Chain processes* sequentially execute the nodes in a chain for a given TIN. They are spawned by RAN processes and initiate all backtracking for the concurrent execution. *RAN processes* execute RANs and are spawned either by the root PAN process or a chain process.

Before describing PCE, it is helpful to analyze the possible execution states of a process. A process is in one of three states as depicted in Figure 4.4, which shows the states for a process P with parent $parent(P)$. When a process is spawned, it is initially *active.* An active process executes partial-solutions to its assigned portion of the execution structure. If it finds a solution, it returns this solution to its parent and suspends itself - i.e. its state becomes *suspended.* An active process also becomes suspended if some other child is chosen as its parent's solution child, in which case it is suspended by its parent. A suspended process is known to the system and can become active again at any time; it is reactivated by its parent. Before a process is suspended, it must return a *resume state* to its parent that contains the information that is needed to resume its execution. A process is *dead* when it no longer has the potential to find more solutions. This happens when it has exhausted all possible executions or when it is terminated by its parent.

Having introduced these states, we describe the relationship between processes and their states. Every process except the root process has a parent process. The parent of a chain process is a RAN process, and the parent of a RAN process is either a chain process or the

root process. The relationship between the possible states of the nodes in a hierarchy exhibit the following properties: (1) all descendents of a suspended node are either suspended or dead, (2) all descendents of a dead node are dead, (3) a parent of an active node is always active, and (4) descendents of an active node can be either active, suspended, or dead.

We now describe the basic execution of a PCE. It begins at the root node PAN of the pyramid. This root process spawns a RAN process for each of the PAN's RANs. The input for each spawned RAN process is the substitution of values for variables that represents the user-activated PAN's input parameters. If all of the RANs fail, then there is no solution pyramid and the user request is rejected. If one of the RANs succeeds, a solution pyramid has been found – the user request is satisfied, and the successful RAN's substitution is used to synthesize the return values of the root PAN.

Each RAN process initializes its substitution to the input values supplied by its parent process, which is either a chain process or the root process. Given this substitution, it then evaluates its condition against the database. It spawns a chain process for each qualifying tuple, passing in the substitution. If all the chains fail, then the RAN reports failure to its parent process and dies.

A solution for the RAN has been found if one of the RAN's chains succeeds. The RAN immediately suspends all spawned chain processes that have not yet failed. It builds a RAN resume state consisting of the set of chain resume states, one for each suspended chain, plus the resume state for the chosen solution chain. It reports success to its parent process, returning as output both the RAN resume state and the substitution of the successful chain. It then suspends itself so that it can be reactivated if its parent chain process is forced to backtrack at some later time.

A chain process tries to find a complete chain for its TIN by sequentially executing the PANs and PINs of its chain. It maintains a substitution that is initialized by its TIN and the input substitution from its parent RAN. If it can successfully execute all the nodes in its chain, it has found a successful execution. It reports success to its parent RAN process, returning the final value of its substitution and its resume state. This resume state contains a *resume stack* which is used during backtracking. It is empty when the process is initially spawned and maintained during the execution and reactivation of the chain.

Since concurrent execution is not performed on pyramids that contain i/o-actions, the only type of PIN node that a chain process must execute are doit-actions. The chain process adds an undo-action for the PIN to the resume stack that, when executed, undoes the

effects of the PIN. Although doit-actions always succeed, they may be blocked by another transaction or another concurrent execution within the same transaction. In this case, the chain process must wait. However, this is not considered a failure since the chain process will either eventually be allowed to proceed or, in the rare event that a deadlock, crash, or media failure occurs, it will be rolled back and restarted by the database concurrency control manager or the mechanism that provides mutual exclusion for the concurrent execution.

A chain process executes a PAN node by spawning a RAN process for each of the PAN's RANs. If one RAN process returns successful, the chain process optimistically assumes that this RAN and its return substitution will lead to a successful completion of the chain.[5] The chain process suspends all of the PAN's spawned RAN processes that have not yet failed. It pushes a PAN resume state on the stack. This resume state consists of the set of RAN resume states returned by the suspended RANs, a RAN resume state for the successful RAN, and the current substitution. It then updates the current substitution according to the return values synthesized from the successful RAN's substitution.

As in PDFE, pyramids can potentially be infinite structures, and PCE may get stuck searching an infinite alternative. This can be prevented by imposing a bound for the maximum height of the search pyramid. Whenever this height is reached, PCE assumes that the current node fails and backtracks to the most recent choice-point.

Backtracking occurs in a chain process when all RAN processes for the PAN return failure. When this happens, the process pops and performs all undo-actions that are on the top of the resume stack until the stack is empty or until a PAN resume state is found. This restores ("rolls") the database state back to the backtrack point. If the backtrack stack is empty, then the chain process has failed and unsuccessfully returns to its parent RAN process. Otherwise, it pops the PAN resume state off the stack, sets the current substitution to the substitution contained in the resume state, and reactivates all the RAN processes whose RAN resume state is in the PAN's resume state.

At any point, a parent process can suspend any of its active subprocesses. When a RAN process is suspended by its parent chain process, it, in turn, suspends all its spawned chain processes that have not yet failed. It builds a RAN resume state consisting of the set of chain resume states returned by each suspended chain. When a chain's parent RAN process asks it to suspend itself, it suspends all currently spawned RAN subprocesses that

[5] At this point we could have chosen concurrently investigate all possible correct completions of the chain for each RAN process that returns successfully.

have not yet failed. It pushes on the resume stack a PAN resume state that contains the current substitution and a set of the RAN resume states returned by the suspended RANs. It returns to the parent RAN a chain resume state that consists of the resume stack.

PCE depends on the ability to resume processes when RAN and chain processes are reactivated. A RAN process resumes its execution by resuming all suspended chain processes in the RAN resume state. A chain process resumes its execution by initially backtracking. Its resume state contains a resume stack whose top element is a PAN resume state that contains the resume states for all RANs that were currently executing when the chain process was suspended. This element is popped from the resume stack, the chain's current substitution is re-initialized to the substitution contained in this PAN resume state, and all the RANs that have a RAN resume state in the PAN resume state are reactivated.

When a solution for the user request is found, there must be a clean-up procedure that terminates and undoes any database updates performed by all the unchosen suspended processes. There is no such procedure required when the user request fails since a failure implies that all sub-processes have failed.

It is possible that a more realistic execution strategy would exploit only rule concurrency and not tuple concurrency, since the expensive computation is the evaluation of conditions against the database. In such a strategy, a separate RAN processes would be spawned for each RAN of a PAN, but the RAN process would sequentially process each TIN and the TIN's corresponding chain. In such a strategy, the functionality of chain processes would be absorbed by the RAN processes.

Theorem 4.7 (Correctness of PCE) PCE correctly executes a procedure activation A in any initial database state X, if there is a solution pyramid with height less than the maximum height bound. The proof is given in Appendix C.

4.6 Mutual Exclusion for Concurrent Execution

The concurrent execution strategy described assumes that the database modifications of concurrently executing processes are mutually exclusive. In this subsection we briefly suggest two ways in which such exclusion can be obtained.

4.6.1 Nested Transactions

The first solution treats the concurrent processes as nested transactions [RM89, Mos85, Mos87, HR87] that compete with each other for access to the database. However, the locking rules for nested transactions do not directly apply. The rules for releasing and inheriting locks differ due to the behavioral differences between a failed process and an aborted process. A *failed process* is one that aborts itself because it detects something in the database state or the user input that indicates that it should not proceed.[6] An *aborted process* is one that is aborted by the system because of a deadlock or fails due to a system crash or a media failure.

The modified locking strategy is as follows. S-locks are obtained when a RAN process evaluates a condition, and X-locks are obtained when a chain process executes a doit-action PIN. Locks can only be obtained if the only other processes that hold the lock are ancestors of the requester. Otherwise, the requester must wait. When a subprocess is chosen as the successful child of its parent, the parent inherits all the locks of the subprocess. The subprocess releases all its locks, but keeps a record of them to use if it is reactivated during backtracking. All other suspended processes hold their locks until they succeed and are chosen or until they are terminated.

When a subprocess fails, its S-locks must be inherited by its parent process. They can only be released when a solution has been found for the root process. If no such solution exists, they must be held until the user transaction commits. However, the X-locks need not be inherited. These issues are further discussed in Section 4.7. The failed subprocess releases all its locks and terminates. When backtracking occurs, suspended processes are reactivated as described before with the exception that the previously chosen child first reclaims the locks it previously held from its parent.

The advantages to this mechanism are that it can use the locking mechanisms provided by the underlying database system and it does not require any copying of page-tables or data items. However, this solution could potentially cause severe blocking, reducing concurrency to a point where the execution is effectively being done sequentially. Furthermore, it is possible for a suspended process to hold a lock that is needed by an active process, blocking the active process indefinitely. If the active process is the pyramid's only opportunity for success when the suspended process is not chosen, a deadlock occurs. The suspended process

[6]This is referred to as program-enforced abort in [HR87].

will not release a lock until the execution fails and backtracks. But the execution will not fail and backtrack unless the active process obtains the desired lock. Therefore, in order to make this solution to mutual exclusion usable, no nodes can be suspended. When a solution for a RAN or PAN node is chosen, all competing processes must be terminated rather than suspended. If the execution backtracks to the RAN or PAN at some future point in the execution, all of the terminated processes must be restarted; all of the previous work performed by these processes is lost.

4.6.2 Shadow Paging

The second solution uses techniques similar to shadow-paging. Each spawned process records its modifications locally. These modifications are seen by all of the process' subprocesses, but are propagated to its parent only if the process is chosen as the parent's successful child.

The advantages of this solution are that it maximizes concurrency and does not redo any work. When necessary, previous computations of processes are reused. The problem is that it requires exponential overhead in copying and propagating database states between processes.

Obviously, the entire database need not be copied each time a process is spawned. Furthermore, chain process are the only processes that modify the database, so modification need only be maintained and propagated by chain processes.

Each chain process maintains a local *delta-page* table that reflects any changes that have been made to the database since the root PAN was activated. If the chain's grandparent is also a chain, then its delta-page table is initially the value of its grandparent's page table. Otherwise, the grandparent is the root and the delta-page table is initially empty.

When a chain process chooses a successful RAN for the execution of a PAN, it replaces its delta-page table with the delta-page table of the chosen RAN's chosen chain. When a chain process executes a doit-action PIN, it must make this modification visible to itself and all processes it subsequently spawns, but invisible to all concurrent executions. If the modification is to be made to a logical page $LP1$ that has been added to the chain's delta-page table since the beginning of the chain, then the modification can be made directly to the page referenced by $LP1$. Otherwise, the physical page $P1$ that is pointed to by $LP1$ is copied into $P1'$. $LP1$ is modified to reference $P1'$ and added to the chain's delta-page table. The modification can then be performed on the local copy of $LP1$.

In the concurrent execution strategy described previously, a doit-action PIN must be undone when a chain backtracks over the PIN. However, the delta-page tables never need to be modified during backtracking; nor do they need to be saved with each PAN resume-state in the chain's resume-stack. When backtracking, a chain can only be successful if some PAN is found that has an alternative solution RAN. When this happens, the chain's delta-page table is replaced with that of the new solution RAN's chosen chain.

When a successful execution is found for the root, then the modifications in the delta-page table for the chain of the successful RAN are applied to the database. All copies of pages for terminated processes should be garbage collected.

4.7 Concurrency Control Issues

The Update Dependency Language is executed in a subsystem of a traditional DBMS and any activation of a procedure is part of some user transaction, referred to as the *top-level transaction.* Hence, a procedure activation executes concurrently with other database operations and other procedure activations. For the most part, concurrency control can be correctly handled by applying the existing mechanisms provided by the DBMS. We assume support for basic two-phase locking [EGLT76], in which locks are obtained on data items that are read or written. Concurrency control for a sequential execution strategy, such as PDFE described in Section 4.4 is the most direct application of the DBMS locking mechanism. S-locks are obtained for all tuples accessed when computing the conditions of RANs and X-locks are obtained for all tuples modified by doit-action PINs.

The concurrency control mechanisms for the concurrent execution strategy described in 4.5 depend on the type of mutual exclusion mechanism used. If the subprocesses are run as nested transactions, then the locks are obtained as described previously. If the shadow-paging technique is used, then locks can be obtained using the DBMS lock manager. However, so that the subprocesses do not conflict with each other, all locks are obtained on behalf of the top-level transaction. S-locks are obtained from the DBMS as the logical pages are read. However, X-locks are only obtained when a solution to the root process is found and the modifications are actually made to the DBMS. They are not obtained when the modifications are made to the subprocess' local pages. Hence, only modifications that persist obtain X-locks, and all X-locks are obtained together.

It may seem that this delay in obtaining X-locks may not guarantee serializability. But

serializability can only be affected if a data item that is read by the execution is subsequently written by another transaction before the execution completes. This will not happen since the S-locks are obtained as the conditions are computed. If another transaction is reading or writing a data item that is to be written by the execution, then the execution must wait until the transaction releases the lock on the item.

Early release of locks

As described in Section 4.3, not all nodes in a pyramid participate in solution pyramids; furthermore, only the effects of one solution pyramid persist after the procedure activation completes. Therefore, there may be several data items that are read or modified at nodes that do not participate in the selected solution pyramid. Consequently, there may be some data items that are S-locked or X-locked by the top-level transaction only at these nodes. We shall refer to such locks as *false* locks. It may be possible to improve concurrency by releasing false locks before the top-level transaction commits, as suggested for partial rollbacks [MHL+91]. In some cases, this can be done without jeopardizing serializabily.

To simplify the descriptions in this section, we make the following observations a priori. We do not consider locks obtained prior to a procedure activation or from some previously visited node in the solution pyramid false locks. Also, the release of an X-lock that is only used by nodes that are not in the solution pyramid is obtained by escalating existing S-locks is performed by demoting the lock back to an S-lock.

In the remainder of this section we prove that X-locks along failed paths can be released (after corrective actions for the failure have been performed) and how all locks obtained on the non-solution paths of a successful execution can be released (after clean-up procedures have been executed). However, S-locks must be kept until a solution pyramid is found or until the end of the transaction, since the search makes decisions based on the data items it reads (or does not read).

Theorem 4.8 (Early Release of X-locks on Failed Sub-pyramids) When an execution detects a failure, it can release (or demote to read) known false X-locks after corrective actions for the failed sub-pyramid have been performed without jeopardizing serializability.

Proof: The false X-locks protect data items that are only modified by the top-level transaction during an unsuccessful execution. If the execution is depth-first, a failure results in

a partial rollback to the database state prior to the most recent choice point. During this partial rollback, all database modifications that resulted from the doit-action PINs between the choice-point and the failure are undone. Similarly, if the execution is concurrent using locking for mutual exclusion, then the modifications are undone when the chain processes backtrack.[7] Hence, all modifications that occurred since the most recent branching node will never be performed on the database since they will not be part of the solution pyramid. In both cases, the modification will not be seen as part of the top-level transaction and will therefore not need to be serialized with other concurrent operations. □

The same principle does not, however, apply to false S-locks as shown by the following theorem.

Theorem 4.9 (Maintenance of False S-locks During Execution) False S-locks must be maintained for the duration of the execution of a modification procedure.

Proof: We prove this theorem by showing a counter-example. Consider a database schema that has one relation for each state that records information about the suppliers in that state. This schema may contain relations such as: `VaSupp(S#, ...)`, `MdSupp(S#, ...)`, ..., `CaSupp(S#, ...)`. Suppose that there is a view `LocShpmnt(S#, P#, Qty, Loc)` which records shipments whose supplier, `S#`, is local to the shipment's location `Loc`. Then the modification procedure may contain the following two rules that apply to shipments that are located in Washington, DC. (For simplicity, assume that all the actions of each of these rules succeed.)

```
insert LocShpmnt(S# = S, P# = P, Qty = Q, Loc = L)
-> L = ''Washington, D.C.'' and VaSupp(S#), ... .          (r1)
-> L = ''Washington, D.C.'' and MdSupp(S#), ... .          (r2)
```

[7] As described above, for concurrent executions that use shadow-paging techniques, locks are never obtained for these modifications since they are not part of the solution pyramid modifications that are performed on the database.

Consider the following database state and set of transactions.

```
DBstate:  supplier S1 is currently in MdSupp.

T1: tries to insert supplier S1 as a local supplier for a shipment
    which is located in Washington

T2: modifies the database to reflect the fact that S1 has
    moved from Maryland to Virginia with the following two actions:

    (1) deletes S1 from MdSupp
    (2) inserts S1 into VaSupp
```

Any serial execution of these two transactions will result in the granting of T1's request. If T1 is executed before T2, then the request will be granted by a successful application of rule r2. If T2 is executed before T1, then the request will be granted by a successful application of rule r1. Now, suppose false S-locks can be released on a failed sub-pyramid when a failure is detected. The following sequence of events shows a concurrent execution that can occur in this case but does not correspond to a serializable schedule.

```
(1) T1:  tries to insert supplier S1 as a local supplier for a
         shipment which is located in Washington

(2) T1:  activates procedure ''insert LocShpmnt''

(3) T1:  tries r1; r1 fails since S1 is not in VaSupp

(4) T2:  delete S1 from MdSupp

(5) T2:  insert S1 from VaSupp
         /* T2 could not do this if the S-lock on VaSupp was
            kept */

(6) T2:  commits and releases all locks

(7) T1:  tries r2; r2 fails since S1 is not in MdSupp
```

This execution results in the rejection of T1's request; it is not equivalent to either of the

serial executions. Hence, S-locks cannot be released early. □

The above non-serializable schedule occurs because the execution makes decisions based on the data items it reads (or does not read). If these false S-locks are released before a solution pyramid is chosen, other transactions can modify data items that would have otherwise been locked. These modifications could potentially cause some of the rejected sub-pyramids to be successful. However, the execution will not reconsider the sub-pyramids and may incorrectly reject a valid modification.

The problem is even worse if the other transactions also make modifications that make it impossible for the execution to find a solution pyramid. Now, this scenario is always possible since nodes can never be locked until they are executed. However, a correct transaction will likely only invalidate a solution pyramid if it makes modifications that create new solution pyramids. So, it is possible for a concurrent execution of a set of transactions to give a value for the success or failure of a procedure activation which could never be obtained with a serial execution of the same set of transactions.

Since the update dependency procedure and the user transaction may make database modifications based on the result of an procedure activation, this type of behavior is not desirable. It may lead to concurrent executions that are not serializable as shown in our proof.

If S-locks are obtained until the end of the procedure execution, this behavior does not occur. Suppose transaction T replaces one correct solution pyramid with another (like transaction $T2$ above). Suppose transaction S is executing a pyramid. If S keeps its false S-locks, the locks prevent T from making solution pyramids that include the locked nodes. If T creates new solution pyramids before invalidating old ones, then T is blocked until S completes. If T invalidates the old solution pyramids first, then, in the worst case, a conflict between T and S will result in a deadlock, which will be resolved by the underlying lock manager. In the above example, `T2` would wait at step `5` until `T1` completes.

Since false S-locks keep other transactions from updating the database in such a way that makes the decision to reject a path incorrect, they should be released only when a solution pyramid is found for the user-activated modification procedure. If there is no solution pyramid, all S-locks must be kept until the top-level transaction commits or aborts.

Recall from Section 4.3 that, to the user transaction, the sequence of actions and queries that a successful modification procedure activation performs corresponds to a left-to-right

traversal of the leaf nodes of the chosen solution pyramid. TINs represent database reads, and PINs represent either database modifications or i/o-actions. Therefore, locks obtained on the solution pyramid must behave as if they were obtained directly by the transaction and not through the activation. However, the false locks can be safely released.

Theorem 4.10 (Release of All False Locks When Solution Found) When a solution pyramid is found for a user request (i.e. a root node PAN), both false S-locks and false X-locks can be released.

Proof: In Theorem 4.8, we showed that false X-locks on failed sub-pyramids can be safely released when a failure occurs. For a similar reason, false X-locks along unchosen paths, which occurs only for concurrent executions, can be safely released. The updates protected by the locks are either never performed or are undone by the clean-up procedures.

All false S-locks an also be released when a solution is found. Suppose the release of such a lock causes a non-serializable schedule. Then the value v of the data protected by the lock must be used to either to update some other data value or affects the decision of some later modification procedure. The only modifications that can use v are also on unchosen sub-pyramids and are undone as previously described. Furthermore, the definition, executions of modification procedures base all decisions on the current database state and input substitutions. Hence, v cannot possibly affect the execution of later modification procedures. □

In summary, for procedure activations that occur in a DBMS that supports two-phase locking, false X-locks can be released as soon as they are detected, and false S-locks can be released only when the request is granted.

4.8 Summary

In this chapter, we formally defined the Update Dependency Language and demonstrated its use with numerous typical database examples. We described two different strategies, depth-first and concurrent, for executing procedures of this language, and we developed new modified two-phase locking strategies with early write lock release for executing these procedures in a traditional database environment.

The Update Dependency Language has been used to specify interoperability in engineering information systems [RMSF91] and multidatabases [LMR90]. We have also implemented a prototype interpreter for the Update Dependency Language in Prolog, which is currently being used in a joint project with The Mechanical Engineering Department at Maryland [HM89]. The purpose of this project is to develop, validate and test operational specifications of a Computer Integrated Manufacturing system that integrates the CAD/CAPP/MRP II Systems. The mechanical engineers have written a large set of modification procedures, and have been able to specify all of the selected operations in their CIM system with only small amount of consultation from us.

We are currently approaching the implementation of this language from two angles. First, we are building an interpreter, UDappl, as an application which issues SQL queries and modifications to a commercial DBMS (we are currently using Oracle). The interface to UDappl will allow the user to issue modification requests in a form similar to the req-actions described in Section 4.2.2. This is the next generation of interpreter that will be used in the CIM project, and is useful because it is easily ported to any commercially available DBMS with an SQL query language.

Secondly, as described in this chapter, we plan to integrate a subsystem for executing modification procedures into a DBMS, such as ADMS [NR91]. This will allow the modification procedures to truly guard their relations and to be activated by set-oriented modifications which are typically issued from SQL. We will also be able to experimentally investigate the concurrency issues for the different control strategies described in Sections 4.3 and 4.7.

Chapter 5

Integrating Production Rules into the Starburst DBMS

5.1 Introduction

The Starburst Rule System [WCL91a] provides a mechanism for specifying and executing set-oriented production rules: rules that are triggered by, query, and perform arbitrary sets of changes to the database. The rule language is based on SQL, since an extended version of SQL is the query language used in Starburst [HCL+90]. The triggering of rules in this language is based on the notion of *transitions* [WF90]. A transition is a database state change resulting from the execution of a sequence of database modification operations. These rules consider the *net effect* of transitions, meaning that: (1) a tuple that is updated several times within a transition is considered as a single composite update; (2) a tuple that is updated and subsequently deleted in the same transition is considered as only a deletion; (3) a tuple that is inserted and updated in the same transition is considered as an insertion of the updated tuple; (4) a tuple that is inserted and subsequently deleted in the same transition is not considered at all. A formal presentation of transitions and their net effects appears in [WF89b].

Rules are activated at the *prepare-to-commit* point of each transaction. The sequence of modification operations that are performed by the transaction up to this point is the transaction's *external transition.* All rules that are triggered by this external transition are considered for execution at this time. As will be described in more detail, rules may issue further modification commands which may, in turn, trigger additional rules. Once there are no triggered rules left to consider, the transaction is committed.

The rule system has been implemented at the IBM Almaden Research Center as an

extension to *Starburst*, a prototype relational database system with a focus on extensibility [HCL+90]. This implementation fully integrates rule definition and execution with database processing, including features such as concurrency control and rollback. Furthermore, the rule system is transparent to those database tasks that do not trigger rules; if a transaction does not perform any operations that trigger rules it will not incur any overhead due to the existence of the rule system.

We begin this chapter with a brief overview of Starburst rule language and examples illustrating its features. Next, we describe the extensibility features of Starburst that were used in the implementation, followed by the general design of the rule system implementation as defined in [WCL91a]. We then present several aspects of the design in detail. First, the auxiliary main-memory structure used to detect rule triggering and to compute transition values is presented. This structure is unique to the Starburst rule language because of its fully set-oriented semantics and the need to compute the net effect of transitions. Second, the processing required to provide the ability to respond to rollback is discussed. Finally, we discuss the few provisions that are taken during rule execution to support the concurrent execution of transactions, all of which may independently trigger rules. [1]

5.2 Language Overview

A Starburst production rule is composed of three main components: a *transition predicate*, an optional *condition*, and an *action*. The transition predicate controls triggering. The condition is evaluated only if the rule is triggered: the action is executed only if the condition is satisfied, clauses for ordering rules are also included, and the syntax for creating production rules is:

create rule *rule-name* **on** *table-name*
when *transition-predicate* ,
[**if** *condition* ,]
then *action-list* ,
[**precedes** *rule-list* ,]
[**follows** *rule-list*] ;

[1] Several aspects of the implementation are not presented in this chapter either because they are not unique to the Starburst language or because they were fully designed and implemented by other team members. These include rule definition and storage, rule execution, rule ordering, and authorization. Rule ordering and concurrency control for rule definition are included in later chapters.

A rule is uniquely identified by its *rule-name*. A rule's *transition predicate* specifies its triggering operations; it is a nonempty subset of

(inserted, deleted, updated [(*column-list*)])

where *column-list* is a list of columns in *table-name*. A rule is triggered by a given transition if at least one of the specified operations occurred on *table-name* in the net effect of the transition. In the case of **updated**, one of the columns in *column-list* must be updated; if no columns are specified, the rule is triggered by updates to any column. Once a rule is triggered, its condition is checked. A rule condition is an SQL **select** expression: if the result of evaluating the **select** expression is nonempty then the condition is true. If the condition evaluates to true, then the rule's actions are executed in sequence. The condition may be omitted, in which case it always evaluates to true. Rule actions are arbitrary Starburst database commands, including **select**, **insert**, **delete**, and **update** expressions, as well as data definition commands and **rollback** requests. The optional **precedes** and **follows** clauses list existing rules and are used to specify rule priorities. If a rule R_1 includes a rule R_2 in its **precedes** list, then if R_1 and R_2 are both triggered, R_1 will be considered first; conversely for **follows**.

The condition and action parts of a rule may refer to the current state of the database through SQL operations. In addition, these components may refer to *transition tables*—logical tables reflecting the changes that have occurred during a rule's triggering transition. There are four transition tables: **inserted**, **deleted**, **new_updated**, and **old_updated**. A rule may refer to any transition table corresponding to one of its triggering operations. Consider a rule R on a table T. At the end of a transition triggering rule R:

- **inserted** refers to those tuples of table T in the current state that were inserted by the transition.
- **deleted** refers to those tuples of table T in the pre-transition state that were deleted by the transition.
- **new_updated** refers to those tuples of table T in the current state for which at least one of the triggering columns was updated.
- **old_updated** refers to those tuples of table T in the pre-transition state for which at least one of the triggering columns was updated.

Transition tables may be referenced in the **from** clauses of **select** operations using Starburst's *table expression* syntax as shown in this example:

select ... from ... v as (inserted()) ... where ...

In the **select** and **where** clauses, references to table variable **v** indicate transition table **inserted.**

Rules are activated at the prepare-to-commit point of each transaction executed by a user or an application program. During the transaction, a sequence of SQL commands generates a sequence of tuple-level insert, delete, and update operations, called the transaction's external transition. The state change resulting from this external transition may trigger some initial set of rules. One rule is chosen from this set for *consideration.* The rule is chosen such that no other triggered rule should precede it according to the rules' **precedes** and **follows** clauses. The chosen rule's condition is checked; if the condition is false then another triggered rule is chosen for consideration. Otherwise, the rule's list of actions is executed. Let R be the first rule whose actions are executed and assume for the moment that it does not specify **rollback.** At this point, one rule has been executed, although several rules may have been considered. All rules *not* previously considered are now triggered only if their transition predicate holds with respect to the composite transition created by the external transition and the sequence of modification operations that resulted from the execution of R's action. That is, these rules see R's action as if it were executed as part of the user-generated transaction. Rules already considered (including R) have already "processed" the external transition. Thus, these rules are triggered again only if their transition predicates hold with respect to the transition created by R's action. This is shown graphically in Figure 5.1. At the prepare-to-commit point, **A**, all rules are triggered relevant to the transition from **S** to **A**. At **B**, all rules considered at **A** are triggered relevant to the transition from **A** to **B**, and all other rules are triggered relevant to the transition from **S** to **B**. At **E** no rules are triggered and the transaction is committed.

Consider now an arbitrary point in rule processing, where zero or more triggered rules have been considered, and those whose conditions were true have been executed. A given rule is triggered at this point if its transition predicate holds with respect to the (composite) transition since the point at which it was most recently considered. If a rule has not yet been considered, then it is considered with respect to the transition since the start of the transaction. If a **rollback** action is encountered during rule execution, the system rolls

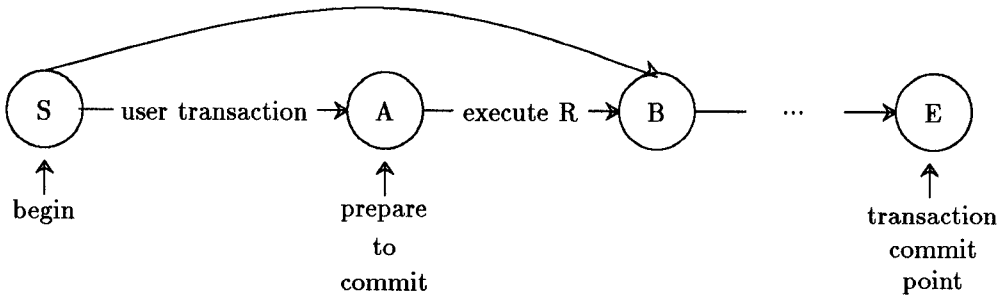


Figure 5.1: Transitions and Rule Triggering

back to the start of the transaction (including undoing all effects of previously executed rule actions) and rule processing terminates. Otherwise, rule processing terminates when the set of triggered rules is empty or when no triggered rule has a true condition; the entire transaction is then committed.

5.2.1 Examples

We illustrate this rule language with three simple examples; for numerous additional examples see [CW90, CW91, WF89b, WF90].

Example 5.1 The first rule controls salaries in a database of employees:

```
create rule sal_control on emp
when (inserted, updated(salary)),
if 'exists (select * from i as (inserted())
                    where i.salary > 100) or

    exists (select * from nu as (new_updated())
                    where nu.salary > 100)',
then ('update emp
         set salary = 50
         where emp.id in (select v.id from v as (inserted()))',
      'update emp
         set salary = .9 * salary
         where salary > 100');
```

This rule is triggered whenever employees are inserted or salaries are updated. The condition holds if any inserted or updated employee has a salary greater than 100. If true,

the action sets the salaries of all inserted employees to 50 and reduces each existing employee's salary by 10% if it is greater than 100. Notice that this rule triggers itself until all salaries are reduced to less than or equal to 100. ∎

Example 5.2 Suppose that in the extreme case, when an inserted or updated salary exceeds 150, the entire transaction should be rolled back. This is implemented by the following rule:

```
create rule sal_extreme on emp
when (inserted, updated(salary)),
if 'exists (select * from i as (inserted())
                    where i.salary > 150) or
    exists (select * from nu as (new_updated())
                    where nu.salary > 150)',
then 'rollback',
precedes sal_control;
```

Since both rules will be triggered at the same time, the rule specifies that `sal_extreme` is considered first, so that salaries greater than 150 are not simply reduced by rule `sal_control`. If `sal_control` is considered first, the condition of `sal_extreme` may never evaluate to true since it sees only the reduced salaries when the net-effect is considered. ∎

Example 5.3 As a final example, we show a rule that implements the cascaded deletion method of enforcing referential integrity. Whenever employees are deleted, the rule deletes all employees managed by the deleted employees:

```
create rule del-cascade on emp
when deleted,
then 'delete from emp
      where emp.mgr-id in (select v.id from v as (deleted()))',
precedes sal_control,
follows sal_extreme;
```

This rule has no condition, so its action is executed whenever it is triggered. The rule triggers itself, with termination occurring when no employees satisfy the predicate in the action, i.e., no deletions occur. For performance, this rule is specified to follow `sal_extreme`—if a transaction is rolled back by `sal_extreme`, then `del_cascade`'s deletes would be undone anyway. However, `del_cascade` is specified to precede `sal_control` since there is no need to adjust salaries for employees who will subsequently be deleted. ∎

5.3 Starburst

Starburst is a prototype relational database system being developed at the IBM Almaden Research Center. One of the goals in the design and implementation of Starburst is to make it an extensible system—a system that can support non-traditional applications and can serve as a testbed for innovations and improvements in database technology. For a detailed description of Starburst, its extensibility architecture, and some of its current extensions, see [HCL+90]. The Starburst Rule System is a substantial extension that takes advantage of several features included in Starburst for extensibility. Aspects of these features used in the rule system implementation are briefly introduced in this section.

The *attachment* mechanism in Starburst permits extensions to the system that require procedures to be called before and/or after each tuple-level database operation. A new *attachment type* is created by registering two sets of procedures: (1) procedures to create, drop and alter *instances* of the attachment; (2) procedures to be invoked before and after each tuple-level insert, delete, or update operation on a table with one or more attachment instances. A table may have many instances of a given attachment type, and instances of a given type may be created on any table. Each instance, however, is associated with exactly one table. Once an attachment type is established, instances of that type are created, dropped, and altered using general data definition commands provided by Starburst. For example, the **create rule** command defined in the previous section may cause either the creation or alteration of a rule attachment as will be described in the next section. An optional *attachment descriptor*, whose structure is defined by the attachment designer, may be associated with each attachment instance for examination and modification by attachment procedures. It is interesting to note that the Starburst attachment mechanism already provides the internal power of a tuple-oriented database rule system.

The Starburst query language, among other features, enhances SQL with *table expressions* [Dat84] and *table functions*. Table expressions are used to bind subqueries as input to operators, and table functions are used for function-generated tables. The table expression "*var* **as** *query*" binds the variable *var* to the subquery *query*. A table function is created by registering a function name, parameter specifications, a table schema, and a procedure for producing the tuples of the table function. Table expressions are used to reference table functions according the following syntax: "**from ... v as** (*fn-name*($p_1 \ldots p_n$)) ...". Appearances of table variable **v** elsewhere in the query are references to table function *fn-*

name. The table produced by *fn-name* at run time has the schema that was registered for the table function. To produce the table, parameters $p_1 \ldots p_n$ are passed to the table function's procedure. The procedure may perform any computations as long as it generates a set of tuples with the specified schema.

The *event queue* mechanism in Starburst is used to schedule the deferred execution of procedures. Once an event queue is declared, arbitrary parameterized procedures can be placed on the queue to be executed when the queue is invoked. Currently there are two built-in event queues: one for procedures to be executed during the *prepare-to-commit* phase of each transaction, and one for procedures to be executed upon actual *commit*. These queues also are invoked if a transaction rolls back; their procedures are passed a flag indicating that a rollback is occurring. Starburst permits *partial rollback*, whereby the process is rolled back to a user-specified save point within the current transaction. During partial rollback, all procedures placed on event queues during the portion of the transaction being rolled back are removed from the queue and executed with the "rollback" flag. Procedures may be placed on event queues with parameters that will then be available when the procedure is executed. Event queue procedures are executed in reverse order of arrival (i.e., the queue behaves as a stack).

5.4 Rule System Design

We briefly describe the overall design structure of the Starburst Rule System. Details of all components and specifics as to how Starburst's extensibility features facilitated their rapid development can be found in [WCL91a].

Figure 5.2 illustrates most of the rule system's execution modules and data structures, showing how they fit together and how they interact with Starburst. In the diagram, Starburst, its query processor, and its data repository appear on the left. The ovals in the center column indicate execution modules of the rule system. The rectangles on the right represent memory-resident data structures maintained by the rule system. An arrow from an execution module to data indicates that the execution module creates the data, while the reverse arrow indicates that the execution module uses the data. A double-headed arrow from one execution module to another indicates that the first module calls the second; the arrows are labeled by the event causing a call to occur. When these arrows pass through or originate from a star, this indicates that the call is made through an extensibility feature of Starburst,

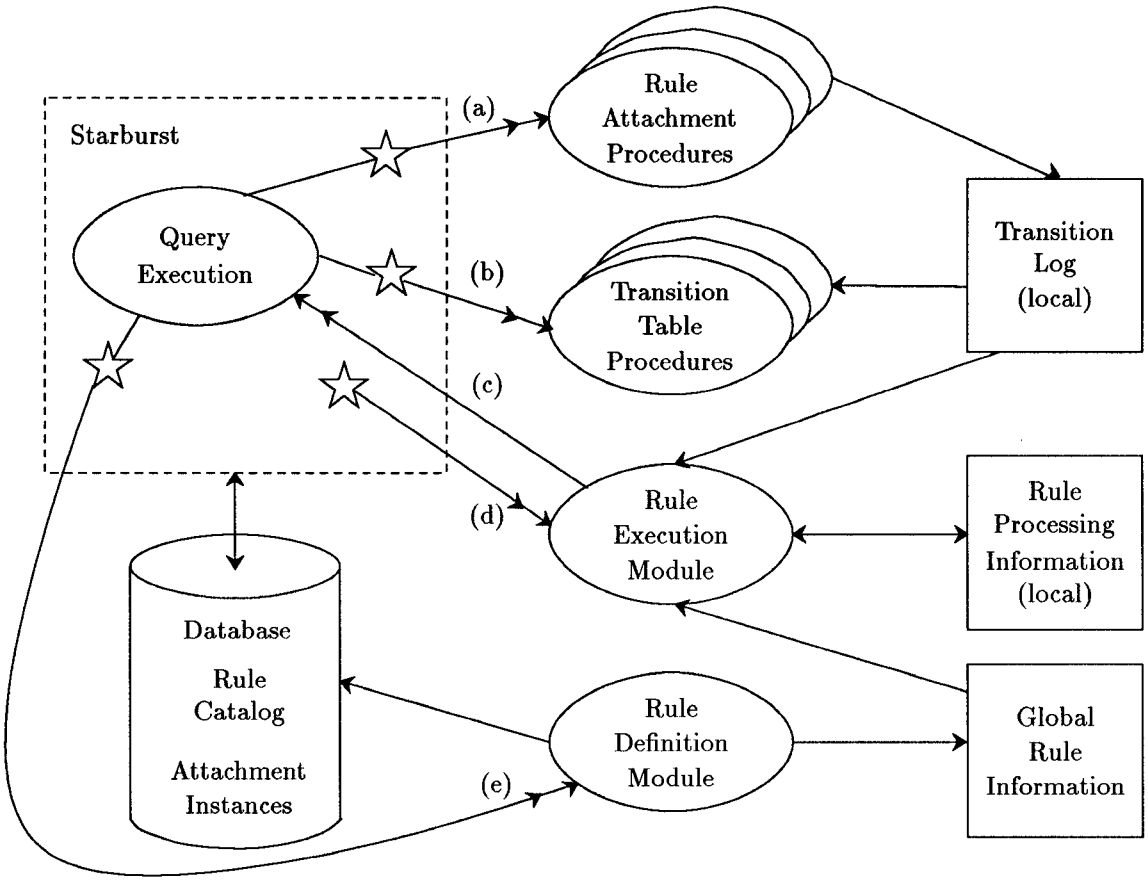
as explained in the caption.

The data maintained by the rule system can be divided into:

- *Rule Catalog*: This resides in the database and stores the set of currently defined rules.
- *Global Rule Information*: For efficiency, some information regarding the set of rules also is stored in main-memory, with the assumption that the number of rules does not exceed the capacity of (virtual) memory (see [HCKW90]). This information is shared by all user processes. Notice that not all information about each rule is kept in main-memory. When a rule is evaluated, its condition is checked and its actions are executed by fetching them from the Rule Catalog and calling the Starburst query processor.
- *Transition Log*: This is a highly structured log of those operations occurring within a transaction that are relevant to currently defined rules. This data structure is *local*, i.e., one Transition Log is maintained for each user process.
- *Rule Processing Information*: This also is local. It includes all information pertinent to executing rules within a given transaction, including which rules have been considered and when, and which rules are potentially triggered at a given point in time (*Potential-Rules*).

The execution modules depicted in Figure 5.2 are:

- *Rule Definition Module*: This component processes rule definition commands **create rule**, **drop rule**, and **alter rule**. It is responsible for maintaining the Rule Catalog and updating the Global Rule Information.
- *Rule Attachment Procedures*: This set of procedures writes to the Transition Log whenever relevant table modifications occur. A rule attachment procedure is called automatically whenever an insert, delete, or update operation occurs on a table with at least one rule.
- *Transition Table Procedures*: This set of procedures produces the transition tables that may be referenced in rule conditions and actions. At run time, these procedures produce transition tables one tuple at a time, extracting the tuples from the Transition Log.



Activating Event	Starburst Extensibility Feature ☆
(a) Table change	ATTACHMENTS
(b) Transition table reference	TABLE FUNCTIONS
(c) Rule condition or action	
(d) Prepare to commit	EVENT QUEUES
(e) Rule definition	ATTACHMENTS

Figure 5.2: Overall Structure of the Rule System

- *Rule Execution Module*: This component is responsible for selecting and executing triggered rules. It is invoked automatically at the commit point of every transaction for which a rule may have been triggered. The *Potential-Rules* structure is initialized with all rules that are triggered (without considering the net effect) by any operation that occurred during the initial transition. Subsequent rules are added to Potential-Rules at the end of each transition if they are triggered by the operations that occurred during the transition. Rules are selected from this structure, their triggering condition evaluated against the net effect, and their actions executed according to the execution semantics described in Section 5.2. When there are no triggered rules left to execute, the Rule Execution Module terminates and the transaction is committed.

There are several utility components of the rule system that are not illustrated in the diagram. A set of *System Start-Up* routines initialize the Global Rule Information from the Rule Catalog when Starburst is started or restarted. Similarly, a set of *Process Start-Up* routines initialize the local data structures required by each process for rule processing. In Starburst, a process may execute a sequence of transactions, so *Transaction Clean-Up* procedures are required to reset the local data structures at the end of each transaction. The *Rollback Handler* is the set of routines that prepare for and respond to unexpected partial or complete rollbacks. Details of this component appear in Section 5.6.

5.5 Transition Logs

Each process maintains a local *Transition Log* which is used during rule execution to determine which rules are triggered and to compute transition tables. The Transition Log for a given process contains information about insert, delete, and update operations occurring in the process' current transaction that are relevant to the currently defined rules. Each log entry, whose structure is shown in Figure 5.3, represents one tuple-level operation.[2] The `op-type`, `table-name`, and `tuple-id` are used to determine which rules are triggered. The `new-vals` and `old-vals` record new values and old values for a single modification and are used to compute transition tables. Fields `time-stamp` and `most-recent` are explained below.

Recall that the semantics of rule execution dictates that, at any given time, different rules may be considered with respect to different starting points: each rule is triggered with

[2]We show here only the fields relevant to the subsequent discussion.

```
op-type          /* type of the operation */
table-name       /* table-name of the operation */
tuple-id         /* tuple identifier of the modified tuple */
new-vals         /* tuple's new attribute values for insert
                    and update operations */
old-vals         /* tuple's old attribute values for delete
                    and update operations */
time-stamp       /* time-stamp of the operation */
most-recent      /* indicates if log is most recent log record
                    written for the tuple */
```

Figure 5.3: Structure of Transition Log Entry

respect to its *relevant transition* – the transition since the rule was last considered or since the start of the transaction if the rule has not yet been considered. Each transaction associates a *time-stamp* with each rule that it has considered. The relevant transition for a given rule is then constructed based on the `time-stamp` values of the log-entries: all entries with a `time-stamp` greater than the rule's time-stamp are in the rule's relevant transition.

The Transition Log is a "double hash table;" it is indexed by two hash tables: `OPTAB` hashes on the key `(op-type, table-name)` and `TABTUP` hashes on the key `(table-name, tuple-id)`. Each hash table entry is a list containing all log entries that share the same hash key values. The lists are kept in descending `time-stamp` order (i.e. most recent first). The Process Start-Up routines initialize the hash tables to empty, and the Transaction Clean-Up procedures reset them to empty at the end of each transaction.

Figure 5.4 illustrates a portion of a Transition Log with deletions, insertions and updates to table *T1*. The solid arrows represent the lists that link together log-entries with common `(op-type, table-name)`, and the dashed arrows represent the lists that link together log-entries with common `(table-name, tuple-id)`. Each log entry is in exactly two such lists. In the figure, the `op-types` I, D, and U refer to insert, delete, and update respectively.

If there is at least one rule defined on a table, then any modification to the table will trigger a rule attachment procedure. This procedure creates a new log entry with the relevant information about the modification, and inserts the log entry into both hash tables. If there is already a hash-entry for the log entry's hash key, then the new log entry is inserted in the front of the existing hash-entry's list (this maintains the descending `time-stamp` order).

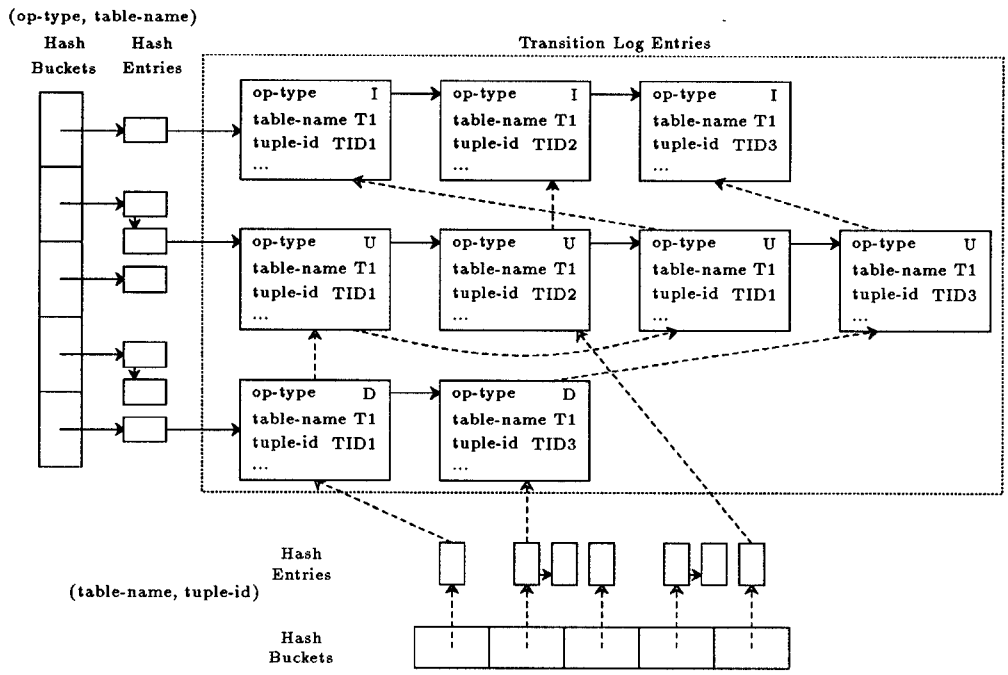


Figure 5.4: Transition Log

Otherwise, a new hash-entry is created, initialized to a list with the new log entry as its only element, and added to the appropriate bucket.

The Transition Log is used both to determine rule triggering and to compute transition tables. During rule execution, rule triggering is determined in two phases. First, all hash-entries in the `OPTAB` hash table are scanned to find all rules that are triggered by operations that occurred during the most recent transition, without considering the net effect. Each `(op-type,table-name)` combination found in `OPTAB` is used to select a set of rules from the Global Rule Information which is added to Potential-Rules.

The actual net effect is checked only when a rule is chosen from Potential-Rules for consideration (the second phase of determining rule triggering). The `(op-type, table-name)` of the rule is used to hash into `OPTAB` to find the corresponding list of log-entries. This list is scanned in descending `time-stamp` order until one log entry is found whose net effect has the same operation as *op-type*. The net effect for a log entry is computed by processing the history of the entry's tuple, found by hashing into `TABTUP` using the `(table-name, tuple-id)` values of the log entry. The computation of transition tables is similar to rule triggering, since, by definition, a triggering operation has occurred if and only if the corresponding

transition table is non-empty.

Not all tables will have rules defined on them, and those that do will likely not have a rule for each possible operation on the table. Furthermore, if the rules do not contain transition table references, there is no need to maintain `new-vals` and `old-vals`. Therefore, we can minimize the amount of information that is kept in the Transition Log for each table based on the triggering operations and transition table references of the rules defined for that table.

The logging requirements for the operations on a table are based on the combination of triggering operations (TOs) and transition table references (TTRs) for all the rules defined on the table. There are many different cases to consider (40): a rule can trigger on any combination of the three triggering operations, and if a given triggering operation occurs in a rule's transition predicate, then its corresponding transition table references can occur in its condition and action. However, rather than enumerating each case, we evaluate the logging requirements for the operations based on the logging requirements for the (TO,TTR) pairs shown in rows of Figure 5.5. The transition table reference "Λ" indicates that there are no transition table references that correspond to the triggering operation. The items in the table represent what information should be logged for a (TO,TTR) pair when a given database operation occurs. The *flg* field indicates whether the occurrence of the operation should be logged. The *old* and *new* fields indicate whether the `old-vals` and the `new-vals` should be recorded for a given occurrence of an operation. The symbol "$\forall$" in the *flg* field means that all occurrences should be logged; "$\emptyset$" means that no occurrence should be logged; a *C1* means that an occurrence should be logged only if the tuple of the operation was inserted previously in the current transaction. The interpretation of the symbols is similar for the *old* and *new* fields.

Notice that the occurrence of all insertions must be recorded if there is at least one rule defined on the table. Consider a simple case, when the triggering operation of the only rule defined on a table is "deleted". Recall that a deletion of a tuple is only considered a deletion in the net effect of a transition X if it was not previously inserted during X. This can only be determined if insertions are recorded. For a similar reason, occurrences of deletions must be recorded when the triggering operation of a rule defined on a table is "inserted". However, the deletions only need to be recorded if they delete a tuple that was previously inserted by the current transaction. Notice also that all new update values for inserted tuples must be recorded if the transition table **inserted()** is referenced, enforcing that the net effect of a tuple that is inserted and then updated is the insertion of the updated tuple.

triggering operation	*transition table reference*	*database operation*		
		insert	delete	update
		flg, old, new	flg, old, new	flg, old, new
inserted	Λ	∀, ∅, ∅	C1, ∅, ∅	∅, ∅, ∅
inserted	inserted()	∀, ∅, ∀	C1, ∅, ∅	C1, ∅, C1
deleted	Λ	∀, ∅, ∅	∀, ∅, ∅	∅, ∅, ∅
deleted	deleted()	∀, ∅, ∅	∀, ∀, ∅	∀, ∀, ∅
updated	Λ	∀, ∅, ∅	∀, ∅, ∅	∀, ∅, ∅
updated	old_updated()	∀, ∅, ∅	∀, ∅, ∅	∀, ∀, ∅
updated	new_updated()	∀, ∅, ∅	∀, ∅, ∅	∀, ∅, ∀

Figure 5.5: Logging Requirements

The cumulative logging requirements for the operations of a table are found by computing the union of the all logging requirements for each (TO,TTR) that occurs in any rule defined on the table. For example, suppose the only rules defined on the `emp` table are the ones shown in Examples 5.1 and 5.2. The triggering operations for these two rules are `inserted` and `updated`, and the transition table references are `inserted()` and `new_updated()`. The logging requirements are found by computing the union of the logging requirements for (`inserted`, `inserted()`) and (`updated`, `new_updated()`). The following information must be logged: (1) all insertions and their values, (2) all deletions but not their values, (3) all updates, their new values, but not their old values.

The logging requirements for a table are compiled into an *information code* that is then associated with the table through the Starburst attachment mechanism in the rule attachment descriptor. The Transition Log is written during transaction execution by rule attachment procedures that reference this code. Figure 5.6 shows the pseudo-code for the portions of the attachment procedures `delete()`, `insert()`, and `update()` that decide what information should be logged when the corresponding database operation occurs.

```
delete()
{  if (not Delete-Trigger)
   { if (C1)
        record Delete Event;
     else
      if (Delete-Ref) record Old-Value;}
   record Delete Event;}

update()
{  if (Update-RefNu)
   {  record New Value;
      if (Update-RefOu OR Delete-Ref) record Old Value;
      record Update Event; }
   else
   {  if (Update-RefOu OR Delete-Ref)
      {  record Old Value;
         if (Insert-Ref and C1) record New Value;
         record Update Event; }
      else
      {  if (Insert-Ref and C1)
         {  record New Value
            record Update Event; }

         else if (Update-Trigger) record Update Event;}}}
```

```
insert()
{  record Insert Event;
   if (Insert-Ref)
      record New Value;}
```

```
KEY: C1:              previously inserted  tuple
     Insert-Ref:      rule on table that references inserted()
     Delete-Trigger:  rule on table triggered by deletions
     Delete-Ref:      rule on table that references deleted()
     Update-Trigger:  rule on table triggered by updates
     Update-RefNu:    rule on table that references new_updated()
     Update-RefOu:    rule on table that references old_updated()
```

Figure 5.6: Pseudo-Code for Rule Attachment Routines

5.6 Provisions for Rollback

Each component of a database system must be prepared for a rollback at any time. Rollbacks can occur when a system error occurs, when the transaction manager detects a deadlock, or they can be self-inflicted by the transaction. Furthermore, Starburst has provisions for partial rollback. That is, users (or programs) can request that the current transaction be rolled back to some previous point in time.

The rule system has been added as an integral part of Starburst. Hence, its components must provide for both complete and partial rollback. Since many of these components are implemented using existing Starburst components (through the extensibility mechanisms), many of the mechanisms required to handle rollbacks were already in place. However, the rule system must ensure that all its memory-resident data structures are modified to undo any changes made during the portion of the transaction being rolled back.

Two memory-resident data structures are maintained throughout the execution of a transaction: the Transition Log, discussed previously, and the Global Rule Information, which is a main-memory index for the current set of rules. A third data structure, Rule Processing Information, is maintained during the rule execution phase of each transaction; it contains information about potentially triggered rules and their triggering transitions. When a transaction is partially or completely rolled back, the rule system must ensure that its data structures are rolled back accordingly. It turns out that the Rule Processing Information never needs to be rolled back: a partial rollback cannot occur during rule execution except within the actions of a single rule, and a complete rollback causes termination of the rule execution phase. Hence, only the Transition Log and the Global Rule Information must be considered.

To handle rollback for the Global Rule Information, we use the *undo strategy*: whenever an action is performed, an opposite *undo-action* is added to a rollback queue and stamped with the current time. When a rollback occurs, all undo-actions on the rollback queue that have a time greater than the rollback point are performed. This strategy for handling rollback is quite natural, and is similar in philosophy to ARIES [MHL+91], the recovery mechanism of Starburst.

The Global Rule Information is a global structure that is initialized from the Rule Catalog on system start-up. It is modified only when the rule set is modified. Whenever a rule definition statement is rolled back, the rule system must ensure that the changes that were

made to the Global Rule Information are undone and the information code (recall Section 5.5) for the rule's table is recomputed. This is achieved by placing a parameterized procedure on a *commit* event queue each time a rule definition statement is executed. This event queue is processed by Starburst when a transaction commits or rolls back. When one of these procedures is invoked with the "rollback" flag, it modifies the Global Rule Information appropriately. For example, when a **create rule** statement is processed, a procedure is queued with a parameter that identifies the created rule. If a rollback occurs, this procedure removes the information for that rule from the Global Rule Information.

Sometimes this undo-action approach can be optimized, grouping several undo-actions together based on the semantics of the data structure. This is the case for the Transition Log. For complete rollback, this log is set to empty as part of the general rule system Transaction Clean-Up procedure for local data structures. This cleanup procedure is placed on the *commit* event queue by each transaction the first time any information is written to the Transition Log. For partial rollback, the rule system must remove all entries in this log corresponding to operations that occurred during the rolled back portion of the transaction. Starburst allows arbitrary procedures to be invoked whenever a partial rollback occurs; these procedures are called with a time-stamp parameter indicating the point to which the transaction is rolling back. This time-stamp corresponds to the `time-stamp` in Transition Log entries. Hence, a procedure for the rule system is invoked on partial rollback and removes from the Transition Log all entries with a time-stamp greater than the time-stamp received as a parameter.

5.7 Concurrency Control in Rule Execution

Since Starburst is a multi-user database system, we must consider the effect on the rule system of concurrently executing transactions. For most transactions, including those with triggered rules, concurrency control for data in the database is handled automatically by the database system, since rule conditions and actions are executed through the Starburst query processor. However, since several processes may be executing simultaneously, the memory-resident data used to trigger rules and maintain the current rule set may be simultaneously accessed by different processes. The semantics of the rule language does not allow modifications in one process to trigger rules in another process. Hence, each process maintains its own Transition Log.

The Global Rule Information is indexed by a hash table, hashed on the key `(op-type,`

`table-name`). It is shared by all processes, however all read and write operations to this structure must appear atomic. This is accomplished by using Starburst's *latching* (semaphore) mechanism for mutual exclusion—data items are latched in shared mode for the duration of a read operation and are latched in exclusive mode for the duration of a write operation. Latches permit greater concurrency since, unlike locks, they need not be held for the duration of the transaction. Furthermore, we obtain latches at the bucket level rather than at the hash table level, which also permits greater concurrency.

5.8 Summary

In this chapter we described the overall implementation of the Starburst Rule System. In addition to giving a very high-level description of the design, we described in detail several aspects of the implementation that are unique to this rule system. We described the structure and optimizations of the Transition Log required for computing the net effect of triggering operations and transition table references. We also described the extra steps necessary to support rollback and concurrency.

The Starburst Rule System was completed in 1991 and is currently being exercised with several applications. It consists of approximately 24,000 lines of C code including comments and blank lines (approximately 8,000 semicolons). The actual coding took only 9 woman-months, however the system was carefully designed before any of the implementation began. The next two chapters describe more general research that resulted from the design and implementation of the Starburst Rule System.

Chapter 6

Maintaining Priorities in Production Rule Systems

6.1 Introduction

In this section, we present a priority system which is particularly suited for rule systems that are coupled to databases. In this system, there are default priorities between all rules and overriding user-defined priorities between particular rules. Rule processing using this system is *repeatable*: for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions is executed twice on the same initial database state. The rule order *adheres* to the default order as closely as possible: rules are considered in the same order as the default order unless user-defined precedence constraints force an inversion.

We present data structures and efficient algorithms for implementing such a priority system. We show how the data structures can be incrementally maintained as user-defined priorities are altered. We also discuss how the proposed scheme can be extended to build a multi-level hierarchical priority system.

6.2 Requirements for Rule Ordering

A central issue in production rule systems is *conflict resolution* [MF78, IS89]. Given that two or more rules are triggered, a conflict resolution mechanism determines which rule is considered first for execution. Some rule systems (for example, Postgres [SHP88]) allow the users to specify absolute numeric priorities to conflicting rules which are used to resolve

[1]This chapter was published in VLDB 1991[ACL91].

conflicts at run time. Other systems (for example, OPS5 [For79]) use a combination of some static properties of the rules (such as the complexity of the antecedents) and some dynamic properties of the data (such as the age of the tuples satisfying the rules) to determine relative priority. In the case that no criterion resolves the conflict, a rule is chosen randomly, making the rule system *non-deterministic*.

Non-determinism in production rule systems has led to systems that have turned out to be much more complex and unwieldy than had been expected [Jac86], which in turn has inspired research into *deterministic* production rule systems [HH91, Ras90, ZH90]. Although not necessarily appropriate for all applications, deterministic production rule systems are more easily understood, maintained, and extended. They are particularly useful for rule bases coupled to databases, since the primary purpose of a rule base in such an environment is to automate deterministic activities [HH91]. We propose a new priority system for deterministic production rule systems in which there is a system defined *default priority* and users can specify *user-defined priorities*. Additionally the execution of rules exhibits *repeatability* and *adherence*, as subsequently defined.

Default Priorities

The rules in the production rule system have *default relative priorities* that are a function of the *static* properties of the rules. This function, p, defines a *default total order* over the production rules. A function yielding the creation timestamp of the rules (assuming creation timestamps are unique) is an example of such a function which gives higher priority to older rules. Production order rules, described in [MF78], provide other examples of such a function. We represent the default total order by $\stackrel{d}{\rightarrow}$ such that, given two rules R and S, if $p(R) < p(S)$ then $R \stackrel{d}{\rightarrow} S$. Default priorities may be specified by the user or induced by the system.

User-Defined Priorities

The user may explicitly specify relative priorities between particular rules by defining a *precedes* relationship between them. If the user has specified that rule R precedes S, and if both R and S have been triggered, then R is considered first for execution, regardless of the default total ordering. User-defined priorities are transitive; that is, if R precedes S and S precedes T, then R precedes T even if S is not triggered. Cycles are not permitted in the

user-defined priorities. $R \Rightarrow S$ represents that rule R has user-defined priority over S. We assume for convenience that for every rule R, $R \Rightarrow R$. If there are k rules T_k (k could be 0) such that $R \Rightarrow T_1 \Rightarrow T_2 \Rightarrow \ldots T_k \Rightarrow S$, we say $R \stackrel{*}{\Rightarrow} S$.

User-defined priorities override default priorities. The user may define priorities at the time of rule definition or separately. User-defined priorities are dynamic — they may be dropped and added at any point during the existence of the rule set.

Precedence relationships are a natural way of expressing user-defined priorities [WCL91a] because they increase rule autonomy [MF78]: they do not force the rule designer to know about all the rules in the system. Such relationships are also often the result of rule analysis [Ras90] and rule generation [CW90], which specify only the precedences that must be satisfied.

Repeatability

If the same set of transactions is executed twice with the same database state, the same set of rules, and the same user-defined and default priorities between the rules, then all rules are considered in the same order. This repeatability property is important since it is essential for a system to have predictable behavior. The repeatability property can be guaranteed if, given a default total order $\stackrel{d}{\rightarrow}$ over a set of rules $\mathcal{R}$ and an overriding partial order $\stackrel{*}{\Rightarrow}$ over a subset of rules in $\mathcal{R}$, we can obtain a new unique total order. The new total order is represented by $\stackrel{n}{\rightarrow}$.

The repeatability property is stricter than the *determinism* property considered in [HH91, Ras90, ZH90]. For example, [HH91] only requires that the production system have a unique fixed point, whereas the repeatability property insists that the computation path to the fixed point is also unique. However, [HH91] places constraints on rule sets to realize production systems with unique fixed points. The repeatability property guarantees determinism without constraining rule sets. Also, just having a unique fixed point can be inadequate for applications having side effects (an action external to the database, for example), since the behavior of the side effects may be monitored for auditing and for detecting anomalous behavior in the system. Hence, we require the stronger repeatability property.

Adherence to Default Order

The new total order $\xrightarrow{n}$ adheres to the default order to the extent permissible within user-defined precedence constraints. Starting with the first rule in the default order, the rules are put in the new order in the same order as the default order unless a user-defined priority forces a rule to come earlier. Consider, for example, the rule system consisting of rules R_0, R_1, R_2, and R_3, where the subscripts associated with the rules also denote their timestamps. Assume that the default order is to order the rules in increasing order of their timestamps, and the user-defined priorities are $R_3 \Rightarrow R_0$ and $R_2 \Rightarrow R_1$. Without the user-defined priority $R_3 \Rightarrow R_0$, the adherence property would require that R_0 come before any other rule in $\xrightarrow{n}$, as R_0 is the first rule in the default order. However, due to user-defined priority of R_3 over R_0, R_3 comes first and then R_0. Having placed R_0, the adherence property requires that R_1 be placed next in $\xrightarrow{n}$. However, the user-defined priority of R_2 over R_1 forces that R_2 be placed before R_1, and thus $R_3 \xrightarrow{n} R_0 \xrightarrow{n} R_2 \xrightarrow{n} R_1$ is the new total order.

Definition 6.1 (Adherence) $\xrightarrow{n}$ adheres to $\xrightarrow{d}$, if and only if, for all R and S, $R \xrightarrow{d} S$ and $S \xrightarrow{n} R$ if and only if:
i) $S \overset{*}{\Rightarrow} R$, or ii) $S \overset{*}{\not\Rightarrow} R$ and $\exists T$ such that $S \overset{*}{\Rightarrow} T$, $R \overset{*}{\not\Rightarrow} T$, and $T \xrightarrow{d} U$ for all U such that $R \overset{*}{\Rightarrow} U$ and $S \overset{*}{\not\Rightarrow} U$. Otherwise, $R \xrightarrow{d} S$ and $R \xrightarrow{n} S$. ∎

In other words, if R precedes S in the default total order then their ordering is reversed in the new total order if and only if the user has specified that S must precede R or that S must precede a rule T that has a higher default ordering than all the rules that R must precede. Any rule that has been specified by the user to follow both R and S is ignored in this decision.

The adherence property has a relationship to *inversions* [Knu73]. $\xrightarrow{n}$ is a permutation of $\xrightarrow{d}$ which satisfy the user-defined precedence constraints. In addition, the adherence property requires that, starting with the first item in $\xrightarrow{d}$, items have the same order in $\xrightarrow{n}$ as in $\xrightarrow{d}$ unless a user-defined precedence forces an inversion. This requirement resembles the priority-driven deadline scheduling of jobs in real-time systems [LL73, BMHD89]. However, in deadline scheduling, if the deadline for a task is missed, the task may not be scheduled at all. On the contrary, rules are never dropped in rule systems (although a higher priority rule may cancel the firing of a lower priority rule).

The priority system proposed in this chapter is the result of an effort to define a priority system for the Starburst Production Rule System. The design of this system allows the user

to define relative priorities between some rules and requires the rule system to be repeatable. However, the original algorithm for determining ordering between rules led to cycles in the rule priorities and, hence, did not produce a total order. Letting $ts(R)$ represent the creation time of rule R, the ordering between two rules R and S in [WCL91b] is determined as follows:

1. If $R \stackrel{*}{\Rightarrow} S$ and $R \neq S$, then $R \stackrel{n}{\rightarrow} S$.

2. If $S \stackrel{*}{\Rightarrow} R$ and $S \neq R$, then $S \stackrel{n}{\rightarrow} R$.

3. Otherwise, if $ts(R) < ts(S)$, then $R \stackrel{n}{\rightarrow} S$ else $S \stackrel{n}{\rightarrow} R$.

However, consider rules R_0, R_1, and R_2, such that $ts(R_0) = 0$, $ts(R_1) = 1$, $ts(R_2) = 2$, and $R_2 \Rightarrow R_0$. $R_0 \stackrel{n}{\rightarrow} R_1$ since $R_0 \stackrel{*}{\not\Rightarrow} R_1$, $R_1 \stackrel{*}{\not\Rightarrow} R_0$, and $ts(R_0) < ts(R_1)$. Similarly, $R_1 \stackrel{n}{\rightarrow} R_2$. Also $R_2 \stackrel{n}{\rightarrow} R_0$, since $R_2 \Rightarrow R_0$. Thus, $R_0 \stackrel{n}{\rightarrow} R_1 \stackrel{n}{\rightarrow} R_2 \stackrel{n}{\rightarrow} R_0$, a cycle. This chapter proposes a correct relative rule ordering algorithm.

The problem of task allocation with precedence relations [CL87, Law73, XP90] has similarities to the priority problem considered in chapter. Task allocation with precedence relations also considers the effect of precedence relations between modules on task scheduling. The precedence relations put constraints on the final order, and the total order satisfies the partial order imposed by these relations. However, conflicts are resolved using dynamic information about the jobs, which does not necessarily impose a total order. We, on the other hand, use the adherence property to arrive at the unique total order.

The organization of the remainder of this section is as follows. In Section 6.3, we present an algorithm for determining the order between two rules given a default total order over a set of rules and an overriding partial order over some rules in this set. We show that this algorithm leads to a new total order that *adheres* to the default order and guarantees *repeatability*. Section 6.4 discusses efficient implementation of this algorithm. Section 6.5 describes how changes in the user-defined priorities between rules can be handled incrementally.

Section 6.6 shows how our scheme can be extended to build a hierarchical priority system. Related rules are grouped into *rule classes*, as in [IBM88]. User-defined priorities are specified separately for rules in each class and also for rule classes themselves. This scheme extends naturally to multi-level hierarchies. We conclude with a summary.

6.3 Rule Ordering Algorithm

Definition 6.2 (Distinguished Rule) Given two rules R and S and an ordering function p that determines the default total order over the set of rules, the *distinguished rule* for R with respect to S and p, $d(R)_{S,p}$, is defined as follows:

1. If $S \stackrel{*}{\Rightarrow} R$, then $d(R)_{S,p} = R$.
2. If $S \stackrel{*}{\not\Rightarrow} R$, then $d(R)_{S,p}$ is defined to be the rule U such that all of the following hold:
   (a) $R \stackrel{*}{\Rightarrow} U$,
   (b) $S \stackrel{*}{\not\Rightarrow} U$, and
   (c) $\forall\, T$ such that $R \stackrel{*}{\Rightarrow} T$ and $S \stackrel{*}{\not\Rightarrow} T$, $p(T) \geq p(U)$.

■

For example, assuming that p is the function yielding the creation time of a rule and that $S \stackrel{*}{\not\Rightarrow} R$, the distinguished rule for rule R with respect to rule S is the oldest rule T that R must precede and that S does not precede in the user-defined priority ordering. Note that $d(R)_{S,p}$ always exists and is unique. Also, $d(R)_{S,p}$ could be R itself.

Algorithm 6.3 (Relative Rule Ordering) Given two rules R and S and an ordering function p that determines the default total order over the set of rules, applying the following two steps in order determines the relative ordering between R and S:

1. If $R \stackrel{*}{\Rightarrow} S$ and $R \neq S$, then $R \stackrel{n}{\rightarrow} S$. If $S \stackrel{*}{\Rightarrow} R$ and $S \neq R$, then $S \stackrel{n}{\rightarrow} R$.
2. Otherwise, let U be $d(R)_{S,p}$ and let V be $d(S)_{R,p}$. If $p(U) < p(V)$, then $R \stackrel{n}{\rightarrow} S$; otherwise, $S \stackrel{n}{\rightarrow} R$.

That is, the relative ordering between two rules is determined by the user-defined priority (direct or transitive) between them when there is one. Otherwise, the relative ordering is determined by the relative value of the default ordering function p for their respective distinguished rules.

For example, let the rule system consist of rules R_0, R_1, R_2, R_3, R_4, R_5, and R_6, where the subscripts associated with the rules also denote their creation time. Let the default total

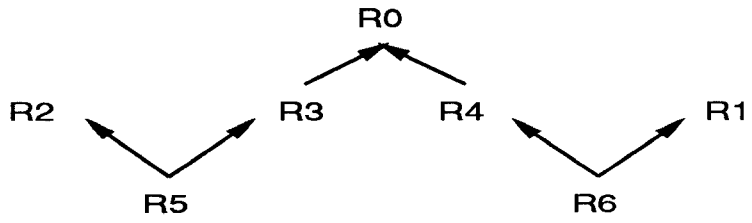


Figure 6.1: User-defined Priorities for Seven Rules

order be determined by the creation time of the rules, and let the user-defined priorities be as illustrated in Figure 6.1. Then, $R_6 \xrightarrow{n} R_5$ because $d(R_6)_{R_5,ts} = R_1$, $d(R_5)_{R_6,ts} = R_2$, and $ts(R_1) < ts(R_2)$.

Theorem 6.4 (Correctness of the relative rule ordering algorithm) Given a set of rules $\mathcal{R}$, the pairwise application of the relative rule ordering algorithm over rules in $\mathcal{R}$ generates a repeatable and adherent total order.

To prove that the relative rule ordering algorithm is correct, we must prove that the relation $\xrightarrow{n}$ defined by the relative rule ordering algorithm is a repeatable, adherent, total ordering of any given of set rules $\mathcal{R}$. This is done by proving several lemmas. Repeatability is satisfied by the fact that $\xrightarrow{n}$ satisfies a total order, and adherence has its own lemma.

Lemma 6.5 (Uniqueness) If R and S are two distinct rules in $\mathcal{R}$ and $R \xrightarrow{n} S$, then $S \overset{n}{\not\rightarrow} R$.

Proof: Suppose $R \xrightarrow{n} S$ and $S \xrightarrow{n} R$ for two distinct rules in $\mathcal{R}$. Now, $R \overset{*}{\Rightarrow} S$ and $S \overset{*}{\Rightarrow} R$ cannot both hold since there are no cycles in the user-defined priorities. If $R \overset{*}{\Rightarrow} S$, then $R \xrightarrow{n} S$, and $S \overset{n}{\not\rightarrow} R$ since the first step of the algorithm is always applied first. A similar argument holds if $S \overset{*}{\Rightarrow} R$. So $p(d(S)_{R,p}) < p(d(R)_{S,p})$ and $p(d(S)_{R,p}) > p(d(R)_{S,p})$ must both be true. This is not possible since $<$ is unique, a contradiction. □

Lemma 6.6 (Totality) Between every pair of distinct rules R and S in $\mathcal{R}$, either $R \xrightarrow{n} S$ or $S \xrightarrow{n} R$.

Proof: Consider two distinct rules R and S in $\mathcal{R}$. If there is a user-defined priority between these two rules, then obviously $\xrightarrow{n}$ holds between these two rules. If there is not

a user-defined priority between the rules, then $d(R)_{S,p}$ and $d(S)_{R,p}$ determine their relative ordering. Now, $p(d(R)_{S,p}) < p(d(S)_{R,p})$ or $p(d(R)_{S,p}) > p(d(S)_{R,p})$ since $d(R)_{S,p}$ and $d(S)_{R,p}$ are distinct and p is a total order. Therefore, either $R \xrightarrow{n} S$ or $S \xrightarrow{n} R$. □

Lemma 6.7 (Adherence) Let R and S be two distinct rules in $\mathcal{R}$. Then $p(R) < p(S)$ and $S \xrightarrow{n} R$ if and only if i) $S \overset{*}{\Rightarrow} R$, or ii) $\exists T$ such that $S \overset{*}{\Rightarrow} T$ and $p(T) < p(U)$, $\forall U$ such that $R \overset{*}{\Rightarrow} U$ and $S \overset{*}{\not\Rightarrow} U$. Otherwise, $p(R) < p(S)$ and $R \xrightarrow{n} S$.

Proof: (if) Suppose $p(R) < p(S)$ and $S \xrightarrow{n} R$. By definition of $S \xrightarrow{n} R$, either $S \overset{*}{\Rightarrow} R$ satisfying (i), or $p(d(S)_{R,p}) < p(d(R)_{S,p})$. Now, $S \overset{*}{\Rightarrow} d(S)_{R,p}$, $R \overset{*}{\Rightarrow} d(R)_{S,p}$, $S \overset{*}{\not\Rightarrow} d(R)_{S,p}$ and $\forall U$ such that $R \overset{*}{\Rightarrow} U$ and $S \overset{*}{\not\Rightarrow} U$, $p(d(R)_{S,p}) <= U$, so $p(d(S)_{R,p}) < U$, satisfying (ii).
(only if) Suppose $S \overset{*}{\Rightarrow} R$. Then, by definition, $S \xrightarrow{n} R$ even if $p(R) < p(S)$.
Suppose (ii) is satisfied. Then $p(d(S)_{R,p}) <= p(T)$, and $d(R)_{S,p}$ is the U with the minimal value of $p(U)$. So $p(d(S)_{R,p}) < p(d(R)_{S,p})$ and $S \xrightarrow{n} R$.
Therefore, $\xrightarrow{n}$ is adherent. □

Lemma 6.8 (Transitivity) If R, S, and T are distinct rules in $\mathcal{R}$ such that $R \xrightarrow{n} S$ and $S \xrightarrow{n} T$, then $R \xrightarrow{n} T$.

Proof: Suppose R, S, and T are distinct rules in $\mathcal{R}$ such that $R \xrightarrow{n} S$ and $S \xrightarrow{n} T$. Then exactly one of the following holds:

1. $R \overset{*}{\Rightarrow} S$ and $S \overset{*}{\Rightarrow} T$.
   Then $R \overset{*}{\Rightarrow} T$ since user-defined priorities are transitive and acyclic.
   Hence, $R \xrightarrow{n} T$.

2. $R \overset{*}{\Rightarrow} S$ and $p(d(S)_{T,p}) < p(d(T)_{S,p})$.
   Now, $T \overset{*}{\not\Rightarrow} R$, otherwise $T \xrightarrow{n} S$.
   Since $R \overset{*}{\Rightarrow} S$, $p(d(R)_{T,p}) <= p(d(S)_{T,p})$ and $p(d(T)_{S,p}) <= p(d(T)_{R,p})$. So, $p(d(R)_{T,p}) < p(d(T)_{R,p})$.
   Hence, $R \xrightarrow{n} T$.

3. $p(d(R)_{S,p}) < p(d(S)_{R,p})$ and $S \overset{*}{\Rightarrow} T$.
   $T \overset{*}{\not\Rightarrow} R$, otherwise $S \xrightarrow{n} R$.

Since $S \stackrel{*}{\Rightarrow} T$, $p(d(S)_{R,p}) <= p(d(T)_{R,p})$ and $p(d(R)_{T,p}) <= p(d(R)_{S,p})$. So, $p(d(R)_{T,p}) < p(d(T)_{R,p})$.
Hence $R \stackrel{*}{\Rightarrow} T$.

4. $p(d(R)_{S,p}) < p(d(S)_{R,p})$ and $p(d(S)_{T,p}) < p(d(T)_{S,p})$. In order to prove this case, we first show that $T \stackrel{*}{\not\Rightarrow} R$, and then prove by contradiction that $p(d(R)_{T,p}) < p(d(T)_{R,p})$. The following observation is useful:

   Observation 6.9 $\forall X$ such that $p(X) < p(d(P)_{Q,p})$ if $P \stackrel{*}{\Rightarrow} X$, then $Q \stackrel{*}{\Rightarrow} X$.

   Suppose $T \stackrel{*}{\Rightarrow} R$. Then $p(d(T)_{S,p}) <= p(d(R)_{S,p})$ and $p(d(S)_{R,p}) <= p(d(S)_{T,p})$, so $p(d(T)_{S,p}) < p(d(S)_{T,p})$, a contradiction. So, $T \stackrel{*}{\not\Rightarrow} R$.

   Suppose $p(d(R)_{T,p}) > p(d(T)_{R,p})$.
   Consider $p(d(S)_{R,p})$ and $p(d(S)_{T,p})$.

   (a) Suppose $p(d(S)_{R,p}) <= p(d(S)_{T,p})$. Then $p(d(R)_{S,p}) < p(d(S)_{T,p}) < p(d(T)_{S,p})$. Further consider $p(d(R)_{T,p})$ and $p(d(R)_{S,p})$.

      i. Suppose $p(d(R)_{T,p}) > p(d(R)_{S,p})$. Now, $R \stackrel{*}{\Rightarrow} d(R)_{S,p}$, so $T \stackrel{*}{\Rightarrow} d(R)_{S,p}$. Obviously, $S \stackrel{*}{\not\Rightarrow} d(R)_{S,p}$, so $p(d(T)_{S,p}) <= p(d(R)_{S,p})$. But $p(d(R)_{S,p}) < p(d(S)_{T,p})$, so $p(d(T)_{S,p}) < p(d(S)_{T,p})$ $(\Rightarrow \Leftarrow)$.

      ii. Suppose $p(d(R)_{T,p}) <= p(d(R)_{S,p})$. Then $p(d(T)_{R,p}) < p(d(S)_{R,p})$. But, by assumption $p(d(S)_{R,p}) <= p(d(S)_{T,p})$, so $p(d(T)_{R,p}) < p(d(T)_{S,p})$, and according to the observation, $S \stackrel{*}{\Rightarrow} d(T)_{R,p}$, so $p(d(S)_{R,p}) <= p(d(T)_{R,p})$, a contradiction.

   (b) Suppose $p(d(S)_{R,p}) > p(d(S)_{T,p})$. Recall that $p(d(R)_{S,p}) < p(d(S)_{R,p})$, so $R \stackrel{*}{\Rightarrow} d(S)_{T,p}$. Obviously, $T \stackrel{*}{\not\Rightarrow} d(S)_{T,p}$, so $p(d(R)_{T,p}) <= p(d(S)_{T,p})$. But the assumption $(p(d(T)_{R,p}) < p(d(R)_{T,p}))$ implies $p(d(T)_{R,p}) < p(d(S)_{T,p})$. Then $S \stackrel{*}{\Rightarrow} d(T)_{R,p}$, so $p(d(S)_{R,p}) <= p(d(T)_{R,p})$. But this implies $p(d(S)_R) < p(d(S)_T)$, a contradiction.

   Hence, $p(d(R)_{T,p}) < p(d(T)_{R,p})$, so $R \stackrel{n}{\rightarrow} T$.

So, in all cases, $R \stackrel{n}{\rightarrow} T$. □

Lemma 6.10 (Total Order) The relation $\stackrel{n}{\rightarrow}$ is a total order.

Proof: Since $\stackrel{n}{\rightarrow}$ is transitive, unique, and total, $\stackrel{n}{\rightarrow}$ defines a total order. □

6.4 Implementation

We now discuss how the relative rule ordering algorithm can be implemented efficiently. The total order $\xrightarrow{n}$ may be constructed by sorting all the rules in $\mathcal{R}$, using for comparison the `precedence` function. The following are the data structures used to compute rule ordering:

1. A rule ordering graph G for a given ordering function p and a given rule set $\mathcal{R}$. G is obtained as follows:
   (a) Corresponding to each rule R in $\mathcal{R}$, create a node R in G. Associate with node R the value of the default ordering function $p(R)$.
   (b) For each user defined priority $R \Rightarrow S$, create an arc from node R to node S in G.
   (c) Create an arc from every node R to itself.
2. The transitive closure G^* of graph G [ADJ90], where each node in G^* contains a sorted list of the transitive successors of the node in ascending order of p.

Given two rules `r1` and `r2`, the following function returns the rule that has the higher precedence.

```
function precedence(rule r1, rule r2)
                            returns rule
  {
    loop
(A)   {
        s1 = next(successor(r1));
        s2 = next(successor(r2));

(B)     // first use the user-defined
        // precedence, if any

        if (s1 == r2)
(C)         return  r1;
        else if (s2 == r1)
(D)         return  r2;

        // now use the default precedence

        else if (p(s1) < p(s2))
```

```
(E)         return r1;
          else if (p(s2) < p(s1))
(F)         return r2;
          else //  p(s2) == p(s1)
            continue;
        }
    }
```

During rule execution, the precedence between two rules is determined by the `precedence` function. The cost of every comparison is $O(n)$, where n is the number of rules in *ruleset*. However, for those pairs of rules that do not share common successors, the cost of computing the highest priority rule is constant. We expect this behavior for a majority of the rule pairs.

It may not be immediately obvious why the loop in the above `precedence` function always terminates before the shorter of the successor lists of `r1` and `r2` runs out. Also, if `r2` $\Rightarrow$ `r1`, then `r1` is not necessarily the first rule in the successor list of `r2` — there may be some other rule `s2` that comes before `r1` in the successor list of `r2`. The following theorem guarantees that, even in this case, the precedence function behaves correctly.

Theorem 6.11 (Correctness of the precedence function)
The function `precedence` generates the same relative ordering between two rules as the relative rule ordering algorithm.

We must prove that the loop terminates, and that at termination, `r1` is returned if and only if `r1`$\stackrel{n}{\rightarrow}$`r2`; `r2` is returned if and only if `r2`$\stackrel{n}{\rightarrow}$`r1`.

Lemma 6.12 (Loop Invariant) Assuming `s1` and `s2` are initially `NULL`, then `s1` = `s2` at `(A)` for each iteration.

Proof: This is clearly the case in the first iteration, since `s1` = `NULL` = `s2`. The loop terminates whenever `s1` $\neq$ `s2` since `p` is a total order. If `s1` $\neq$ `s2` then `p(s1)` $\neq$ `p(s2)`, and the loop exits at `(E)` with `p(s1)` $<$ `p(s2)`, or at `(F)` with `p(s1)` $>$ `p(s2)`. □

Lemma 6.13 (Termination) The loop does not execute indefinitely.

Proof: The loop will terminate since `r1` and `r2` are both in their own list of successors and there is a finite number of rules. Suppose `r1`$\stackrel{*}{\Rightarrow}$`r2` or `r2`$\stackrel{*}{\Rightarrow}$`r1`. Then the loop will terminate

at either (C) or (D) if not before. Suppose instead that $\texttt{r1} \overset{*}{\not\Rightarrow} \texttt{r2}$ and $\texttt{r2} \overset{*}{\not\Rightarrow} \texttt{r1}$. Then the loop will terminate when s1 = r1 or s2 = r2, if not before, since r1 is not in r2's successor list and r2 is not in r1's successor list. □

Observation 6.14 By the definition of `next(successor())`,

- p(s1) > all previously visited successors of r1,
- p(s1) < all unvisited successors of r1,
- p(s2) > all previously visited successors of r2, and
- p(s2) < all unvisited successors of r2.

Lemma 6.15 (Correctness of Function) Precedence returns r1 if and only if $\texttt{r1} \overset{n}{\rightarrow} \texttt{r2}$, and precedence returns r2 if and only if $\texttt{r2} \overset{n}{\rightarrow} \texttt{r1}$.

Proof:

1. Suppose precedence returns r1. Then the loop exited at either (C) or (E). If the loop exited at (C) then s1 = r2. So r2 is in r1's successor list. So $\texttt{r1} \overset{*}{\Rightarrow} \texttt{r2}$ and $\texttt{r1} \overset{n}{\rightarrow} \texttt{r2}$. If the loop exited at (E), then p(s1) < p(s2). Let s1a be the value of s1 and s2a be the value of s2 in the iteration preceding loop termination. Now p(s1a) = p(s2a) and p(s1a) < p(s1) < p(s2). So, by observation 6.14, s1 is a not in r2's successor list. Note, however, that s2 might be in r1's successor list. Suppose there is a user precedence between r1 and r2. Now, it cannot be the case that $\texttt{r2} \overset{*}{\Rightarrow} \texttt{r1}$, because then s1 would be in r2's successor list. So $\texttt{r1} \overset{*}{\Rightarrow} \texttt{r2}$ and $\texttt{r1} \overset{n}{\rightarrow} \texttt{r2}$. Suppose there is not a user precedence between r1 and r2. By observation 6.14 and the loop invariant, $\texttt{s1} = \texttt{d(r1)}_{\texttt{r2,p}}$ and $\texttt{s2} = \texttt{d(r2)}_{\texttt{r1,p}}$. Since p(s1) < p(s2), then $\texttt{r1} \overset{n}{\rightarrow} \texttt{r2}$.
2. Suppose precedence returns r2. Then the loop exited at (D) or (F). The proof that $\texttt{r2} \overset{n}{\rightarrow} \texttt{r1}$ follows in the same fashion as (1).
3. Suppose $\texttt{r1} \overset{n}{\rightarrow} \texttt{r2}$. The loop terminates at exactly 4 points.
   Suppose precedence does not return r1. Then precedence returns r2. But then, by (2), $\texttt{r2} \overset{n}{\rightarrow} \texttt{r1}$. But $\overset{n}{\rightarrow}$ is unique, a contradiction. So precedence must return r1.

4. Suppose r2$\xrightarrow{n}$r1. The proof that `precedence` returns `r2` follows in the same fashion as (3).

Therefore, the function `precedence` is correct. □

6.5 Addition and Deletion of Rules and Priorities

Rule systems are not static. Rules are continuously added and deleted, and user-defined priorities between existing rules are altered. Every time rules and/or priorities are added or deleted the rule ordering graph G could be reconstructed and its transitive closure G^* recomputed. However, instead of recomputing G^* from scratch, we can incrementally update G^*. The problem of incrementally updating compressed transitive closure has been considered in [ABJ89] and that of incrementally updating path information in [AJ89]. The techniques we present here apply to the general problem of incremental maintenance of complete transitive closure of acyclic directed graphs with sorted successors.

6.5.1 Incremental Additions

Addition of a new rule R simply results in the creation of a new node R in G^*. Also, there is an arc from R to R in G^*.

When the user wants to add a new priority for rule R over rule S, we need to first ensure that $S \stackrel{*}{\Rightarrow} R$ does not already exist; otherwise, the creation of $R \Rightarrow S$ will cause a cycle in the user-defined priorities. This requires a simple lookup in S's successor list. If $R \Rightarrow S$ is a legal addition, then the successor list of every predecessor of R (and R itself) needs to be updated, as they can now reach S and all the successors of S.

The following procedure incrementally updates G^* when a new user-defined priority $R \Rightarrow S$ is added:

```
// Addition of the user-defined priority,
// R => S

procedure addpriority(rule R, rule S)
 {
  // check for potential cycle in the
  // user-defined priorities
```

```
  if R is a successor of S in G*
   {
    disallow priority of R over S;
    return;
   }

  // legal user-defined priority ---
  // update data structures

  L = successors(S); // new reachable
                     // successors;
                     // S is included
                     // in successors(S)
  add(R,L);
 }


// Recursive procedure that adds to R and all
// its predecessors the rules in L

procedure add(rule R, list L)
 {
  // omit from L those rules that are
  // already in the successor list of R

  L = L - successors(R);

  // Add rules in L to the successor list
  // of R and its predecessors,
  // maintaining the correct order

  if L is not empty
   {
    // update the successor list of R
    successors(R) = successors(R) + L;
                     // maintain order

    // update the successor list of
    // predecessors of R
    for all P such that
      P is an immediate predecessor of R do
        add(P, L)  } }
```

The recursion terminates when all the predecessors of R have been updated. It is possible that the **add** procedure is not executed for some predecessor P if L becomes empty for all its successors.

Multiple visits to a predecessor of R can be avoided by some book-keeping. The first time a predecessor is visited, a bit is set for this rule indicating that this rule has already been visited. Now, before calling **add** for a predecessor P, this bit is tested to ensure that P has not been already visited.

6.5.2 Incremental Deletions

Deletion of a user-defined priority $R \Rightarrow S$ does not necessarily imply that S and all its successors should be deleted from the successor list of R and all its predecessors — there may be alternative paths.

The following procedure incrementally updates G^* when a user-defined priority $R \Rightarrow S$ is deleted:

```
// Deletion of the user-defined priority,
// R => S

procedure deletepriority(rule R, rule S)
 {
  L =  successors(S);  // rules potentially
                  // unreachable from R via S;
           // S is included in successors(S)

  delete(R, S, L);
 }


// Recursive procedure that deletes from R
// and all its predecessors the rules in L
// that are not any more reachable from them

procedure delete(rule R, rule S, list L)
 {
  // omit from L those rules for which
  // alternative path exists

  L = L -
    successors(immediate-successors(R) - S);
```

```
  // Delete rules in L from the successor
  // list of R and its predecessors

  if L is not empty
   {
    // update the successor list of R
    successors(R) = successors(R) - L;

    // update the successor list of
    // predecessors of R
    for all P such that
      P is an immediate predecessor of R do
        delete(P, R, L)
   }
}
```

As in the case of incremental addition, the recursion terminates when all the predecessors of R have been updated. It is possible that the `delete` procedure is not executed for some predecessor P of R if L becomes empty for at least one node on every path from P to R.

However, it is incorrect to apply the marking optimization discussed with `addpriority` to avoid multiple visits to a predecessor of R. The reason is that, to propagate the addition of a rule l in L to some predecessor P of R, it is sufficient to add l to one of the successors of P and then let P inherit l from this successor. However, to propagate the deletion of a rule l in L to some predecessor P of R, l must not be reachable from any successor of P. If l is only reachable from P through R, then l will only be deleted from P on the last visit to node P.

Deletion of a rule R results in the deletion of all incoming arcs into R and all outgoing arcs from R in G. The `deletepriority` procedure can be applied for each such arc, followed by the deletion of the node R in G.

6.6 Hierarchical Priority System

Rules are sometimes grouped into rule classes, as in [IBM88]. Rule classes are useful for structuring problem-solving by allowing related rules to be grouped into a separate class. User-defined priorities may be specified between rule classes and between rules within a class. The algorithm presented in Section 6.4 can be extended to handle such a hierarchical priority

system:

1. Create a *class ordering graph* $\mathcal{CG}$ as follows:
   (a) For every rule class C, create a node C in $\mathcal{CG}$. Associate with a node C a value which is the smallest of the value of the application of the default ordering function p on all the rules in C.
   (b) Create an arc C to D in $\mathcal{CG}$ if the rule class C has been specified to have a priority over the rule class D.
   (c) For every rule class C, create an arc from C to C in $\mathcal{CG}$.
2. Compute the transitive closure $\mathcal{CG}^*$ of $\mathcal{CG}$.
3. Create a rule ordering graph G and its transitive closure G^* separately for each rule class as in Section 6.4.
4. Now to determine the relative precedence between two rules, use $\mathcal{CG}^*$ if they belong to different rule classes, and use the corresponding G^* if they belong to the same rule class.

The preceding algorithm can be extended in a straightforward manner to handle multi-level class hierarchies. However, a limitation of this algorithm is that it does not directly admit user-defined precedence between rules in different classes.

6.7 Summary

We presented a priority system that is incrementally maintainable for combining user-defined priorities with default priorities. Such priority systems are becoming increasingly important in integrating production systems with database systems which require deterministic behavior. Precedence relationships are a natural way of expressing user-defined priorities [WCL91a] because they increase rule autonomy [MF78]: they do not force the rule designer to know about all the rules in the system. Such relationships are also often the result of rule analysis [Ras90] and rule generation [CW90], which specify only the precedences that must be satisfied.

Rule processing using this priority system is repeatable: for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions

is executed twice on the same initial database state. The rule order *adheres* to the default order as closely as possible: rules are considered in the same order as the default order unless user-defined precedence constraints force an inversion.

We also presented data structures and efficient algorithms for implementing such a priority system. User-defined priorities are dynamic — new priorities may be added and existing priorities may be deleted or altered. We showed how data structures required for priority determination can be incrementally maintained. Finally, we showed how the proposed scheme can be extended to build a multi-level hierarchical priority system. We are considering the implementation of this priority system in the Starburst extensible database system [HCL+90].

Chapter 7

Concurrency Control for Rule Definition

7.1 Introduction

The *Rule Definition*[1] *Module* is an essential component of every rule system. Even in the most basic research prototypes, users must at least be able to create and delete rules. Additional support for changing existing rules enhances the functionality and usability of the rule system and should be provided in any commercial product.

The integration of the Rule Definition Module into database systems has received relatively little attention in database rule system prototypes. Exceptions are the HiPAC project, which treats rules as first-class objects in an object-oriented database, and the Starburst Rule System, which provides concurrency control and a flexible language for changing rules. Both of these systems integrate rule definition with normal database processing. Rather than requiring that all rule definition commands be executed in single-user mode or off-line, these systems allow rule definition commands as standard operations in any user transaction. To support this feature, the rule system must ensure concurrency control between transactions that execute rule definition commands (referred to as rule definition transactions) and transactions that may trigger and execute rules.

Rule definition commands change the existing rule set. Rules are triggered by database operations, and the dependency between these operations and rules (especially non-existing rules) is not adequately represented in the relational and object-oriented representation of rules. Hence, concurrency control for rule definition is not automatically handled by the existing concurrency control mechanisms of the database system. To insure correct behavior of rule execution in the presence of concurrently executing transactions that change the rule set,

[1] We use the term *rule definition* to mean any changes to the rule set.

we define consistency requirements and provide solutions for enforcing these requirements. There are two kinds of consistency to consider:

1. *Rule Set Consistency* specifies that the rule set used by a transaction remains consistent throughout its execution, and that transactions are serializable with respect to rules (as well as data).

2. *Ordering Consistency* specifies that the order in which a transaction considers two rules does not change for the duration of the transaction.

We begin this chapter with a description of the rule definition language of the Starburst Rule System. We then discuss the above two consistency requirements separately. We formally define each requirement, propose a solution that meets requirement, and then prove that this solution is correct.

Throughout this chapter, we assume that the DBMS supports hierarchical two-phase locking (see Section 2.3.2), acquiring locks throughout a transaction as they are needed and holding them until the transaction commits or rolls back. Recall that any concurrent execution of a set of transactions must be equivalent to some serial execution of the set of transactions. With strict two-phase locking (i.e. locks are only released at commit time), the concurrent execution of a set of transactions is based on commit time.

7.2 The Starburst Rule Definition Language

In the Starburst Rule System, users create new rules with the **create rule** command described in Section 5.2. This system also supports commands that allow users to delete and change rules. All aspects of a rule, except its name, table, and transition predicate, can be modified using the command:

alter rule *rule-name* **on** *table-name*
[**if** *condition*]
[, **then** *action-list*]
[, **precedes** *rule-list*]
[, **follows** *rule-list*]
[, **nopriority** *rule-list*] ;

With the exception of **precedes**, **follows**, and **nopriority**, each specified clause replaces the existing attribute for that rule. For **precedes** and **follows**, the *rule-list* specifies rules

that should be added to or removed from the existing lists. The **nopriority** clause is used to remove priorities between the rule being altered and the rules listed in the **nopriority** rule list. The syntax for deleting rules is:

drop rule *rule-name* **on** *table-name* ;

Additionally, existing rules may be temporarily *deactivated* and *reactivated* using the following syntax:

alter rule *rule-name* **on** *table-name* **deactivate** ;
alter rule *rule-name* **on** *table-name* **activate** ;

The Starburst Rule System provides concurrency control for all rule definition commands. Hence, these commands can be executed as standard operations from any user transaction during normal database processing. The consistency requirements and solutions described in the remainder of this chapter are those developed for the Starburst Rule System.

7.3 Rule Set Consistency

There are two types of rule set consistency: *intra-transaction consistency*, which specifies that relevant rules remain consistent throughout a transaction, and *inter-transaction consistency*, which specifies that transactions are serializable with respect to rules (as well as data). These requirements are formally defined as follows.

Definition 7.1 (Requirements for Intra-Transaction Consistency) Let X be a transaction. The set of rules on any table modified by X cannot change after the first time X modifies the table. ∎

For example, if a database operation triggers a rule in some part of the transaction, then it should trigger the rule whenever the database operation occurs within the transaction.

Definition 7.2 (Requirements for Inter-Transaction Consistency) Let X_1 and X_2 be two transactions such that X_1 precedes X_2 in the serial schedule induced by the database system's concurrency control mechanism. If X_1 performs rule definition on a table modified by X_2, then X_2 must see the effect of X_1's rule definition. If X_2 performs rule definition on a table modified by X_1, then X_1 must not see the effect of X_2's rule definition. ∎

For example, when a new rule is defined, all transactions that contain a database operation corresponding to the rule's triggering event but do not trigger the rule must be serialized before all transactions that trigger the rule.

Intra-transaction consistency is clearly desirable. If X modifies a table and subsequently adds rules that are defined on that table, any further modifications that X performs on the table will trigger rules that were not triggered by the previous modifications. Similarly, if X modifies a table and subsequently deletes rules that are defined on that table, any further modifications that X makes to the table will not trigger rules that were triggered by similar previous modifications. Violations to intra-transaction consistency are illustrated by the following two examples.

Example 7.3 Assume that there are no rules triggered by operations on a table T. Now consider a transaction $X1$ that inserts a tuple $v1$ into T, subsequently defines a rule r that triggers on insertions to table T, and finally inserts another tuple $v2$ into T. This scenario is illustrated in Figure 7.1. Rule r will be triggered by the insertion of $v2$, but not by the

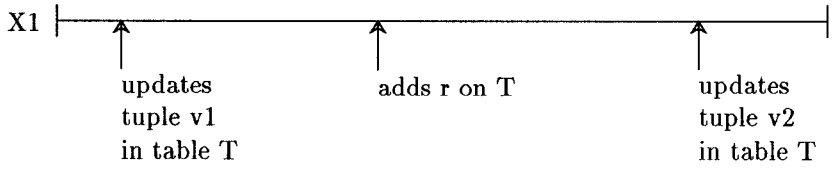


Figure 7.1: Intra-transaction Consistency Violation 1

insertion of $v1$. Hence, intra-transaction rule set consistency is violated. ∎

This type of violation can be prevented quite easily by allowing a transaction X to change the rules on a table T only if X has not previously modified T. However, T can be modified by X if the modification occurs after the rule definition command. The latter is considered consistent since all modifications to the table occur only after the change is made to the rules on the table.

Example 7.4 Assume that there are no rules triggered by operations on a table T. Now consider two transactions $X1$ and $X2$ such that transaction $X1$ inserts two tuples $v1$ and $v2$ into table T and transaction $X2$ creates a rule r that triggers on insertions to table T. Suppose the two transactions execute concurrently as shown in Figure 7.2.

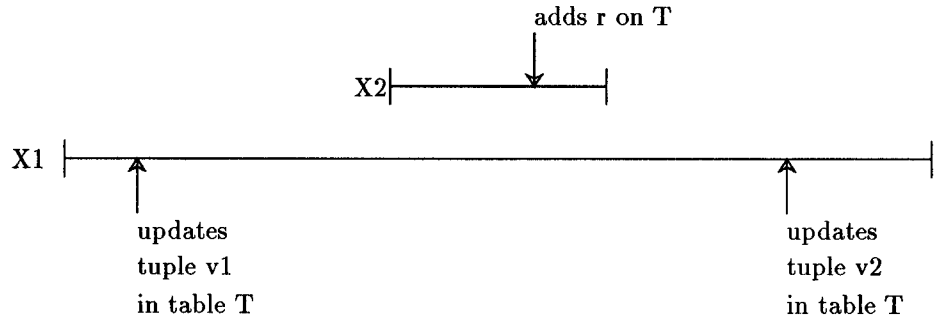


Figure 7.2: Intra-transaction Consistency Violation 2

$X1$ inserts tuple $v1$ into table T before $X2$ begins executing, so no rules are triggered by this modification to T. Subsequently, $X2$ creates rule r on table T that is triggered by insertions, then commits. Finally, $X1$ inserts tuple $v2$ into table T, which triggers rule r. This scenario violates intra-transaction consistency because $X1$ does not have a consistent view of the rule set. ∎

We present an algorithm below based on hierarchical locking that prevents such intra-transaction consistency violations, and enforces inter-transaction consistency as well. First, we discuss inter-transaction consistency in more detail.

Inter-transaction consistency specifies that transactions are serializable with respect to rules (as well as data). It imposes restrictions on how changes to the rule set made by a rule definition transaction RDT may affect concurrently executing transactions. All transactions that precede RDT in the resulting serial execution should not be affected by the changes made by RDT; all transactions that follow RDT in the resulting serial execution must be affected by these changes. Furthermore, if RDT is aborted, then no transactions should be effected by RDT's execution. As an example, suppose RDT deletes a rule r from the rule set. If r's triggering event is an update operation and the corresponding update occurs in any transaction that precedes RDT, then the transaction should trigger r. However, r should not be triggered by any transaction that follows RDT, even if the transaction performs a triggering update. A violation of inter-transaction consistency is illustrated by the following example.

Example 7.5 Assume there are no rules that are triggered by operations on a table T. Now consider two transactions $X1$ and $X2$ such that transaction $X1$ inserts a tuple $v1$ into table

T and transaction $X2$ creates a rule r that triggers on insertions to table T.

Suppose the two transactions execute concurrently as shown in Figure 7.3.

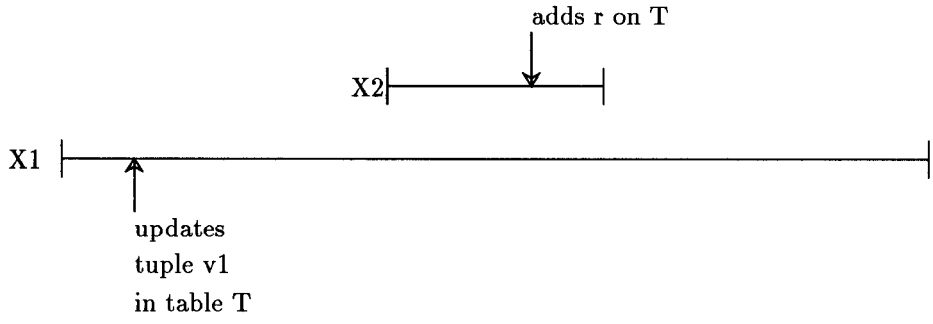


Figure 7.3: Inter-transaction Consistency Violation 1

$X1$ inserts a tuple $v1$ into table T before $X2$ begins executing, so no rules are triggered by this modification to T. Subsequently, $X2$ creates a rule r on table T that is triggered by insertions. $X2$ commits before $X1$, so $X1$ follows $X2$. But $X1$ does not trigger r and therefore consistency is violated. ∎

One seemingly obvious and easy solution to inter-transaction rule set consistency is to have every transaction take a snapshot of the rule set before it begins execution. However, this solution allows unserializable schedules as demonstrated with the following example.

Example 7.6 Assume there are no rules defined on table T. Consider three transactions $X1$, $X2$, and $X3$ such that (1) $X1$ updates two tuples $v1$ and $v2$ in T, (2) transaction $X2$ adds a rule r that is triggered by updates to T, and (3) transaction $X3$ also updates tuple $v2$. Suppose the three transactions execute concurrently as in Figure 7.4. $X3$ updates $v2$ and commits before $X1$ updates $v2$, so $X3$ precedes $X1$ in a serial schedule. However, $X3$ follows $X2$, so $X3$'s rule set contains rule r. Therefore, $X2$ precedes $X3$ in a serial schedule. Now, since $X1$ takes a snapshot of the rule set at the time it begins execution, it does not see rule r. Therefore, it precedes $X2$ in a serial schedule. Transitively, this implies that $X1$ precedes $X3$, conflicting with the schedule that is dictated by the updates to the data. ∎

This example demonstrates that a solution that satisfies inter-transaction consistency must consider dependencies between changed rules and database operations that trigger these rules. It is not sufficient to protect rules and data independently. Our solution, therefore, involves blocking the database operations that trigger changed rules.

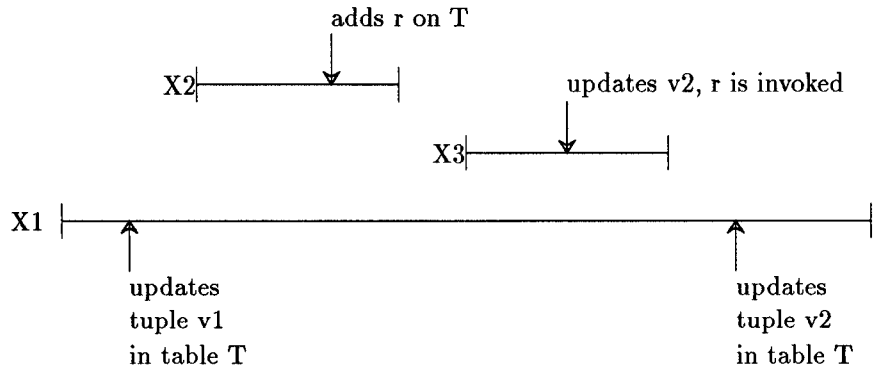


Figure 7.4: Inter-transaction Consistency Violation 2

Inter- and intra- transaction rule set consistency is enforced quite easily in a DBMS by (1) requiring that transactions do not perform rule definition on tables that they have previously modified and (2) using hierarchical locking, as follows. Table-level S-locks are obtained for all tables on which rules are defined, forcing the rule definition transaction to wait until all transactions currently modifying tables associated with the changed rules are finished, and disallowing any other future changes to these tables until the rule definition transaction commits. This is summarized in the f algorithm.

Algorithm 7.7 (Rule Set Consistency) Let X be a transaction that performs rule definition. Consider one rule definition command in X and let T be the table of the created, dropped, or altered rule. Before X can perform the rule definition command it must first do the following:

1. Check to see if it has modified T. If so, the rule definition statement is rejected, and an error is returned to the user.

2. Unconditionally obtain a table-level S-lock on T.

Theorem 7.8 (Correctness of the rule set consistency algorithm)
The rule set consistency algorithm enforces intra- and inter- rule set consistency for any set of transactions.

Proof: (Intra-transaction Consistency) Suppose that X is a transaction such that X does not have a consistent rule set. Then there must exist some table T and some rule R defined on T such that R is either created, dropped, or altered after the first time X modifies T.

Suppose that some other transaction $X2$ changes R. When X modifies T, it must obtain an IX-lock on T before it can X-lock the tuples that it modifies. But, $X2$ must obtain an S-lock on T before it changes R. This S-lock cannot be obtained between the time X first modifies T and the time X commits, since X holds an IX-lock on T. So changes to the global rule set by other transactions do not make X's rule set inconsistent.

Suppose instead that X changes R. Before X can change R, X must first check to see if it has previously modified T. If so, the rule definition does not proceed. Therefore, X can only change R before it changes T the first time. So changes to the global rule set by a transaction X do not make X's rule set inconsistent.

Therefore, Algorithm 7.7 enforces intra-transaction consistency.

(Inter-Transaction Consistency) Consider two transactions $X1$ and $X2$, where $X1$ precedes $X2$ ($X1 < X2$) in the serial schedule induced by concurrency control for the data. By definition, $X1$ commits before $X2$.

Suppose $X1$ performs rule definition. If $X2$ does not modify any tables for which rules were changed by $X1$, then consistency is maintained trivially. But, suppose $X2$ modifies a table T for which a rule was changed by $X1$. If $X2$ does not see the effects of the rule definition on table T, then $X2$ was modifying T before $X1$ modified the rule set for T. This implies that $X2$ obtained an X-lock on a tuple in T (and hence, an IX-lock on T) before $X1$ obtained an S-lock on table T. But $X1$ cannot obtain the S-lock until $X2$ commits, since $X2$ holds an IX-lock on T. Therefore $X2 < X1$;but, by assumption, $X1 < X2$, a contradiction.

Suppose $X2$ performs rule definition. If $X1$ does not modify any tables for which rules were changed by $X2$, then consistency is maintained trivially. But, suppose $X1$ modifies a table T and a rule on T was changed by $X2$. If $X1$ sees the changed rule set, then $X2$ must have changed the global rule set before $X1$ started modifying T. But $X2$ had to obtain a table-level S-lock to perform this modification, so $X1$ cannot change T until $X2$ releases the S-lock, and this does not happen until $X2$ commits. So $X2 < X1$. But, by assumption, $X1 < X2$, a contradiction.

Therefore, Algorithm 7.7 enforces inter-transaction consistency. □

7.4 Ordering Consistency

Ordering consistency specifies consistency requirements for rule priorities: if a transaction considers rule R before rule S at one point in its execution, then it always considers R before rule S. Formally, this requirement is stated as follows:

Definition 7.9 (Requirement for Ordering Consistency) Let X be any transaction and let R_1 and R_2 be any two rules on tables modified by X. The precedence relationship between R_1 and R_2 must not change during X from the first time the relationship is used in rule selection. ∎

Since rule set consistency prevents rule definition on tables that are currently being modified, it might appear that the algorithm for maintaining rule set consistency would also maintain ordering consistency. Rule set consistency does guarantee that the priority between two currently triggered rules cannot be explicitly changed. However, user-defined priorities are transitive (recall Section 6.1); that is, if R precedes S and S precedes T, then R precedes T even if S is not triggered. Transitive priorities may cause a reversal of the ordering between two rules in the rule set even if neither of the rules occur explicitly in the rule definition command. As an example, consider the rules R_1, R_2, and R_3 with the user-defined priorities $R_3 \rightarrow R_2$ and $R_2 \rightarrow R_1$. Then, by transitive rule priorities, $R_3 \rightarrow R_1$. If rule R_2 is dropped, the ordering between R_3 and R_1 is reversed. Therefore, ordering consistency is not guaranteed by rule set consistency.

Note that all rule definition commands—**create rule**, **drop rule**, and **alter rule**—may force recomputation of the priorities that determine rule ordering. When a rule definition command is executed, there is some minimal set N of nodes in the rule ordering graph whose successor lists are modified. This list is readily identifiable from the incremental algorithms used to update the rule ordering graph. There will be a node in N representing at least one of the rules from each pair of rules whose relationship is reversed by the resulting recomputation of priorities.

To enforce ordering consistency, we introduce locks on rules. Let an *RS-lock* be a shared rule lock and let an *RX-lock* be an exclusive rule lock. The compatibility of the locks is similar to standard S-locks and X-locks: RS-locks are compatible with other RS-locks but not other RX-locks; RX-locks are not compatible with either RX-locks or RS-locks. We add the restriction that RS-locks cannot be upgraded to RX-locks to prevent ordering relationships

from changing within a transaction that performs rule definition. Recall, also, that during its rule execution phase, each transaction maintains a sorted data structure, *Potential-Rules*, of potentially triggered rules. Ordering consistency is enforced by the following algorithm.

Algorithm 7.10 (Ordering Consistency) Let X be any transaction that triggers rules. Each time X adds a rule to Potential-Rules, it obtains an RS-lock on that rule. Let Y be a transaction that performs rule definition that modifies rule ordering and consider one such rule definition command in Y. Before performing the changes to the rule ordering graph caused by this command, RX-locks must be obtained for each of the rules in the graph whose successor lists are modified.

Theorem 7.11 (Correctness of the ordering consistency algorithm)
The ordering consistency algorithm satisfies ordering consistency for any set of transactions.

Proof: Consider a transaction X. Suppose, for the sake of a contradiction, that X considers the priority relationship between two rules, $r1$ and $r2$, twice. The first time $r1 < r2$, but the second time $r1 > r2$. A transaction only considers the priority between rules that are added to Potential-Rules, so the first time the priority relationship is computed, X holds RS-locks on $r1$ and $r2$. The priority can only change if one of the following occurs:

1. When X first computes the relationship between $r1$ and $r2$, it uses uncommitted changes made by $X1$ to the rule priority graph, and $X1$ subsequently aborts. But, if $X1$ changes the rule priority graph, it must obtain an RX-lock on either $r1$ or $r2$. So X could not obtain both RS-locks on $r1$ and $r2$. Therefore, X cannot use uncommitted changes to the priority graph.

2. Some transaction $X2$ (which is possibly X) changes the relationship between $r1$ and $r2$ between the times that X compares them. But, $X2$ must first obtain an RX-lock on at least one of the rules. This is impossible since X holds RS-locks on both of the rules until it commits.

Therefore, Algorithm 7.10 guarantees ordering consistency. □
This scheme is more restrictive than required by the specification of ordering consistency since an RX-lock on a rule r will block any other transaction T that considers r even if T does not consider the rules in r's successor list that were added or deleted by the change in the priority graph.

7.5 Comparison With Other Systems

With the exception of the HiPAC project, which treats rules as first-class objects, no database rule systems other than Starburst have considered full integration of the Rule Definition Module including concurrency control. In this section, we discuss HiPAC's solution and show one case in which rule set consistency is violated unless something similar to our hierarchical locking mechanism is used.

HiPAC treats rules as first-class database objects. As such, HiPAC rules are automatically provided with the creation, deletion, and modification operations that are common to all database objects. Furthermore, HiPAC rules have the following additional operations: fire, which runs the rule; disable, which disables the automatic firing of the rule; enable, which enables automatic firing of a disabled rule.

All database objects are subject to the following transaction semantics [Cha89]: a lock-type is associated with each operation and, before a transaction executes the operation on an object, it must obtain the associated lock for the object. In HiPAC, the rule firing operation requires an S-lock, and rule definition commands require X-locks. Locks are held for the duration of the transaction. Hence, any transaction that fires a rule must wait for any transaction that changes the rule.

Example 7.12 Consider a scenario similar to Example 7.5. Assume there are no rules that are triggered by operations on objects of type `T`. Now consider two transactions `X1` and `X2` such that transaction `X1` updates an instance `v1` of type `T` and transaction `X2` creates a rule `r` that triggers on updates to objects of type `T`. Suppose the two transactions execute concurrently as shown in Figure 7.5. `X1` updates object `v1` of type `T` before `X2` begins executing, so no rules are triggered by this modification to `T`. Subsequently, `X2` creates a rule `r` that is triggered by updates to objects of type `T`. `X2` commits before `X1`, so `X1` follows `X2`. But `X1` does not trigger `r` and consistency is violated. ∎

A solution in HiPAC's environment that is comparable to our hierarchical locking scheme is to use hierarchical locking of objects as follows.

1. Any rule definition operation must obtain S-locks on all object classes referenced in the triggering condition.

2. All modifications to objects must first obtain IX-locks on the object class before obtaining X-locks on the objects and performing the modification.

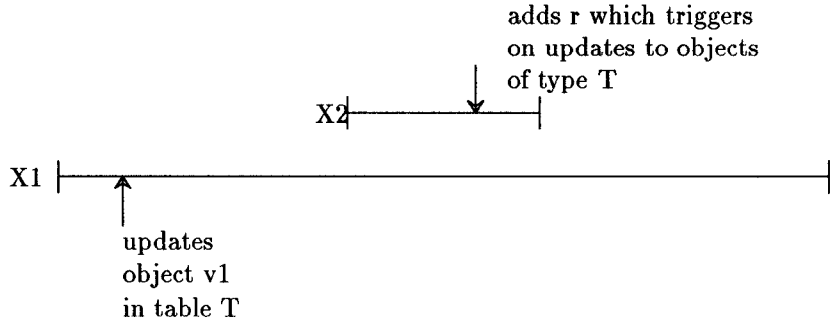


Figure 7.5: Intra-transaction Consistency Violation for Objects

Although not presented in the HiPAC papers [Cha89, CBB+89, MD89], HiPAC appears to include a concurrency control mechanism similar to this. However, our proposal was the first formal treatment of this issue in the literature.

Chapter 8

Conclusions and Future Research

The focus of this dissertation is the integration of active rule systems into multi-user database systems. This chapter concludes the major results of the dissertation and provides suggestions for future research.

8.1 Conclusions

An important issue in supporting active rules in a database system is to ensure that rule definition and processing behave correctly in the presence of concurrent users. Any features added to support a rule subsystem must coexist with existing database features such as concurrency control and recovery. We have addressed these issues by investigating two very different database rule systems: the Update Dependency Language, which uses a tentative goal-oriented search strategy, and the Starburst Rule System, which uses a forward-chaining irrevocable control strategy.

We described several different requirements for each of these systems. For the Update Dependency Language (Chapter 4), we defined requirements for safety of conditions and procedures, concurrency of rule execution, and mutual exclusion for the concurrent execution of a modification procedure. For the Starburst Rule System (Chapter 5), we also discussed concurrency of rule execution. In addition we considered the provisions that must be taken to support rollback. We also discussed two issues whose solutions were developed for the implementation of the Starburst Rule System but are generally applicable to dynamically changing rule sets in any system. In Chapter 6 we investigated the problem of maintaining rule priorities in database rule systems, and in Chapter 7 we described provisions that must be taken to support an environment in which users can modify their rule applications during normal database operation.

In this section, we summarize the issues considered for each rule system and describe the factors that influenced the need for handling each one. In particular, we focus on those issues that were described for one of the rule systems but not the other. We then suggest directions for future research.

8.1.1 Safety

Requirements for safe procedures and safe conditions, defined for Update Dependency procedures, guarantee that all variables in the head of a procedure are instantiated, as a side-effect, upon a successful execution of the procedure. The values of these variables may be subsequently used by the procedure, user, or application program requesting the modification. In the Starburst Rule System, the rules are indirectly triggered by operations, and the user or program that issues the triggering operation does not obtain return values from the execution of the rules. Within the rules, variables are only introduced through transition tables and in the SQL conditions and actions. These variables are inherently safe since all attributes of the transition tables are bound to the values of the tuples that trigger the rule and the SQL conditions and actions must be legal SQL. Requirements for safety should be supplied for any rule system, (e.g. Prolog and Datalog) in which a side-effect of rule execution is the instantiation of variables.

8.1.2 Rule Execution in a Multi-user Environment

The execution of rules in a multi-user environment (i.e. concurrency of rule execution) is considered for both rule systems. In general, the correctness of the execution of rules in separate user transactions is handled by the concurrency control mechanisms of the DBMS since the rule actions are operations on the database. However, depending on the language and the implementation, other issues related to concurrent rule execution must be considered.

If global main memory structures are used as auxiliary mechanisms for detecting rule triggering and performing rule execution, then access to these structures must be protected by the use of latches (i.e. semaphores). In Starburst, a global hash table containing information about rules is maintained in shared memory. Therefore, whenever a process reads from this table, it must obtain a latch on the data structure for the duration of the read. Also, if a process creates or deletes a rule it must obtain write latches on the table during these operations. For the Update Dependency Language, it has not been decided what main

memory structures will be used to facilitate rule processing.

In some systems, auxiliary structures for detecting triggering and processing rules are implemented using database tables (e. g. DIPs [SLR88]) rather than main memory. Such systems accommodate very large rule sets, and are protected by the locking mechanism of the DBMS. However, in contrast to main memory structures that are protected by latches which are only held for the duration of the operation, the locks are held for the duration of the transaction and may reduce concurrency.

When tentative control strategies are used for rule execution, two-phase locking may be overly restrictive. For this reason, we have considered relaxations to this strategy for the execution of the Update Dependency Language. We have shown that it is semantically incorrect to release S-locks on failed paths before a successful solution for a user-issued modification is chosen. Hence, the existing semantics for nested transactions and partial rollback do not support the notion of failure (as opposed to abort). However, two-phase locking can be relaxed to allow early release of X-locks along failed paths and early release of S-locks along failed and unchosen paths once a successful execution of the user-issued modification is chosen. Such optimizations are not considered for irrevocable control strategies, such as the one used to execute Starburst rules, since the modifications issued by the execution are never retracted. Hence, although HiPAC also uses the concept of nested transactions for rule execution, they need not consider the problem of early release of S-locks since the rule execution is irrevocable; the applicability of a rule triggered by an event is independent of the success of the other rules triggered by the same event.

8.1.3 Concurrent Processing Within Rule Execution

Concurrent processing within rule execution refers to the use of concurrency for executing rules within the same user transaction. The execution of a modification procedure in Update Dependencies involves finding and applying one of the procedure's successful rules. Since exactly one of the rules can be applied, the execution can concurrently apply all of the rules. However, care must be taken to insure that the applications are independent and, if there exist successful rules, the effects of the application of exactly one of these rules persists. Therefore, the concurrent processing must ensure mutual exclusion for the application of the different rules. We have described two techniques for mutual exclusion: one using nested transactions and one using shadow paging.

In Starburst, the events that trigger rules are based on the net effect of all database operations that have occurred since the rule was last executed, or since the beginning of the transaction if the rule has not been executed during the current transaction. There are several aspects of this semantics that are inherently sequential. For example, the actions of a given rule may invalidate the triggering event of simultaneously triggered rules, alter the values of transition tables in triggered rules, or change the database state seen by triggered rules. Hence, the Starburst semantics does not naturally lend itself to concurrent processing,

In HiPAC, all rules triggered by an event are executed, even if some subsequent action invalidates the event. HiPAC rules are processed concurrently through the use of nested transactions. Since all of the rules are applied, the only requirement for their concurrent execution is that the subtransactions are serializable. This is guaranteed by the mechanism for executing nested transactions.

8.1.4 Recovery of Rule Execution

All components of a database system must be prepared for a rollback at any time. For rule systems in which rule processing for a given event occurs in the same transaction (Starburst) or under a top-level user transaction (the Update Dependency Language), recovery for full rollback is handled completely by the recovery mechanism of the underlying database system. However, any modifications to the rule set must be undone as will be discuss in Section 8.1.6.

In HiPAC, rule processing can be decoupled from the transaction in which the triggering event occurred, and the transaction T_E in which an event occurred may commit before the transaction T_R that executes the triggered rule. If a system crash occurs after T_E commits and before T_R commits, T_R is aborted but T_E persists. Hence, HiPAC automatically restarts all such aborted transactions for rules triggered by the events of committed transactions. Starburst and Update Dependencies need not consider such a recovery mechanism since rule processing is not decoupled from the triggering transaction.

Rule systems that have deferred bindings, such as the Starburst Rule System, must make provisions for partial rollback. All memory-resident data structures, even local ones, that are updated during a time in which a partial rollback can be issued must be recoverable. When the transaction is partially rolled back, these data structures must be rolled back accordingly. Recovery is automatically handled by the DBMS for any auxiliary structures maintained by relations. Provisions for partial rollback were not considered by any other

system, probably because not all DBMSs support partial rollback. This was an issue with the Starburst Rule System since Starburst supports this feature.

Rule systems with immediate bindings, such as Update Dependencies, PRSI, and PRSII do not need to provide for recovery during partial rollback since the triggering and execution of the rule is within a single user statement. Notice that this is different from utilizing partial rollback as an implementation technique for backtracking in tentative control strategies.

8.1.5 Priorities

In Chapter 6 we investigated the problem of maintaining rule priorities in database rule systems that is directly applicable to the Starburst Rule System. We developed a priority system that combines a user-defined partial rule order and a default system-generated total rule order in such a way that the resulting rule execution is repeatable and adherent. This priority system can also be applied to any of the rule systems in which a default behavior can be identified and the user specifications define a partial order among the rules. Such systems include PRSI and Ariel which allow users to specify absolute priority values to rules and PRSII which allows users to define an exception hierarchy.

The Update Dependency Language does not have a facility for allowing users to specify priorities between rules. If there are several possible solution rules for a given activation of a procedure, then the choice of the rule is nondeterministic. If the designer wants control over which rule is chosen, then the conditions of the candidate rules should be mutually exclusive. Forcing the users to control rule ordering through a more explicit specification of rule conditions is not desirable with general production rule facilities, such as Starburst, since one of the benefits of production rules is the ability to incrementally build applications. With Update Dependencies, only the rules of a given procedure are triggered simultaneously, and we expect this number to be relatively small. Furthermore, when changes are made to a modification procedure (to enhance, improve, or correct the procedure), all of the rules must be reviewed since the semantics state that exactly one of the rules is applied for a given procedure activation. However, it might be useful to provide heuristics for searching the rules in the form of a partial order on the rules. If this is done and deterministic behavior is desired, our priority system could be used to determine the order in which the rules are tried.

8.1.6 Concurrency and Recovery of Rule Definition

In Chapter 7 we described provisions that must be taken to support an environment in which users can modify their rule applications during normal database operation. With the exception of the HiPAC project, which treats rules as first-class objects, no database rule systems other than Starburst have considered full integration of the Rule Definition Module including concurrency control. Although our solution has been presented in terms of the Starburst Rule System, it is generally applicable to any of the rule systems that are defined for relational systems and have rules that trigger on modifications. Further techniques must be investigated for rules triggered by other event types.

As discussed in Chapter 5.7, the main memory structure used to index the rules (Global Rule Information) must be protected during rule definition commands using write latches. This provision must be taken for any other rule system that allows rule definition during normal database operation and maintains a global main memory representation for the rules. Furthermore, if the transaction performing the rule definition is rolled back, then the operations that undo the effects of the rule definition (in the rule catalogs) must be issued.

8.2 Directions for Future Research

8.2.1 Rule System Application and Evaluation

We presented several techniques for integrating the Update Dependency Language and the Starburst Rule System into multi-user database systems, and implementations for both of these systems exist (to different degrees). However, to appropriately evaluate the techniques used for these and other database rule systems, substantial applications must be developed using these languages. These applications can then be analyzed to determine typical usage of the language and to evaluate how our mechanisms behave under normal usage.

In the case of the Update Dependency Language, we presented two different strategies, one concurrent and one sequential, for executing modification procedures. Once the typical usage of this language is better understood, these strategies should be evaluated. For example, if the conditions of the rules in a procedure are typically mutually exclusive, then the concurrent execution strategy quickly reduces to depth-first execution strategy. Also, if operations on competing execution paths in the execution pyramid of a procedure typically write and read the same data pages, then the optimization for early-release of X-locks may

cause more overhead with little gain in concurrency.

Several applications have been built using the Starburst Rule System. For a number of typical database applications such as constraint maintenance, incremental view maintenance, and deduction, rules can be semi-automatically generated from declarative specifications of the application [CW90, CW91]. However, a large-scale application by external users, such as the CIM system described in Chapter 4.8, would demonstrate typical and extensive usage of Starburst rules. Furthermore, it would test the performance of the transition log and the benefits gained from its optimizations.

8.2.2 Rule Languages for Multidatabases

The Update Dependency Language and the Starburst Rule System support consistency maintenance within one user transaction. With the emergence of multidatabases [LMR90], these rule languages must be adapted to facilitate consistency maintenance in autonomous, interoperable databases.

New language constructs must be introduced to support multidatabase systems' modified notions of consistency, concurrency control and transactions. We suggest the introduction of two such constructs.

- A decoupling mode similar to the one proposed by the HiPAC project. This mode would (perhaps implicitly) be used to decouple requests from one site for modifications on an autonomous site. The assumption here is that the validity of a modification on a site should not be dependent on modifications at other autonomous sites. However, modifications at a site can be requested from remote sites.

- Timing constraints for remote modification requests. This construct would allow a time limit for the success or failure of remote modification requests. This allows the requesting site to take alternative measures if a remote request fails. This is particularly useful if the remote site cannot be reached due to a network failure.

8.2.3 Efficiency Issues for the Update Dependency Language

We have described execution strategies for modification procedures that involve the complete computation and processing of rule conditions. However, the semantics of the language require that only one of the tuples that results in a successful execution of the actions is

found. In the worst case, each tuple that satisfies the condition must be tried since there is no way to determine a priori which tuples (if any) result in successful execution of the actions. Using depth-first execution, tuples are only tried until a successful one is found, and all effort spent computing the unexecuted tuples is wasted. Therefore, a strategy that generates tuples one unit (i.e. a page) at a time should be considered. Furthermore, with the potential for backtracking, the results of condition evaluation should be cached in such a way that the backtracking need not reevaluate conditions or perform differences on already checked tuples. However, these caches must behave as snapshots, since any future modifications must not be reflected in them when backtracking occurs. The modification procedures that we have seen are typically recursive and often the same condition is reevaluated when only minor changes have been made to the relation. This property suggests the use of incremental computation methods [NR91].

8.2.4 Efficiency Issues for the Starburst Rule System

The current implementation of the Starburst Rule System manages transition information for rules that trigger on one table very efficiently. This information is kept in main-memory and used to limit the number of rules considered during rule processing. These rules also contain a condition clause which may reference data in the database as well as the transition information. When a rule's transition predicate is satisfied, this condition must also be computed to determine if the rule's actions should be executed. Other research [SLR88, SLR90] has proposed techniques for optimizing the detection of rule triggering by maintaining auxiliary information for conditions during database modification. These techniques also avoid the recomputation of similar conditions for each triggered rule. Some of these optimizations may be applicable to the conditions in the Starburst rules. The condition information can be used as a filter that will reduce the amount of transition information kept and, hence, reduce the number of rules considered that have a condition which is not satisfied. This work involves supporting simple selection filters, complex join filters, and filters that are the union of the conditions of all rules that have similar transition predicates. Complex join filters involve maintaining projections of values from a relation, which may be large. Auxiliary structures will need to be developed for maintaining this information.

During rule execution, rules are considered based on transition information. Their actions are executed if their transition predicates and conditions are satisfied by the database and

the transition information. Rule actions typically modify the database and may affect the evaluation of transition predicates and conditions of other rules. Furthermore, rules often have conditions (which are queries to the database) that appear again in the action of the rule. Both static and run-time analysis may be useful for composing and grouping rules in a manner that minimizes the cost of detecting rule triggering and optimizes condition evaluation and action execution.

8.2.5 Structuring Mechanisms For Rule Programs

Rule classes [IBM88, SJGP90] are useful for structuring problem-solving by allowing related rules to be grouped into a separate class. However, the process of building a rule program is iterative, and the classification of rules evolves with time. Structuring mechanisms should be developed for rule programs. This includes extending the rule language and providing tools for the rule programmer.

In Chapter 6, we extended the priority algorithm for rule classes. However, we still need to specify how the priority system should evolve with the evolving rule classes. In our system, we only allow priorities between rule classes and between rules within the same class. If a new subclass is formed within an existing class, it must be determined how existing priorities between rules are reflected in the new class structure, especially when the new subclass separates rules for which a user-defined priority has been defined. Modularity is an important structuring mechanism in other programming paradigms, and should be useful as a tool for database rule programming as well.

8.2.6 Priorities

In Chapter 6, we described an adherence property for a rule priority system in which the resulting order between two rules is always the default order of the two rules' highest priority non-common successors. We have recently discovered that there are alternative adherence properties that one might like to minimize. We are currently investigating the general problem of minimizing the total number of inversions between the new total order $\xrightarrow{n}$ and the default total order $\xrightarrow{d}$ while also minimizing a given metric. With different metrics we can then define a class of adherence properties that are applied based on the needs of the application.

Appendix A

Syntax of Modification Procedures

The syntax of the definition of modification procedures for the Update Dependencies Language is given by the following syntax, assuming that the symbols *expressions*, *variables*, *attr-name*, and *rel-name* are terminals whose structure is further defined by a lexical analyzer.

proc-def ::= **define proc** *proc*
proc ::= *insert-proc* | *delete-proc* | *update-proc*
insert-proc ::= **insert** *rel-name*(*attr-parm-list*) *rule-set*
delete-proc ::= **delete** *rel-name*(*attr-parm-list*) *rule-set*
update-proc ::= **update** *rel-name*(*attr-parm-list*; *attr-parm-list*) *rule-set*
rule-set ::= *rule rule-set*
rule ::= ->*cond*, *action-list*.
action-list ::= *action*, *action-list*
| ϵ
action ::= *doit-act* | *req-act* | i/o-act
req-act ::= **insert** *rel-name*(*attr-expr-list*)
| **delete** *rel-name*(*attr-expr-list*)
| **update** *rel-name*(*attr-expr-list* ; *attr-expr-list*)
doit-act ::= **ins** *rel-name*(*attr-expr-list*)
| **del** *rel-name*(*attr-expr-list*)
| **upd** *rel-name*(*attr-expr-list* ; *attr-expr-list*)
i/o-act ::= **write** (*variable*)
| **read** (*variable*)
cond ::= *rel-name* (*attr-parm-list*)
::= *symbol relop symbol*
::= ϵ
::= **nonvar** (*variable*)
::= **var** (*variable*)
::= **not** *cond*
::= *cond* **and** *cond*

	::=	*cond* **or** *cond*
	::=	**exists** *var-list cond*
	::=	(*cond*)
symbol	::=	*variable* | *expression*
var-list	::=	*variable var-list*
	|	*variable*
relop	::=	< | <= | > | >= | = | <>
attr-expr-list	::=	ϵ
	|	*attr-name* = *expression*, *attr-expr-list*
	|	*attr-name* = *expression*
attr-parm-list	::=	ϵ
	|	*attr-name* = *variable*, *attr-parm-list*
	|	*attr-name* = *variable*

Appendix B

Examples of Modification Procedures

Example B.1 View Materialization

Assume that the view `v(x,y,z)` is defined as the join of `r(x,y)` with `s(y,z)`. If the system maintains a materialized version of `v` (i.e. the database administrator defines `v` as a base relation, but `v` can only be modified through modifications to `r` and `s`), then modification procedures can be used to specify the incremental maintenance of `v`. The following insert procedures perform incremental maintenance for insertions to `v` that result from insertions to `r` and `s`. We show only the tuple insertion rule for `r`; the rule for `s` is similar.

```
insert v(x=v1, y=v2, z=v3)
-> nonvar(v1) and nonvar(v2) and nonvar(v3) and
   not v(x=v1, y=v2, z=v3),
   ins v(x=v1, y=v2, z=v3).

insert r(x=v1, y=v2)
-> r(x=v1, y=v2) and
   not exists v3 (s(y=v2, z=v3) and not v(x=v1, y=v2, z=v3)).
-> r(x=v1, y=v2) and
   s(y=v2, z=v3) and not v(x=v1, y=v2, z=v3),
   insert v(x=v1, y=v2, z=v3),
   insert r(x=v1, y=v2).
-> not r(x=v1, y=v2),
   ins r(x=v1, y=v2),
   insert r(x=v1, y=v2).
```

The insertion procedure for `v` simply enforces a set-semantics (i.e. no duplicates). The insertion procedure for `r` recursively triggers insertions into `v` until `v` is consistent with `r`. Note that the conditions for the rules of this procedure are mutually exclusive. If there were a fixed order for selecting qualified rules, the designer may have been tempted to use

this order to imply semantics about the conditions. For example, suppose rules are tried in the order they appear in the procedure. Then, she may have been tempted to exclude the existential clause from the second rule's condition and to not include any condition for the third rule.

Note that it is possible to generate correct modification procedures for incrementally maintaining views. Incremental maintenance is a fundamental design goal of ADMS [Rou91], which performs incremental maintenance for all SPJ views defined. Recently, [CW91] specified how Starburst production rules can be generated to support incremental view maintenance. ∎

Example B.2 View Modification - Application Independent

Suppose that an application requires the ability to specify modifications for the view `v` defined in the previous example. Then `v` is defined as a base relation with the additional integrity constraint, `v(x,y,z) =r(x,y) join s(y,z)`. Modification procedures to maintain this integrity constraint must be specified. The following is one such insertion procedure for `v`, where the tuple insertion rules on r and s are specified above.

```
insert v(x=v1, y=v2, z=v3)
-> not v(x=v1, y=v2, z=v3),
   ins v(x=v1, y=v2, z=v3),
   insert r(x=v1, y=v2),
   insert s(y=v2, z=v3).
```

The tuple procedure for `v` makes the insertion into `v` and requests insertions into `r` and `s`; these, in turn, recursively request any additional needed insertions into `v` as above. This is another example where the modification procedure can be generated from the view definition. The view contains all attributes from its deriving relations and hence this modification is well-defined, independent of the application semantics of `v`. ∎

Example B.3 Data Evolution

Suppose an IRS database records information about taxpayers and dependents. Information about independent taxpayers is recorded in the relation
`taxpayer(payer-ss#, payer-name, addr, #dep)` with key `payer-ss#` and the dependents of all independent taxpayers is recorded in the relation
`dependent(payer-ss#, dep-ss#, dep-name)` with key `dep-ss#`. The following rules govern the evolution of data in this database:

- a person cannot simultaneously be an independent taxpayer and a dependent of an independent taxpayer.
- no person can claim themselves as a dependent
- A person is never deleted from the database. However, dependents can evolve into independent taxpayers.

The following set of modification procedures maintains these rules.

```
insert taxpayer(payer-ss#=s, payer-name=n, addr=a)
-> nonvar(s) and nonvar(n) and nonvar(a) and
   not taxpayer(payer-ss#=s) and not dependent(dep-ss#=s),
   ins taxpayer(payer-ss#=s, payer-name=n, addr=a, #dep=0).
-> nonvar(s) and nonvar(n) and nonvar(a) and
   not taxpayer(payer-ss#=s) and dependent(payer-ss#=p, dep-ss#=s),
   ins taxpayer(payer-ss#=s, payer-name=n, addr=a, #dep=0),
   delete dependent(dep-ss#=s),
   update taxpayer(payer-ss#=p, #dep=nd; #dep=nd - 1).

insert dependent(payer-ss#=p, dep-ss#=d, dep-name=n)
-> nonvar(p) and nonvar(d) and nonvar(n) and
   not taxpayer(payer-ss#=d),
   update taxpayer(payer-ss#=p, #dep=nd; #dep=nd+1),
   ins dependent(payer-ss#=p, dep-ss#=d, dep-name=n).

delete dependent(dep-ss#=d)
-> nonvar(d) and dep(payer-ss#=p, dep-ss=d, dep-name=n)
   and taxpayer(payer-ss#=p, addr=a) and not taxpayer(payer-ss#=d),
   del dependent(dep-ss#=d),
   insert taxpayer(payer-ss#=d, payer-name=n, addr=a).
-> nonvar(d) and taxpayer(payer-ss#=d),
   del dependent(dep-ss#=d).
-> nonvar(d) and not dep(dep-ss#=d) and
   taxpayer(payer-ss#=d).
```

The procedure for updating taxpayers is left as an exercise to the reader. It should ensure that the taxpayer being modified is an independent taxpayer.

■

Example B.4 Non-Full Functional Dependencies

Normalization [Cod72, Cod74, Arm74, Ber76, Fag77, Fag79] is a design technique that was

introduced to eliminate inherent *functional dependencies* that result from the existence of redundancy. Although we do not advocate the use of our language as a replacement for normalization, it provides an alternative to enforcing functional dependencies when it is undesirable to normalize. If a functional dependency exists from an attribute X of relation R to an attribute Y of R (written $X \rightarrow Y$) then each X-value in R must have associated with it exactly one Y-value. X is the determinant of the functional dependency. A relation that is in Boyce/Codd normal form (BCNF) is one in which all determinants are candidate keys of the relation. That is, given a value V for any determinant of R, V uniquely identifies a tuple in R. The advantages to normalized relations is that the functional dependencies can be enforced by key constraints.

However, it is often undesirable to normalize a relation, in which case functional dependencies that have a determinant which is not a candidate key of the relation cannot be enforced by key constraints. For example, the relation `shipments(s#, sname, saddr, p#, qty)` has the functional dependencies `s#` $\rightarrow$ `(sname, saddr)`, and `(s#,p#)` $\rightarrow$ `qty`. Both `s#` and `(s#, p#)` are required as keys to enforce these functional dependencies but cannot coexist as such. The following set of procedures ensures that modifications to relations do not violate the functional dependencies despite the fact that the relation is not properly normalized. Note that, in several cases, the actions taken to enforce the functional dependencies are one of several possible solutions and the choice depends on the semantics of the data.

```
delete shipments(s#=s, p#=p)
-> nonvar(s) and nonvar(p) and shipments(s#=s, p#=p),
   del shipments(s#=s, p#=p).
-> nonvar(s) and nonvar(p) and not shipments(s#=s, p#=p).
```

The first rule of this procedure assumes that the key `(s#,p#)` uniquely identifies a tuple. The second rule allows the deletion of non-existing tuples.

```
insert shipments(s#=s, sname=n, saddr=a, p#=p, qty=q)
-> nonvar(s) and nonvar(n) and nonvar(a) and nonvar(p) and
   nonvar(q) and not shipments(s#=s),
   ins shipments(s#=s, sname=n, saddr=a, p#=p, qty=q).
-> nonvar(s) and nonvar(p) and nonvar(q) and
   not shipments(s#=s, p#=p) and
   shipments(s#=s, sname=n, and saddr=a),
   ins shipments(s#=s, sname=n, saddr=a, p#=p, qty=q).
```

This procedure maintains the functional dependencies during insertions to **shipments**. The first rule handles the case when the inserted shipment is the first one for the specified supplier. In the second rule, the specified supplier is the supplier for existing shipments, but does not supply the specified part for any of these shipments. In this case, if the specified (sname,saddr) are inconsistent with those in the database for the specified supplier, then the insertion fails. Otherwise, it succeeds. Note that the procedure assumes that the functional dependencies are enforced at the outset.

```
update shipments(s#=s1, p#=p1;
                 s#=s2, sname=n2, saddr=a2, p#=p2, qty=q2)

/* Update p#; s# remains the same */
-> nonvar(s1) and nonvar(p1) and nonvar(p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   p1<>p2 and s1=s2 and not shipments(s#=s2,p#=p2),
   upd shipments(s#=s1, p#=p1; p#=p2, qty=q2),
   update shipments(s#=s2; sname=n2, saddr=a2).

/* Update s# to a new supplier; p# may be updated as well;
   supplier name and address must be given  */
-> nonvar(s1) and nonvar(p1) and nonvar(s2) and s1<>s2 and
   nonvar(n2) and nonvar(a2) and
   ((nonvar(p2) and p1<>p2) or p1=p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   not shipments(s#=s2),
   upd shipments(s#=s1, p#=p1;
                 s#=s2, p#=p2, sname=n2, saddr=a2, qty=q2).

/* Update s# to existing supplier; p# may updated as well; */
-> nonvar(s1) and nonvar(p1) and nonvar(s2) and s1<>s2 and
   ((nonvar(p2) and p1<>p2) or p1=p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   shipment(s#=s2, sname=x, saddr=y),
   upd shipments(s#=s1, p#=p1;
                 s#=s2, p#=p2, sname=x, saddr=y, qty=q2),
   update shipments(s#=s2; sname=n2, saddr=a2).

/* Update of sname and saddr */
-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2)))
   and shipments(s#=s1, p#=p1, sname=x, saddr=y, qty=q2) and
```

```
   (nonvar(n2) or n2=x) and (nonvar(a2) or a2=y) and
   (x<>n2 or y<>a2),
   upd shipments(s#=s1, p#=p1; sname=n2, saddr=a2),
   update shipments(s#=s1; sname=n2, saddr=a2).

/* Termination rules */
-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2)))
   and nonvar(n2) and var(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists x (shipments(s#=s1, sname=x) and x<>n2).

-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2)))
   and var(n2) and nonvar(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists y (shipments(s#=s1, saddr=y) and y<>a2).

-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2)))
   and nonvar(n2) and nonvar(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists x y (shipments(s#=s1, sname=x, saddr=y) and
   (x<>n2 or y<>a2)).
```

This procedure maintains the functional dependencies during updates to the relation `shipments`. The rules isolate updates to different fields; one rule updates one field, and requests all other updates recursively. The first rule performs all updates to part numbers (`p#`) when the supplier number (`s#`) remains the same. Any further updates of supplier name and addresses are handled by a recursive request. The second and third rules handle all cases where the supplier is updated. For the second rule, the update must be updating the supplier number to a number that is not already in the database. If this is the case, then the supplier name and address must also be given, and the execution terminates. The third rule updates the supplier number to a number for an existing supplier. A shipment tuple satisfying the selection criterion is updated with the new supplier number given the name and address of another shipment in the database with the new supplier number. Any further updates of supplier name and address are handled by a recursive request and performed by the fourth rule. To maintain the functional dependencies, all shipments with the same supplier number

must be updated with the new supplier name and address, and this is also handled by a recursive request to the same procedure. The second, fifth, sixth, and seventh rules terminate execution of the procedure. The fifth, sixth, and seventh rules have empty action clauses, and serve just to successfully terminate the procedure. ∎

Example B.5 Dependency Preservation

It is often impossible both to preserve intra-relational dependencies and to obtain Boyce-Codd normal form (BCNF) in a relational scheme. The well-known SJT example from [Dat86] illustrates this problem. The database designer wants to model class enrollment with the relation `enroll(student,teacher,subject)`. However, there are two restrictions about the class enrollment that must be observed. First, each student takes a given subject from only one teacher; this is represented by the functional dependency (`student`, `subject`) → `teacher`. Second, each teacher teaches only one subject, but there may be several teachers of the same subject; this is represented by the functional dependency
`teacher` → `subject`. So the original relation, `enroll(student,teacher,subject)`, is not in BCNF since `teacher` → `subject` and `teacher` is not a candidate key of `enroll`. Hence, the original relation is replaced with the two projections `roster(student,teacher)` and `courses(teacher,subject)`, in which the original intra-relational functional dependency (`student`, `subject`) → `teacher` is no longer preserved. If the two projections are modified independently, they may violate this dependency. However, this functional dependency can be expressed as an inter-relational constraint by specifying modification procedures for the insert and modification requests of the two projections. The following are the modification procedures for `insert` requests:

```
insert roster(student=x, teacher=y)
-> nonvar(x) and nonvar(y) and
   not roster(student=x, teacher=y)
   and not exists  y1 z (roster(student=x, teacher=y1) and
      courses(teacher=y1, subject=z) and
      courses(teacher=y, subject=z) and
       y1 <> y)),
   ins roster(student=x, teacher=y).
-> roster(student=x, teacher=y).

insert courses(teacher=y, subject=z)
-> nonvar(y) and nonvar(z) and
   not courses(teacher=y) and
```

```
   not exists x  y1 (roster(student=x, teacher=y1) and
       courses(teacher=y1, subject=z) and
       roster(student=x, teacher=y) and
       y1 <> y)),
   ins courses(teacher=y, subject=z).
-> courses(teacher=y, subject=z).
```

Notice that the last rule in each procedure enforces a set-semantics: attempts to insert a duplicate tuple returns true without modifying the database. ∎

Appendix C

Correctness of the Update Dependency Language Execution Strategies

In this appendix we prove the correctness of the execution strategies developed in Sections 4.4 and 4.5 for the Update Dependency Language.

Proof of Correctness for PDFE

In this section we prove Theorem 4.6. In order to show the correctness of PDFE we must show the following:
(A) If there is a successful execution, within the maximum depth d, of a PAN given a start state $X_s = \langle S_s, D_s \rangle$ then PDFE returns success with final state $X_f = \langle S_f, D_f \rangle$ such that D_f is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs in one of the solution pyramids, PYR, to D_s and S_f is a superset of S_s including the substitutions obtained (via parameter passing and tuple instantiation) from the execution of PYR. (B) If there is no such solution, PDFE fails and the database state is restored to some time preceding the execution of the PAN. For the root node PAN, the database state is restored to the database state immediately preceding the execution of the PAN.

We begin by proving several lemmas that guarantee certain properties about PDFE of RAN and PAN nodes and chains.

Lemma C.1 (Preservation of Database State Upon Failure) If PDFE returns failure for a node N (or chain with TIN node N) for start state $X_s = \langle S_s, D_s \rangle$, then the database state is restored to some time preceding the execution of N.

Proof: (Lemma C.1)

The backtrack stack is initialized with one element, $\langle \texttt{FAIL}, S_s, 0, TS_s \rangle$ where S_s is the input substitution from the user request and TS_s is the timestamp before the execution of the procedure. All subsequent states, $\langle N, S, D, TS \rangle$ are pushed onto the stack in increasing order of timestamp, and hence popped in decreasing order of timestamp), and S is the substitution of N's parent. When a node P (or chain with TIN node P) fails, backtracking occurs by popping the top stack state, $\langle N_t, S_t, D_t, TS_t \rangle$, rolling back the database state to time TS_t, and restoring the substitution to S_t. Since stack states are popped in decreasing order of timestamp, TS_t can be no later than the most recent choice point, and S_t is the start substitution for N_t's parent (which is also an ancestor of P). Therefore, Lemma C.1 holds. In the case that P is the root node, the database state is restored to the time immediately preceding the procedure execution (i.e. the database remains unchanged). □

Lemma C.2 (Exhaustive Search before Failure for PDFE) (A) PDFE returns failure for a node N given start state $X_s = \langle S_s, D_s \rangle$ only after executing all sub-solution pyramids of the node whose height is less than the maximum depth d. (B) PDFE returns failure for a chain $C = \langle N, A_1, \ldots, A_n \rangle$ given start state $X_s = \langle S_s, D_s \rangle$ only after executing all sequences of sub-solution pyramids rooted at the A_i such that $X_{s_1} = \langle S_s \cup N, D_s \rangle, X_{s_i} = X_{f_{i-1}} \text{for} i > 1$, and the height of each pyramid is less than the maximum depth d.

In order to prove this lemma, we first prove the following three lemmas that assume this lemma holds for a particular type of node and a fixed maximum depth d.

Lemma C.3 Suppose Lemma C.2 holds for RANs when $d = k$. Then Lemma C.2 holds for PANs when $d = k + 1$.

Proof: (Lemma C.3)

Suppose N is a PAN and $d = k + 1$. When N is first visited, it chooses one RAN to execute and pushes the remainder of the RANs on the stack. Suppose N fails and some sub-solution pyramid with depth no greater than $k+1$ has not been executed. This sub-solution pyramid must contain one of the RANs, say R. But if N fails, then an execution of R is attempted. So there must be some sub-solution pyramid of R with depth no greater than k that is not executed, a contradiction of our assumption. □

Lemma C.4 Suppose Lemma C.2 holds for PANs when $d = k+1$, and consider the sequential PDFE of the actions $\langle A_1, \ldots, A_n \rangle$ for the start state $X_s = \langle S_s, D_s \rangle$ such that the start state S_{s_i} of $A_i, i > 1$ is the final state $S_{f_{i-1}}$ of A_{i-1} and the start state S_{s_1} of A_1 is S_s. Before A_1 fails, all successful executions of the sequence of sub-solution pyramids rooted at the A_i that have depth no greater than $k+1$ and that satisfy the start and final state requirements have been executed.

Proof: (Lemma C.4)
Consider a sequence, *SEQ*, of sub-solution pyramids such that the start and final states satisfy the above requirements and the depth of each sub-solution pyramid is no greater than $k+1$. Suppose A_1 fails and *SEQ* is not executed. Let $PYR_1, PYR_2, \ldots PYR_n$ represent the sub-solution pyramid's of *SEQ*.

Suppose there is only one action A_1 in *SEQ*. Then PYR_1, which has depth no greater than $k+1$, is not executed when A_1 backtracks, a contradiction of our assumption. So when there is one action in the *SEQ*, Lemma C.4 holds.

Assume Lemma C.4 holds when there are $n-1$ actions. Suppose there are n actions. Before A_1 fails, all of its sub-solution pyramids within depth no greater than $k+1$ will be executed. In particular PYR_1. The execution of PYR_1 generates a substitution $S_{f_1} = S_{s_2}$ that leads to a successful execution of the sequence $\langle A_2, \ldots, A_n \rangle$. A_2 must fail before A_1 fails. But, by assumption, each time the sequence of actions $\langle A_2, \ldots, A_n \rangle$ are executed all sequences of sub-solution pyramids with depth no greater than $k+1$ are executed before A_2 fails. In particular, $PYR_2, \ldots, PYR_n$. This means that PDFE executes $PYR_1, PYR_2, \ldots PYR_n$, a contradiction. □

Lemma C.5 Suppose Lemma C.2 holds for PANs when $d = k+1$. Then Lemma C.2 holds for chains when $d = k+1$.

Proof: (Lemma C.5)
Suppose C is a chain $\langle N, A_1, \ldots, A_n \rangle$ and $d = k+1$. Suppose, given some start state S_s, C fails and there is some sequence Q of sub-solution pyramids for the execution of the actions $\langle A_1, \ldots, A_n \rangle$, given start state $S_s \cup N$, that has not been executed. In PDFE, a chain returns failure when its first node (i.e. A_1) fails. Since Lemma C.2 holds for PANs when $d = k+1$, then given start state $S_{s_1} = S_s \cup N$, Q must execute before A_1 fails. So C must

execute Q before C fails, a contradiction. □

Proof: (Lemma C.2)

We now proceed to prove Lemma C.2.

Suppose N is a PIN (or the TIN of a chain whose actions are only PINs). Then Lemma C.2 is trivially satisfied, since the first time N is visited its only sub-solution pyramid is executed. Now, we proceed to prove the case when N is a PAN, RAN, or TIN of a chain whose actions contain PANs. We prove this by induction on the maximum depth bound.

Base Case: $d = 2$. The only nodes that have a non-trivial sub-solution pyramid within depth bound 2 are RANs. When N is first visited, it selects all TINs whose tuples are consistent with S_s and satisfy N's condition on D_s. Suppose there are no qualifying tuples. Then N fails and there are no sub-solution pyramids of N. Suppose there are qualifying tuples. PDFE chooses one such TIN to execute and pushes the remaining TINs on the stack. Suppose N fails and some sub-solution pyramid has not been executed. This sub-solution pyramid must include one of the RAN's TINs and one of the TIN's chains. This chain contains only PIN actions since it must be a solution within the maximum depth 1, so any execution of the chain will succeed. If N fails, then an execution of the TIN's chain is attempted and succeeds, a contradiction.

Hypothesis: Assume Lemma C.2 holds for RANs when $d = k$.

Inductive Step: Show that Lemma C.2 holds if N is a RAN when $d = k+2$. Since Lemma C.2 holds for RANs when $d = k$, then by Lemma C.3, Lemma C.2 holds for PANs when $d = k+1$. Since Lemma C.2 holds for PANs when $d = k + 1$, then by Lemma C.5, Lemma C.2 holds for chains when $d = k + 1$.

Suppose N is a RAN and $d = k + 2$. When N is first visited, it selects all TINs whose tuples are consistent with S_s and satisfy N's condition on D_s. Suppose there are no qualifying tuples. Then N fails and there are no sub-solution pyramids of N. Suppose there are qualifying tuples. PDFE chooses one such TIN to execute and pushes the remaining TINs on the stack. Suppose N fails and some sub-solution pyramid has not been executed. This sub-solution pyramid must be the execution of the chain for one of the TINs within the maximum depth $k + 1$. But Lemma C.2 holds for chains when $d = k + 1$, so the chain must have been executed, a contradiction.

So Lemma C.2 holds for a RAN at an arbitrary depth. Hence Lemma C.2 also holds for PANs and chains at arbitrary depth by Lemma C.3 and Lemma C.5. □

The following two lemma's guarantee the correct execution of chains and RAN's.

Lemma C.6 (Correctness of PDFE of Chains) If there is a successful execution, within the maximum depth d, of a chain $\langle T, A_1, \ldots, A_n \rangle$ for a given start state $X_s = \langle S_s, D_s \rangle$, the PDFE returns success with final state $X_f = \langle S_f, D_f \rangle$ such that (1) D_f is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs for the sequence (according to the order of the nodes in the chain) of sub-solution pyramids rooted at the nodes of the chain, and (2) $S_f = S_{f_n}$, and for all nodes A_i, S_{f_i} includes the substitution implied by T, is a superset of S_s, and is a superset for S_{s_j} for all start states for the nodes A_j that precede A_i. (B) If there is no such execution, PDFE fails and the database state is restored to some time preceding the execution of the chain.

Lemma C.7 (Correctness of PDFE of RANs) (A) If there is a successful execution, within maximum depth d, of a RAN R given a start state $X_s = \langle S_s, D_s \rangle$, then PDFE returns success with final state $X_f = \langle S_f, D_f \rangle$ such that D_f is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs in a sub-solution pyramid PYR rooted at R to D_s and S_f is a superset of S_s including the substitutions obtained (via parameter passing and tuple instantiation) from the execution of PYR. (B) If there is no such execution, PDFE fails and the database state is restored to some time preceding the execution of the RAN.

In order to prove Lemma C.6 and C.7, we first prove the following three lemmas that assume Lemma C.6 holds for a particular type of node and a fixed maximum depth d.

Lemma C.8 If Lemma C.6 holds when $d = k$, then Lemma C.7 holds when $d = k + 1$.

Proof: (Lemma C.8)
Case 1: Proof of (A). Assume there is at least one successful execution, within maximum depth $k + 1$, of RAN R given start state $X_s = \langle S_s, D_s \rangle$.

Assume the execution of R returns failure. When R is first visited, it selects all TINs whose tuples are consistent with S_s and satisfy R's condition on D_s. It chooses one such TIN to execute and pushes the remaining TINs on the stack. R only fails if there are no qualifying tuples or when all of its TINs have been executed and the chain for the last TIN fails. Since R's condition evaluates to true, then there is at least one qualifying tuple. Since there is a

successful execution with depth no greater than $k+1$ for R for start state $\langle S_s, D_s \rangle$, then there is a solution with depth no greater than k for at least one of the chains given start state $\langle S_s, D_s \rangle$. Let T be the first TIN for which such a chain exists. Then, by our assumption, the execution of the chain for T returns success to R with some final state $X_f = \langle S_f, D_f \rangle$ as described in Lemma C.6. But R returns success as soon as it finds a successful chain, a contradiction. So R returns success with final state $X_f = \langle S_f, D_f \rangle$.

The sub-solution pyramid, PYR, rooted at R consists of the successful chain as the only child of R. Since D_f satisfies the requirements for the chain, then D_f is a left-to-right traversal of the doit-action PINs for PYR. Since S_s is the start state of the chain, S_f is a superset of S_s and S_f contains all substitutions obtained from the execution of PYR.
Case 2: Proof of (B). Suppose there is not a successful execution, within maximum depth k+1, of the RAN R for $X_s = \langle S_s, D_s \rangle$. Then either R's condition fails, or all of the chain's do not have a successful execution within depth no greater than k. Suppose R's condition fails. Then the execution of R returns failure and X_s is not modified. Suppose the condition does not fail. Then the execution tries to execute the chain for each TIN whose tuple satisfies the condition. Let C be such a chain. By our assumption and Lemma C.6, the execution of C fails. All such executions of R's chains fail. By Lemma C.1, the database state and substitution are restored to some time previous to the execution of R. □

Lemma C.9 If Lemma C.7 holds when $d = k+1$, then Theorem 4.6 holds when $d = k+2$.

Proof: (Lemma C.9)
Case 1: Proof of (A). Assume there is at least one successful execution, within maximum depth $k+2$, of PAN P given start state $X_s = \langle S_s, D_s \rangle$. Then there must exist a solution for at least one of the RANs given start state $X_s = \langle S_s, D_s \rangle$. The resulting state X_f is the final state of exactly one of the PAN's RANs.

Assume that the execution of P returns failure. Now, when P is first visited, it chooses one RAN to execute and pushes the remainder of the RANs on the stack. P only fails when all of its RANs have been executed and the last RAN popped fails. When each RAN is executed, the database state is rolled back to D_s and the substitution is set to S_s. Now, there must exist a solution within maximum depth $k+1$ for at least one of the RANs given start state $X_s = \langle S'_s, D_s \rangle$, where S'_s is the binding of S_s to the input parameters of P. Let R be the first such RAN. Then, by our assumption, R returns success with some final state

$X_f = \langle S_f, D_f \rangle$ as described in Lemma C.8. But P returns success as soon as one of its RANs returns success, a contradiction. So P returns success with final state $X_f = \langle S'_f, D_f \rangle$, where S'_f is the binding of S_f to the output parameters of P.

The sub-solution pyramid, PYR, rooted at P consists of the successful RAN R as the only child of P. Since D_f satisfies the requirements for the RAN R, and the addition of P as the root does not add any doit-action PINs or any new variable, then D_f is a left-to-right traversal of the doit-action PINs for PYR. Since S'_s binds values of S_s into the scope of the execution of P, S_f is a superset of S'_s, S_f contains substitutions obtained from the execution of the sub-pyramid rooted at R, and S'_f binds values of S_f back into the scope of the activation of P, then S'_f is a superset of S_s and contains substitutions obtained from the execution of the sub-pyramid rooted at P.

Case 2: Proof of (B). Suppose there is not a successful execution, within maximum depth $k+2$, of PAN P for $X_s = \langle S_s, D_s \rangle$. The execution tries to execute all of P's RAN's. Let R be such a RAN. By our assumption, the execution of R fails, so all such executions of P's RANs fail and therefore P fails. By Lemma C.1, the database state and substitution are restored to some time previous to the execution of the P. □

Lemma C.10 If Theorem 4.6 holds for $d = k$, then Lemma C.6 holds for chains of depth k.

Proof: (Lemma C.10)

Base Case: C contains one action.

Case 1: Proof of (A). Assume there is at least one successful execution, within maximum depth k of chain $C = \langle T, A_1 \rangle$ given start state $X_s = \langle S_s, D_s \rangle$. The resulting state $X_f = \langle S_s, D_f \rangle$ is the final state of the the execution of A_1.

A_1 is a PAN since we are only considering chains with at least one PAN. Now, if there is one successful execution of C within depth bound k for start state X_s, then there is a successful execution of A_1 within depth bound k for start state $\langle S_s \cup T, D_s \rangle$. Since Lemma C.7 holds for $d = k$, then PDFE returns success for A_1 with some resulting state $X'_f = \langle S'_s, D'_f \rangle$. Then PDFE returns success for C with resulting state $X_f = X'_f$.

Case 2: Proof of (B). Suppose there is not a successful execution, with maximum depth k of chain C. Then there is not a successful execution with maximum depth k of A_1. We assume that Theorem 4.6 holds for $d = k$, so A_1 fails. By Lemma C.1, the database state

and substitution are restored to some time, TS, previous to the execution of the A_1. Since C does not contain any previous choice points, TS is previous to C as well.
Hypothesis: Assume Lemma C.10 holds when a chain contains $n-1$ actions.
Induction Step: Show Lemma C.10 holds when C contains n actions.
Case 1: Proof of (A). Assume there is at least one successful execution, within maximum depth k of chain $C = \langle T, A_1, \ldots, A_n \rangle$ given start state $X_s = \langle S_s, D_s \rangle$. The resulting state $X_f = \langle S_s, D_f \rangle$ is the final state of the the execution of A_n.

Suppose PDFE returns failure for C. Then PDFE returns failure for $C' = \langle T, A_1, \ldots, A_{n-1} \rangle$ given start state X_s. But by Lemma C.2, this only happens after PDFE executes all sequences of sub-solution pyramids rooted at the A_i such that $X_{s_1} = \langle S_s \cup T, D_s \rangle$ and $X_{s_i} = X_{f_{i-1}}$ for $i > 1$. So at some point, the execution of C' must execute at least one sequence of sub-solution pyramids that result in a final state X in SET. So PDFE must return failure for the execution of A_n with start state $X = \langle S_{n-1}, D_{n-1} \rangle$ when $d = k$. But, by assumption Theorem 4.6 holds for $d = k$, and there is a successful execution, within depth bound k, of A_n with start state X, so PDFE returns success, a contradiction.

By the hypothesis, D_{n-1} is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs for the sequence (according to the order of the nodes in the chain) of sub-solution pyramids rooted at the nodes of the chain C' to D_s; by Theorem 4.6, D_n is obtained by applying the left-to-right traversal of the doit-action PINs for the sub-solution pyramid rooted at A_n; so D_n is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs for the sequence of sub-solution pyramids rooted at the nodes of chain C. When a chain is executed the final state X_f is set to the final state of the last node in the chain, so $S_f = S_{f_n}$ by definition. By assumption, for all nodes $A_i, i < n, S_{f_i}$ includes the substitution implied by T, is a superset of S_s, and is a superset for S_{s_j} for all start states for the nodes A_j that precede A_i. By Lemma C.9, S_{f_n} is a superset of S_{s_n} (which is the same as $S_{f_{n-1}}$) including the substitutions obtained the execution of solution pyramid rooted at A_n. So S_{s_n} includes the substitutions implied by T, is a superset of S_s and is a superset for S_{s_j} for all start states of the nodes A_j that precede A_i.
Case 2: Proof of (B). Suppose there is not a successful execution, with maximum depth k of chain C with n actions. Suppose that PDFE finds a successful execution. Then PDFE returns success for C' with start state X_s and final state X, and PDFE returns success for A_n with start state X. By our hypothesis, there must be a successful execution of C' with start

state X_s resulting in final state X, and by Theorem 4.6, there must be a successful execution of A_n with start state X. Then there must be a successful execution of C, a contradiction. Therefore the execution of chain C must return failure, and by Lemma C.1, the database is rolled back to some time previous to the execution of the chain. □

We now use Lemmas C.8, C.9, and C.10 to prove Lemma C.6 by induction on the maximum depth d.

Proof: (Lemma C.6)

Base Case: Lemma C.6 holds when $d = 1$.

If there is a successful execution, within the maximum depth 1, of the chain then either there are no A_i (i.e. $n = 0$) or all A_is are PINs. Otherwise the execution exceeds the maximum depth. Since PINs always succeed, PDFE will return success. The nodes are evaluated in the order in which they appear, therefore the modifications for the doit-action PINs are executed in the order in which they appear and D_f is obtained correctly. When the chain is chosen by its parent for execution, the substitution is updated with T, so S_{s_1} contains T. Assume that the substitution S_{f_i} is correct for $i < k$. Consider S_{f_k}. By definition S_{s_k} is the same as $S_{f_{k-1}}$, which is correct. Now suppose A_k is a doit-action PIN or a write i/o-action PIN. Then $S_{f_k} = S_{s_k}$ and is correct. Suppose that A_k is a read i/o-action PIN. Then $S_{f_k} = S_{s_k}$ augmented with the value of the external read.

Suppose there is not a successful execution, within maximum depth 1, of the chain for X_s. Then one of the A_i must be a PAN (otherwise there is a successful execution of the chain since PINs do not fail). When the first PAN is executed, the execution fails because the maximum depth bound is violated. By Lemma C.1, the database state and substitution are restored to some time previous to the execution of the chain.

Hypothesis: Lemma C.6 holds when $d = k$.

Inductive Step: Show that Lemma C.6 holds when $d = k + 2$. By Lemma C.8, Lemma C.7 holds for $d = k+1$, and by Lemma C.9, Theorem 4.6 holds for $d = k+2$. So by Lemma C.10, Lemma C.6 holds when $d = k + 2$. □

Proof: (Lemma C.7)

This lemma is a consequence of Lemma C.8 and Lemma C.6. □

Proof: (Theorem 4.6)

The correctness of PDFE is a consequence of Lemma C.9 and Lemma C.7. □

Proof of Correctness for PCE

In this section we prove Theorem 4.7. In order to show the correctness of PCE we must show the following:
(A) If there is a successful execution, within maximum depth d, of a root node PAN given a start state $X_s = \langle S_s, D_s \rangle$ then PCE returns success with some final state $X_f = \langle S_f, D_f \rangle$ such that D_f is obtained by applying the sequence of modifications defined by the left-to-right traversal of the doit-action PINs in one of the solution pyramids, PYR, to D_s and S_f is a superset of S_s including the substitutions obtained (via parameter passing and tuple instantiation) from the execution of PYR. (B) If there is no such solution, PCE fails and the database state is restored to the state immediately preceding the execution of the PAN.

We begin by proving several lemmas that guarantee certain properties about RAN and chain processes.

Lemma C.11 (Exhaustive Search before Failure for PCE) A RAN process, P_r, returns failure for a given start state X_s if and only if all sub-solution pyramids for X_s whose height is less than the maximum depth d have been executed.

In order to prove this lemma, we first prove the following lemma that assumes Lemma C.11 holds for a particular type of node and a fixed maximum depth d.

Lemma C.12 If Lemma C.11 holds for RAN processes when $d = f$, then a chain process, P_c, for a chain $\langle N, A_1, \ldots, A_n \rangle$ returns failure for a given start state $X_s = \langle S_s, D_s \rangle$ if and only if all sequences of sub-solution pyramids rooted at the A_i such that $X_{s_1} = \langle S_s \cup N, D_s \rangle$, $X_{s_i} = X_{f_{i-1}}$ for $i > 1$, where the height of each pyramid is less than the maximum depth $d = f + 1$ have been executed.

Proof: (Lemma C.12)
Assume Lemma C.11 holds for RAN processes when $d = f$.
(only if) Let C be a chain $\langle T, A_1, \ldots, A_n \rangle$ that contains sub-solution pyramid of depth $d = f + 1$ and let P_c be the process spawned for the chain. Suppose that P_c fails and that

there is some sequence $S = \langle S_1, \ldots, S_n \rangle$ of sub-solution pyramids as defined above that is not executed.

As P_c executes the chain, it pushes all non-failed RANs for each of the PANs on a backtrack stack. Let A_j be the first PAN action in the sequence. P_c fails only when its backtrack stack becomes empty during backtracking. This only occurs when all RAN processes spawned for A_j fail for the given start state updated with the actions $A_1, \ldots, A_{j-1}$.
Base Case: Suppose P_c contains only one action. If A_1 is a PIN, then there is exactly one sub-solution pyramid for the chain and S must be it. P_c will succeed since the PIN will succeed, a contradiction. Therefore A_1 is a PAN. P_c only fails when all RAN processes spawned for the PAN fail. Therefore, there is some sub-solution pyramid for one of the failed RAN processes within depth f that is not executed. This contradicts the assumption.
Hypothesis: Assume that Lemma C.12 holds for chains containing $n-1$ actions.
Inductive Step: Show Lemma C.12 holds for chains containing n actions.

Recall that S is unexecuted sequence of sub-solutions. Consider the sequence $S' = \langle S_1, \ldots, S_{n-1} \rangle$ and the chain $C' = \langle T, A_1, \ldots, A_{n-1} \rangle$. By the hypothesis, the process executing C' does not fail without executing the sequence S'. Therefore, at some point P_c executes S' and then looks for a sub-solution pyramid for A_n. If A_n is a PIN, then there is exactly one sub-solution pyramid which must be S_n, so S is executed, a contradiction. Hence A_1 must be a PAN. S_n is not found only if all RAN processes spawned for A_1 fail. But S_n is within depth f, a contradiction. Therefore Lemma C.12 holds for all chains.
(if) Let C be a chain $\langle T, A_1, \ldots, A_n \rangle$ that does not have a sub-solution pyramid of depth $d = f + 1$ and let P_c be the process spawned for the chain. Suppose that P_c succeeds. Then there must be some RAN process that succeeds and does not have a sub-solution pyramid of depth $= f$. This contradicts our hypothesis. □

Proof: (Lemma C.11)
Base Case: $d = 2$.

The only nodes that have a non-trivial sub-solution pyramid within depth bound 2 are RANs. Let P_r be the RAN process for a RAN node N.
(only if) Suppose that N contains a sub-solution pyramid of depth 2 that is not executed before P_r fails. When P_r is first spawned, it spawns a chain process for all TINs whose tuples are consistent with S_s and satisfy N's condition on D_s. Since there is a sub-solution pyramid, then there must be one such tuple. Let P_c be the process for the chain of the

sub-solution pyramid that is not executed. Then P_c must return failure since P_r fails only when all of its spawned chain processes fail. But P_r is of depth 1 so it must contain only TINs and PINs, which do not fail, a contradiction.
(if) Suppose that N does not contain a sub-solution pyramid of depth 2 but P_r succeeds. Then there must be some chain process that returns success but does not contain a solution. Since the depth is 2, this chain process contains only PIN nodes. But such a process succeeds, a contradiction.
Hypothesis: Assume Lemma C.11 holds for RAN processes when $d = k$.
Inductive Step: Show Lemma C.11 holds for RAN processes when $d = k + 2$.
(only if) Suppose a RAN process P_r returns failure for a given start state X_s and there is some sub-solution pyramid, PYR, that was not executed. P_r returns failure only if all its chain processes return failure, so there must be some chain process that returns failure that does not execute the sub-solution pyramid $PYR' = PYR$ without its root node RAN. This contradicts the hypothesis and Lemma C.12. Therefore Lemma C.11 holds for all RAN processes.
(if) Suppose P_r returns success. Then there must be some chain process that returns success but does not contain a solution for depth $k + 1$. But this contradicts the hypothesis and Lemma C.12. □

Proof: (Theorem 4.7)
Case 1: Proof of (A). Suppose a root node PAN has a solution pyramid for the start state $X_s = \langle S_s, D_s \rangle$. Then this solution pyramid contains a sub-solution pyramid for one of the RANs. The root node PAN spawns a RAN process for each of the PAN's RANs. By Lemma C.11, if there is a sub-solution pyramid for one of the RANs, the RAN will return success.

S_f is correct since each spawned process is initialized accordingly (accounting for parameter passing), and the final substitution of each process is the final substitution of exactly one of its subprocesses.

D_f is correct since the effects of only the chosen processes remain. The root node finds a successful rule immediately, terminates, and undoes the effects of all unchosen suspended processes. The effects of failed subprocesses is undone when the subprocess fails.
Case 2: Proof of (B). Suppose there is no solution pyramid. If the root node process succeeds, then one of the root node's RAN process succeeds. But a RAN process only succeeds

if there is some sub-solution pyramid. If there is a sub-solution pyramid for one of the RANs then there is a solution pyramid for the root node, a contradiction. □

Bibliography

[ABJ89] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In SIGMOD89 [SIG89], pages 253–262.

[ACL91] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In VLDB91 [VLD91], pages 479–487.

[ADJ90] R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Trans. on Database Systems*, 15(3):427–458, September 1990.

[AJ89] R. Agrawal and H. V. Jagadish. Materialization and incremental update of path information. In IEEE Computer Society [IEE89], pages 374–383.

[Arm74] W. W. Armstrong. Dependency structures of data base relationships. In IFIP [IFI74], pages 580–583.

[BBB+90] D. Beech, P. Bernstein, M. Brodie, M. Carey, B. Lindsay, L. Rowe, and M. Stonebraker. Third-generation data base system manifesto. In *Proc. IFIP TC-2 Conf. on Object Oriented Databases.* North-Holland, 1990.

[Ber76] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Trans. on Database Systems*, 1(4):277–298, 1976.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1987.

[BM91] C. Beeri and T. Milo. A model for active object oriented database. In VLDB91 [VLD91], pages 337–349.

[BMHD89] A. Buchmann, D. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In IEEE Computer Society [IEE89], pages 470–480.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. on Database Systems*, 6(4):557–575, December 1981.

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

[CDF+86] M. Carey, D. DeWitt, D. Frank, et al. The architecture of the EXODUS extensible DBMS. In *Proc. ofthe International Workshop on Object-Oriented Database Systems*, September 1986.

[CdM89] J.-P. Cheiney and C. de Maindreville. Relational storage and efficient retrieval of rules in a deductive DBMS. In IEEE Computer Society [IEE89], pages 644–651.

[Cha89] S. Chakravarthy. Rule management and evaluation: An active DBMS perspective. In Sellis [Sel89], pages 20–28.

[CL87] W. W. Chu and L. M-T. Lan. Task allocation and precedence relation for distributed real-time systems. *IEEE Transactions on Computers*, C-36(6):667–679, June 1987.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, Courant Computer Science Symposium, Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N. J., 6th edition, 1972.

[COD73] CODASYL data description language journal of development, June 1973. NBS Handbook 113 (1973).

[Cod74] E. F. Codd. Recent investigations into relational data base systems. In IFIP [IFI74].

[CW89] S. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proc. 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 200–209, Philadelphia, PA, 1989.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In VLDB90 [VLD90], pages 566–577.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In VLDB91 [VLD91], pages 577–589.

[Dat83] C. J. Date. *An Introduction to Database Systems*, volume 2. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1983.

[Dat84] C. J. Date. A critique of the SQL database. *ACM SIGMOD Record*, 1984.

[Dat86] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[DD91] K. Dittrich and U. Dayal. Active database systems, September 1991. Tutorial Notes, 17th Int. Conf. on Very Large Data Bases.

[DE88] L.M.L. Delcambre and J.N. Etheredge. A self-controlling interpreter for the Relational Production Language. In SIGMOD88 [SIG88], pages 396–412.

[DE89] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: A production language for relational databases. In L. Kerschberg, editor, *Expert Database Systems – Proc. from the Second Int. Conf.*, pages 333–351, Redwood City, California, 1989. Benjamin/Cummings.

[dMS88] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proc. of the 4th Int. Conf. on Data Engineering*, pages 234–241, Los Angeles, California, February 1988.

[Don89] G. Dong. On distributed processibility of Datalog queries by decomposing databases. In SIGMOD89 [SIG89], pages 26–35.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule management in object-oriented databases: A uniform approach. In VLDB91 [VLD91], pages 317–326.

[DWE89] L.M.L. Delcambre, J. Waramahaputi, and J.N. Etheredge. Pattern match redution for the Relational Production Language. In Sellis [Sel89], pages 59–67.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19(11):624–633, November 1976.

[Esw76] K.P. Eswaran. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. IBM Research Report RJ 1820, IBM San Jose Research Laboratory, August 1976.

[Fag77] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. on Database Systems*, 2(3):262–278, 1977.

[Fag79] R. Fagin. Normal forms and relational database operators. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 9–36, Boston, Massachusetts, May 1979.

[FC85] A. L. Furtado and M. A. Casanova. Updating relational views. In *Query Processing in Database Systems*. Springer-Verlag, Berlin, 1985.

[For79] C. L. Forgy. *On the Efficient Implementation of Production Systems.* Ph.D. dissertation, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, February 1979.

[For81] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In VLDB91 [VLD91], pages 327–336.

[GLPT76] J. Gray, R. Lorie, F. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proc. IFIP Working Conf. on Modelling of Database Management Systems*, Freudenstadt, 1976. Also available as IBM Research Report RJ1654, San Jose.

[Gra78] J. Gray. *Notes on Database Operating Systems*, volume 60. Springer-Verlag, Berlin, 1978. Also available as IBM Resarch Report RJ2188(30001), San Jose.

[GST90] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of Datalog queries. In SIGMOD90 [SIG90], pages 143–152.

[H+75] G. Held et al. INGRES: a relational database system. In *Proc. 1975 National Computer Conf.*, Anaheim, California, June 1975.

[Han89] E.N. Hanson. An initial report on the design of Ariel: A DBMS with an integrated production rule system. In *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems* [Sel89], pages 12–19.

[HCKW90] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In SIGMOD90 [SIG90], pages 271–280.

[HCL+90] L. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: as the dust clears. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HH91] J. M. Hellerstein and M. Hsu. Determinism in partially ordered production systems. Technical Report Working Paper, IBM Almaden Research Center, January 1991.

[HM89] G. Harhalakis and L. Mark. A knowledge-based prototype of a factory-level cim system. *Journal of Computer Integrated Manufacturing Systems*, 2(1):11–20, 1989.

[How86] L. Howe. Sybase data integrity for on-line applications. Technical report, Sybase, Inc., Emeryville, CA, 1986.

[HR85] F. Hayes-Roth. Rule based systems. *Communications of the ACM*, 28(9):921–932, September 1985.

[HR87] T. Härder and K. Rothermel. Concepts for transaction recovery in nested transactions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 239–248, San Francisco, May 1987.

[HW92] E.N. Hanson and J. Widom. Rule processing in active database systems. In L. Delcambre and F. Petry, editors, *Advances in Databases and Artificial Intelligence*, volume 1. JAI Press, Greenwich, Connecticut, 1992.

[IBM88] IBM. *IBM Knowledge Engineering Environment/370 (KEE/370), User's Guide, Release 1*, December 1988.

[IEE89] *Proc. 5th Int. Conf. on Data Engineering*, Los Angeles, California, February 1989.

[IFI74] *Proc. 1974 IFIP Congress*, Amsterdam, 1974. North Holland.

[ING89] INGRES Corporation. *INGRES/SQL Reference Manual*, November 1989. Version 6.3.

[IS89] Y. Ioannidis and T. Sellis. Conflict resolution of rules assigning values to virtual attributes. In SIGMOD89 [SIG89], pages 205–214.

[Jac86] P. Jackson. *Introduction to Expert Systems.* Addison-Wesley Publishing Co., 1986.

[KdMS90] J. Kiernan, C. de Maindreville, and E. Simon. Making deductive databases a practical technology: A step forward. In SIGMOD90 [SIG90], pages 237–246.

[KKM83] S. Kasif, M. Kohli, and J. Minker. PRISM: A parallel inference system for problem solving. In *Logic Programming Workshop*, pages 123–152, Praia da Falesia, Algarve, Portugal, 1983.

[Knu73] D. E. Knuth. *Sorting and Searching.* The Art of Computer Programming. Addison-Wesley Publishing Co., 1973.

[Law73] E. L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546, January 1973.

[Lin81] C.-C. Lin. *Coupling Production Systems and Database Systems: A Homogeneous Approach.* Ph.D. dissertation, University of Maryland, Department of Computer Science, College Park, MD, 1981.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LS86] E. Shapiro L. Sterling. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[Mar85] L. Mark. *Self-Describing Database Systems - Formalization and Realization.* Ph.D. dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1985. TR-1484.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In SIGMOD89 [SIG89], pages 215–224.

[MF78] J. McDermott and C. Forgy. Production system conflict resolution strategies. In D.A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*, pages 177–199. Academic Press, 1978.

[MHL+91] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 1991.

[Min87] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufmann, Los Altos, California, 1987.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. 9th International Conf. on Very Large Data Bases*, pages 34–42, Florence, Italy, October 1983.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[Mos87] J. E. B. Moss. Log-based recovery for nested transactions. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 427–432, Brighton, England, September 1987.

[MRC91] L. Mark, N. Roussopoulos, and R. Cochrane. Update dependencies in the relational model. Technical Report SRC-TR-91-94, Department of Computer Science, University of Maryland, College Park, Maryland, October 1991.

[Nil80] N. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, California, 1980.

[NR91] A. Stamenas N. Roussopoulos, N. Economou. Adms: A testbed for incremental access methods. *IEEE Trans. on Knowledge and Data Engineering*, 1991.

[Ras90] L. Raschid. Maintaining consistency in a stratified production system program. In *Proc. AAAI National Conf. on Artificial Intelligence*, 1990.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proc. of the 15th Int. Conf. on Very Large Data Bases*, Amsterdam, August 1989.

[RMSF91] N. Roussopoulos, L. Mark, T. Sellis, and C. Faloutsos. An architecture for high performance engineering information systems. *IEEE Trans. on Software Engineering*, 17(1):22–33, January 1991.

[Rou91] N. Roussopoulos. The incremental acces method of view cache: Concept and cost analysis. *ACM Trans. on Database Systems*, June 1991.

[RSD91] L. Raschid, T. Sellis, and A. Delis. On the concurrent execution of production rules in a database implementation. Technical Report UMIACS-TR-91-125,

Institute for Advanced Computer Studies, University of Maryland, College Park, MD, September 1991.

[SEH87] M. Stonebraker, E.Hanson, and C.-H. Hong. The design of the POSTGRES rules system. In *Proc. 3rd Int. Conf. on Data Engineering*, pages 48–66, Los Angeles, CA, February 1987.

[Sel89] T. Sellis, editor. *Special Issue on Rule Management and Processing in Expert Database Systems*, volume 18(3). SIGMOD Record, September 1989.

[Sha87] E. Y. Shapiro, editor. *Concurrent Prolog, Collected Papers.* MIT Press, 1987.

[SHP88] M. Stonebraker, E.N. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Trans. on Software Engineering*, 14(7):897–907, July 1988.

[SHP89] M. Stonebraker, M. Hearst, and S. Potamianos. A commentary on the POSTGRES rules system. In *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems* [Sel89], pages 5–11.

[SIG88] *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, June 1988.

[SIG89] *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, June 1989.

[SIG90] *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, June 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In SIGMOD90 [SIG90], pages 281–290.

[SJR82] M. Stonebraker, R. Johnson, and S. Rosenberg. A rules system for a relational data base management system. In *Proc. 2nd International Conf. on Databases*, pages 323–335, Jerusalem, Israel, June 1982.

[SLR88] T. Sellis, C-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment:concepts and algorithms. In SIGMOD88 [SIG88], pages 404–412.

[SLR90] T. Sellis, C-C. Lin, and L. Raschid. Coupling production systems and database systems: A homogeneous approach. Technical Report UMIACS-TR-87-68.1 (CS-TR-1960.1)(subm. to IEEE TKDE, under revision), Dept. of Computer Science, University of Maryland, March 1990.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I*, volume 1. Computer Science Press, Rockville, Maryland, 1988.

[VLD90] *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990.

[VLD91] *Proc. of the 17th Int. Conf. on Very Large Data Bases*, Barcelona (Catalonia, Spain), September 1991.

[WCL91a] J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In VLDB91 [VLD91], pages 275–285.

[WCL91b] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. IBM Research Report RJ 7979, IBM Almaden Research Center, February 1991.

[WF89a] J. Widom and S.J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems (extended abstract). In *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems* [Sel89], pages 36–45.

[WF89b] J. Widom and S.J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. Technical Report IBM Research Report RJ 6880, IBM Almaden Research Center, June 1989. Revised March 1990.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In SIGMOD90 [SIG90], pages 259–270.

[Wid91] J. Widom. Active database rule systems, September 1991. Tutorial Notes, EDBT Summer School On Advances in Database Technology.

[WO90] O. Wolfson and A. Ozeri. A new paradigm for parallel and distributed rule-processing. In SIGMOD90 [SIG90], pages 133–142.

[Wol88] O. Wolfson. Sharing the load of logic program evaluation. In *Proc. of the Intl. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, 1988.

[WS88] O. Wolfson and A. Silberschatz. Distributed processing of logic programs. In SIGMOD88 [SIG88], pages 329–336.

[XP90] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[ZB90] D. R. Zertuche and A. P. Buchmann. Execution models for active database systems: A comparison. Technical Report TM-0238-01-90-165, GTE Laboratories Incorporated, January 1990.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In Francois Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *Proc. International Conference on Extending Data Base Technology*, volume 416 of *Lecture Notes in Computer Science*, pages 361–370, Venice, March 1990. Advances in Database Technology - EDBT '90, Springer-Verlag.