Chapter 3

Applications of Active Databases

Active rules satisfy different application needs. Broadly speaking, we distinguish between internal and external applications of active rules:

- The applications of active rules to support classical database management features, such as integrity maintenance, derived data maintenance, and replication management, are denoted as *internal*; these rules are often system-generated and hidden to the user, and the generation process is fully automatic. In many cases, internal rules can be *declaratively specified* (e.g., in the format of integrity constraints, data derivation rules, or requirements on replicated copies). Although not yet widespread, several other internal applications of active rules can be envisioned, including version maintenance, security administration, demand logging, and event tracking. The class of internal applications can be extended to incorporate rules that, although not relevant to database management, can be built from abstract specifications, such as rules for workflow management.

- Other rules embed applicative knowledge, and are classified as *external*; these are often called *business rules*, as they perform a part of the business computation that normally would be contained in the application code. Examples of business rules are trading rules following bond fluctuations, inventory management rules following stock variations, and energy management rules to respond to changes in power distribution requirements. A significant class of business rules is *alerters*; actions of these rules consist of simply giving a warning or display-

effectively with software evolution and schema changes. Following an introduction to object-oriented databases, the last part of this book presents an in-depth discussion of the topic of schema evolution.

In introducing these areas of database research, we have built on our extensive experience in teaching these advanced topics in regular courses. The various parts are structured in a didactic progression that offers the flexibility of stopping at a level selected by the course instructor; moreover, each chapter is endowed with an extensive bibliography and a rich set of exercises. Many exercises combine problems from different parts of the book, thus reinforcing the many technical threads that connect the various topics of the book and bringing different areas of database research a step closer together.

Tomado del libro:
Advanced Database Systems
Zaniolo, Ceri, Faloutsos, Snodgrass,
Subrahmanian, Zicari
Morgan Kaufmann Publishers 1997
Caps. 2 (sec. 2.2) y Cap. 3

9

Part I

Active Databases

The first part of this book consists of Chapters 2, 3, and 4, which are dedicated to active databases. Active databases provide a facility, tightly integrated with the database system software, for creating and executing production rules. These rules follow the so-called *Event-Condition-Action* paradigm; they autonomously react to events occurring on the data, by evaluating a data-dependent condition, and by executing a reaction whenever the condition evaluation yields a truth value. The processing of rules is performed by an *active rule engine*, which monitors events caused by database transactions and schedules rules according to given policies; the resulting processing guarantees a *reactive database behavior*, which contrasts with the passive behavior of conventional database systems.

When a database exhibits a reactive behavior, a large fraction of the semantics that is normally encoded within applications can be expressed by means of active rules, giving to database applications a new independence dimension, called *knowledge independence*: applications are freed from the need of expressing the knowledge about reactive processing, which is instead coded in the form of active rules. Consequently, knowledge is encoded once and for all in the schema and is automatically shared by all users; knowledge modification is managed by changing the content of rules and not by changing applications. Clearly, knowledge independence introduces a new style in database programming; for this reason, active rule design becomes a relevant part of database design. This explains why in the final chapter of this part (Chapter 4) we concentrate on design methods for active rules.

This part of the book is organized as follows. Chapter 2 provides an introduction to the alternative semantics and basic features of active rules and discusses current trends toward commercial consolidation and standardization of these features. The chapter is organized bottom up; first, four active

rule systems are presented in turn, and next their common features are abstracted and generalized, introducing as well some features not present in any of these four systems. Our bottom-up approach is motivated by the lack of a universal model for active databases, in spite of the existence of many research prototypes and commercial products. Indeed, the search for higher conceptual models and for a unifying theory is one of the factors that make active databases an interesting open field for research.

Chapter 3 is dedicated to the applications of active rules and introduces the classical uses of active rules for modeling integrity maintenance, data derivation, data replication, workflow management, and business rules. A large example shows the use of active rules for energy management.

Chapter 4, the final chapter of this part, is dedicated to active rule design methods and concentrates on rule analysis, modularization, and prototyping. With this shift of the discussion, we describe general concepts and problems, such as modularization and termination, that apply to all the different active database systems. At the end of the chapter, we sketch the features of the IDEA Methodology for active rule design, prototyping, and implementation, and the tools that can be used for supporting the methodology.

Chapter 2

Syntax and Semantics of Active Databases

The first chapter of this part is dedicated to the understanding of the syntax and semantics of active rules. This topic is developed by examples, through the careful analysis of four representative active database systems. We start with the Starburst active rule system and conclude with Chimera. These are research prototypes for relational and object-oriented databases, respectively, that emphasize advanced semantic features, such as set-oriented database processing, sophisticated event composition and management, and the ability to access transition values representing past states. In between these research prototypes, we present Oracle and DB2, two of the most popular relational products. The emphasis on relational products is justified by the fact that all relational products support *triggers*. By this term they denote active rules; therefore we will use the terms "trigger" and "active rule" as synonyms. We will conclude this chapter with a comparative review of the various features of active rules and triggers that were progressively introduced.

It is interesting to position the two relational systems relative to the evolution of the SQL3 standard. An effort for defining a standard for SQL triggers has been ongoing since the late 1980s; however, triggers missed the publication deadline for SQL-92, probably because of the inadequacy of the standard document, which was very intricate, especially in a section listing measures for avoiding mutual triggering. Given the importance of standard compliance, all vendors have tried to produce systems as close as possible to the preliminary standard document by disregarding some of its most exotic features, but the document left a number of open issues, resolved by vendors

in different ways. Oracle is a typical example of a system that was carefully developed along the guidelines of the SQL3 standard document dominating between 1990 and 1995.

More recently, in the summer of 1996, the designers of DB2 presented a change proposal for the SQL3 standard that introduces a novel semantics of triggering, in particular with respect to integrity checking. This proposal, which adapts well to the features of most relational products, was recently evaluated and accepted by the SQL3 standard body. Thus, the definition of triggers for DB2 gives us the most recent snapshot on the current SQL3 proposal for triggers, both syntactically and semantically; we cannot easily predict now when and how such a standard proposal will become definitive and be published.

2.1 Starburst

Starburst is the name of a project developed at the IBM Almaden Research Center; an active database extension developed in this framework, called the Starburst Active Rule System, has gained popularity mainly because of its simple syntax and semantics, which adopts the classical set-oriented syntax and semantics of relational databases.

The Active Rule System component extends the data definition language (DDL) of Starburst by enabling the creation and deletion of active rules. Active rules are based on the Event-Condition-Action (ECA) paradigm:

- Events are SQL data manipulation primitives (**INSERT**, **DELETE**, **UPDATE**).
- The condition is a boolean predicate on the database state, expressed in SQL.
- Actions may perform arbitrary SQL queries (including **SELECT**, **INSERT**, **DELETE**, **UPDATE**); in addition, actions may include rule manipulation statements and the transactional instruction **ROLLBACK WORK**.

The ECA paradigm has a simple, intuitive semantics: *when* the event occurs, *if* the condition is satisfied, *then* perform the action. This basic semantics is followed by most active rule systems; we say that a rule is *triggered* when its relevant event occurs, is *considered* when its condition is evaluated, and is *executed* when its action is performed. However, as it will become clear, there are subtle differences in the way in which different

systems define the semantics of triggering, consideration, and execution of active rules.

Active rules are added to the schema and shared by all applications; however, each transaction can dynamically activate and deactivate existing rules. In Starburst, rules can be grouped, and a user can selectively activate and deactivate the rules of a group.

As an initial example, consider the following SalaryControl rule, which monitors the hiring, firing, and change of salary of employees. The rule is defined by a conservative manager who keeps the salary of his employees under control by imposing a salary reduction whenever the average salary goes beyond a given threshold.

Example 2.1 Salary control rule

```
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF     (SELECT AVG (Sal) FROM Emp) > 100
THEN   UPDATE Emp
       SET Sal = .9 * Sal
```

We now proceed with a systematic description of the syntax and semantics of active rules in Starburst.

2.1.1 Syntax of the Starburst CREATE RULE Statement

The main DDL extension for active rules enables the creation of active rules. Its syntax is defined in the following:[1]

```
<Starburst-rule> ::=CREATE RULE <rule-name> ON <table-name>
                    WHEN <triggering-operations>
                    [ IF <SQL-predicate> ]
                    THEN <SQL-statements>
                    [ PRECEDES <rule-names> ]
                    [ FOLLOWS <rule-names> ]
```

[1]Throughout this part of the book we adopt a simple BNF notation where optionality is denoted by square brackets and alternatives by curly brackets; pluralized grammar production names (such as <rule-names>) indicate one or more repetitions of that grammatic construct, separated by commas.

<triggering-operation> ::= **INSERTED** | **DELETED** |
UPDATED [(<column-names>)]

Each active rule in Starburst has a unique name and is associated to a specific table, called the rule's *target*. Each rule monitors multiple events, specified as the rule's triggering operations; the same SQL statement can be monitored by multiple rules. The ordering of rules is regulated by means of a *partial order* between rules; at rule creation time, it is possible to specify that a given rule precedes given rules and follows given rules. The precedence relationships must be acyclic.

2.1.2 Semantics of Active Rules in Starburst

Rules in Starburst are processed in the context of a given transaction; rule processing is implicitly initiated when the transaction issues a **COMMIT WORK** command, thus yielding to a *deferred* execution; in addition, rule processing can be initiated with an explicit **PROCESS RULES** command.

A rule is either *triggered* or *untriggered*; it is untriggered at the transaction's start, and it becomes triggered because of the occurrence of its triggering events (we will define triggering more precisely later). The triggered rules at a given time form the *conflict set*. Starburst's rule processing algorithm consists of the iteration of the following steps:

Algorithm 2.1 *Starburst's rule processing algorithm*

While the conflict set is not empty

1. *Select one rule R from the conflict set among those rules at highest priority; R's state becomes untriggered.*
2. *Evaluate the condition of R.*
3. *If the condition of R is true, then execute the action of R.*

Rule selection at step 1 of the above algorithm produces one of the rules at highest priority (i.e., those rules in the conflict set that are not preceded by any other rule in the set); given that rules are partially ordered by their rule creation statements, there may be many rules that satisfy the above condition. In order to guarantee repeatability of execution (i.e., the same system's behavior on the same database state and input transaction), the system maintains a total order of rules, consistent with the user-specified partial order; rules at the same user-specified priority level are further ordered by the system on the basis of their creation time.

Action execution may cause the triggering of rules, including the retriggering of the rule being executed. Thus, the above algorithm could lead to an infinite execution, caused by rules that cyclicly trigger each other; in this case, we say that the rule execution is *nonterminating*. As we will see, ensuring termination is one of the main problems of active rule design. When instead rule execution terminates with an empty conflict set, we denote the corresponding database state as a *quiescent state*.

The precise notion of rule triggering requires the definition of *state transitions*; these are transformations from a given state S_1 of the database to another state S_2 because of the execution of SQL statements by transactions. State transitions are defined by means of the sets *inserted*, *deleted*, or *updated*; these sets include the tuples that are, respectively, inserted, deleted, or updated by the SQL data manipulation operations that transform S_1 into S_2. From the set *updated* we can extract subsets containing tuples affected by updates on specific attributes, for example, *updated*(Sal).

Rules consider the *net effect* of operations between two well-defined database states; the net effect is obtained by composing multiple operations occurring on the same tuple. Because of the net effect, each tuple affected by a database operation appears at most in one of the above *inserted*, *deleted*, and *updated* sets. For instance, the net effect of the insertion of a tuple t followed by its deletion is null, while the insertion of a tuple t_1 followed by the update of t_1 into t_2 appears as the insertion of the tuple t_2.

We are now in the position of specifying whether a given rule is triggered. Intuitively, a rule R is triggered if any of the sets corresponding to its triggering operations is not empty, relative to a given state transition. At the first consideration of a rule we consider the transition from the initial state S_0 of the transaction to the state S_1 that exists when R is selected by the rule processing algorithm; this is called the *initial transition* for R. After consideration, R is detriggered (regardless of whether the rule is executed or not). In order to determine whether R is triggered in a later state S_2, we consider the transition from S_1 to S_2, and so on, until rule processing terminates on a quiescent state S_f, in which rule R, as well as all other rules, is not triggered.

The use of net effects and the notion of transitions as defined above has the consequence that each rule processes each database change to individual tuples exactly once.

In the condition or action parts of the rule, it is possible to make references to temporary relations storing the instances of the tuples that characterize state transitions. More precisely, four transition tables are defined, respectively called **INSERTED**, **DELETED**, **OLD-UPDATED**, and **NEW-**

UPDATED. The **INSERTED** and **DELETED** transition tables store the tuples belonging to the sets of *inserted* and *deleted* tuples as defined above; **OLD-UPDATED** stores the content of these tuples *before* the update, while **NEW-UPDATED** stores the content of updated tuples *after* the update. Referring to transition tables within SQL statements of active rules may be more efficient than referring to the current content of regular tables because transition tables are usually small and are resident in main memory. Transition tables are not defined outside of the scope of active rules.

2.1.3 Other Active Rule Commands

The DDL extensions provided by the Active Rule Language of Starburst also include commands for dropping, altering, activating, and deactivating rules:

<deactivate-rule> ::= **DEACTIVATE RULE** <rule-name> **ON** <table>
<activate-rule> ::= **ACTIVATE RULE** <rule-name> **ON** <table>
<drop-rule> ::= **DROP RULE** <rule-name> **ON** <table>

Rules in Starburst can be organized within a *ruleset*. Rulesets are created and manipulated by DDL instructions:

<create-ruleset> ::= **CREATE RULESET** <ruleset-name>
<alter-ruleset> ::= **ALTER RULESET** <ruleset-name>
[**ADDRULES** <rule-names>]
[**DELRULES** <rule-names>]
<drop-ruleset> ::= **DROP RULESET** <ruleset-name>

Rule processing can be invoked within a transaction on a specific ruleset, or even on a specific rule, by means of commands:

<rule-processing-commands> ::= **PROCESS RULES**
| **PROCESS RULESET** <ruleset-name>
| **PROCESS RULE** <rule-name>

2.1.4 Examples of Active Rule Executions

Let us consider an active rule system with only one rule, the SalaryControl rule presented before and repeated here for convenience:


```
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF     (SELECT AVG (Sal) FROM Emp) > 100
THEN   UPDATE Emp
       SET Sal = .9 * Sal
```

Consider the following initial database state:

Employee	*Sal*
Stefano	90
Patrick	90
Michael	110

Consider a transaction that performs the inserts of tuples (Rick,150) and (John,120). The insertion triggers the rule, which is selected at step 1 of the rule processing algorithm. The condition holds (the average is now 112), and the action is executed; the new database state is shown below:

Employee	*Sal*
Stefano	81
Patrick	81
Michael	99
Rick	135
John	108

The action again triggers the rule, this time because of an **UPDATE** operation to the Sal attribute; the rule is again considered, the condition holds again (the average is now 101, which is still too high), and the action is again executed, yielding the database state below:

Employee	*Sal*
Stefano	73
Patrick	73
Michael	89
Rick	121
John	97

At this point, the action again triggers the rule, the rule is again considered, but the condition evaluation yields FALSE (the average is 91); thus, rule processing terminates on a quiescent state. Note that termination occurs because the rule's action decreases the average salary and the rule's condition checks that the average salary is below a given threshold; thus,

rule processing *converges* to a quiescent state where the rule's condition is met. If, instead, the multiplicative factor were equal to or greater than 1 in the action, then the execution would have been infinite. Generally, it is the responsibility of the rule designer to ensure that all executions terminate.

Let us consider now the effect of adding to the active database the rule HighPaid, which inserts within a view the tuples of those newly inserted employees whose salary exceeds 100.

Example 2.2 High paid rule

```
CREATE RULE HighPaid ON Emp
WHEN INSERTED
IF      EXISTS (SELECT * FROM INSERTED
                WHERE Sal > 100)
THEN    INSERT INTO HighPaidEmp
        (SELECT * FROM INSERTED
        WHERE Sal > 100)
FOLLOWS AvgSal
```

Consider again the transaction adding tuples for Rick and John to a database storing tuples for Stefano, Patrick, and Michael as illustrated above. Both rules SalaryControl and HighPaid are triggered by the insert, but the latter follows the former in rule ordering. Thus, rule SalaryControl is considered twice by the rule processing algorithm, yielding to the same final state relative to the Emp table. At this point, rule SalaryControl is not triggered, while rule HighPaid is still triggered because of the initial **INSERT** operations. Thus, the rule is considered, and its condition is satisfied. Because of net effects, the rule considers as **INSERTED** the following temporary table:

Employee	*Sal*
Rick	122
John	97

Note that this table is quite different from the value of the tuples that were actually inserted, as it reflects the application of two updates to each of the inserted tuples. The rule's action is executed next, causing the tuple (Rick,122) to be inserted into the view HighPaid. The tuple (John,97) is not inserted, although John was originally inserted by the transaction with a salary that was higher than 100, because this salary was rectified by the execution of the SalaryControl rule. Rule prioritization, in this case, serves the purpose of enabling the consideration of rule HighPaid only after the (recursive) consideration and execution of rule SalaryControl.

2.2 Oracle

Oracle supports general-purpose triggers, developed according to preliminary documents on the SQL3 standard for triggers. Triggers in Oracle may execute actions that contain arbitrary PL/SQL code; this feature makes triggers in Oracle particularly powerful and expressive. Triggers in Oracle respond to data manipulation operations (insert, delete, update) on a given target table. They support two distinct granularities, *row-level* and *statement-level*. With row-level triggers, triggering occurs for each row that is affected by the operation; thus, row-level triggers have an *instance-oriented semantics*, different from the one supported by Starburst. On the other hand, statement-level triggers have a set-oriented semantics, as in Starburst.

Another distinguishing feature of triggers in Oracle is that they are *immediate*: trigger consideration is tightly coupled with the execution of the triggering SQL statement. Recall that, in contrast, Starburst has a *deferred* execution mode, where triggers are executed either because of an explicit transactional statement or at commit time. In particular, triggers in Oracle can be considered and executed either *before* or *after* the triggering operation. Thus, by combining two granularities and two raising times, we get four possible combinations:

1. row-level before-triggers
2. statement-level before-triggers
3. row-level after-triggers
4. statement-level after-triggers

2.2.1 Syntax of the Oracle CREATE TRIGGER Statement

The syntax for creating triggers in Oracle is

```
<Oracle-trigger> ::= CREATE TRIGGER <trigger-name>
                     { BEFORE | AFTER } <trigger-events>
                     ON <table-name>
                     [ [ REFERENCING <references> ]
                       FOR EACH ROW
                       [ WHEN ( <condition> ) ] ] <PL/SQL block>

<trigger event> ::= INSERT | DELETE | UPDATE
                    [ OF <column-names> ]
```

<reference> ::= **OLD AS** <old-value-tuple-name> |
NEW AS <new-value-tuple-name>

Let us discuss each syntactic feature in detail. Each trigger may monitor any combination of the three data manipulation operations (insert / delete / update) on the target table; when the trigger's reaction to events being monitored depends on the event and the trigger is row-level, special predicates **INSERTING**, **DELETING**, and **UPDATING** may be used in the action to detect which of the triggering events has occurred.

The granularity of the trigger is dictated by the clause **FOR EACH ROW**, which needs to be included for tuple-level triggers; when this clause is omitted, the granularity is statement-level. A condition is optionally supported only when the trigger is row-level, and consists of a simple row-level predicate that has to be true in order for the action to be executed on the specific row. For statement-level triggers, the condition part is not present; a condition can nonetheless be expressed within the action as a **WHERE** clause of an SQL statement, but in this case its failure does not prevent the action from being taken, which may result in the endless activation of the trigger.

For both row- and statement-level triggers, the action is an arbitrary block written in PL/SQL, with the limitation that the block should not include DDL or transactional commands.

References to past states are possible only when the trigger is row-based, and they are restricted to the specific tuple being changed; it can be referenced by means of correlation variables introduced by the **REFERENCING** clause, or implicitly by built-in variables called **OLD** and **NEW**.

2.2.2 Semantics of Oracle Triggers

Oracle versions prior to 7.1 impose a severe limitation, namely, that each operation can be monitored by at most one trigger per mode; thus, each operation has at most four associated triggers, corresponding to the options defined above. The limitation on the number of triggers is dropped in Oracle 7.2 and subsequent versions, however without providing an explicit mechanism for prioritizing the triggers that respond to the same event and mode, whose order is system-controlled and cannot be influenced by the database administrator.

The execution of a specific SQL statement is interleaved with the execution of triggers according to the following approach:

Algorithm 2.2 *Oracle's rule processing algorithm*

1. *Execute the statement-level before-triggers.*
2. *For each row in the target table:*
   a. *Execute the row-level before-triggers.*
   b. *Perform the modification of the row and row-level referential integrity and assertion checking.*
   c. *Execute the row-level after-triggers.*
3. *Perform statement-level referential integrity and assertion checking.*
4. *Execute the statement-level after-triggers.*

Each database manipulation performed from within the action part may cause the activation of other triggers; in this case, the execution of the current trigger is suspended and the other triggers are considered by applying to them the execution algorithm illustrated above. The maximum number of *cascading* (i.e., activated) triggers is 32; when the system reaches this threshold, which indicates the possibility of nontermination, execution is suspended and a specific exception is raised. If an exception is raised or an error occurs during trigger execution, then all database changes performed as a result of the original SQL operation and all the subsequent triggered actions are rolled back. Thus, Oracle supports partial (per statement) rollbacks instead of transactional rollbacks.

2.2.3 Example of Trigger Executions

The following trigger Reorder is used to automatically generate a new order (by inserting a tuple into the table PendingOrders) whenever the available quantity (PartOnHand) for a given part in the Inventory table goes below a given threshold (ReorderPoint). The attribute Part is the key of both tables Inventory and PendingOrders.

Example 2.3 Reorder rule

```
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
WHEN (New.PartOnHand < New.ReorderPoint)
FOR EACH ROW
    DECLARE NUMBER X
    BEGIN
        SELECT COUNT(*) INTO X
```

```
      FROM PendingOrders
      WHERE Part = New.Part;
    IF X=0
    THEN
      INSERT INTO PendingOrders
      VALUES (New.Part, New.OrderQuantity, SYSDATE)
    END IF;
  END;
```

The above trigger has a row-level granularity and is considered immediately after each update to the PartOnHand attribute. The condition is checked for each row by taking into account the values that are produced by the update, denoted by means of the New correlation variable. The action contains code in PL/SQL, which however is straightforward; a variable X is declared and then used to store the number of pending orders for the particular part. If that number is equal to zero (i.e., if there are no pending orders for that part), then a new order is issued by inserting into PendingOrders a tuple with the name of the part, the (fixed) part's reorder quantity, and the current date. It is assumed that tuples in the PendingOrders table are deleted by a suitable application when parts are received at the inventory. Note that values of the row that is associated to the trigger's execution can be accessed in the action part of the trigger by means of the New correlation variable.

Consider the following initial state for the Inventory table; assume that PendingOrders is initially empty:

Part	*PartOnHand*	*ReorderPoint*	*ReorderQuantity*
1	200	150	100
2	780	500	200
3	450	400	120

Consider the following transaction (executed on October 10, 1996):

```
T1: UPDATE Inventory
    SET PartOnHand = PartOnHand - 70
    WHERE Part = 1
```

This transaction causes the trigger Reorder to be executed, resulting in the insertion into PendingOrders of the tuple (1,100,1996-10-10). Next, assume that the following transaction is executed on the same day:


```
T2: UPDATE Inventory
    SET PartOnHand = PartOnHand - 60
    WHERE Part >= 1
```

At this point, the trigger is executed upon all the tuples, and the condition holds for parts 1 and 3. However, no new order is issued for part 1 (because of the order pending to this part), while a new order is issued for part 3, resulting in the new tuple (3,120,1996-10-10), which is also inserted into the table PendingOrders. Note that, although T_1 and T_2 are separately committed, the effects of T_1 are seen by T_2, since the two transactions share the tables of the database, and active rules run within the scope of their triggering transactions.

2.3 DB2

Triggers for DB2 Common Servers were recently defined at the IBM Almaden Research Center; efforts were focused on giving to triggers a precise, unambiguous semantics, especially with respect to integrity constraint maintenance, by taking into account the experience gained in the earlier development of the Starburst system.

2.3.1 Syntax of the DB2 CREATE TRIGGER Statement

Every trigger in DB2 monitors a single event, which can be any SQL data manipulation statement; updates can be optionally qualified by a list of column names. As in Oracle, triggers are activated either **BEFORE** or **AFTER** their event, and have either a row- or a statement-level granularity. The syntax of triggers in DB2 is shown below:

```
<DB2-trigger> ::= CREATE TRIGGER <trigger-name>
                  { BEFORE | AFTER } <trigger-event>
                  ON <table-name>
                  [ REFERENCING <references> ]
                  FOR EACH { ROW | STATEMENT }
                  WHEN ( <SQL-condition> )
                  <SQL-procedure-statements>
```

condition is satisfied, and the action yields to the same final database state as in the previous example of execution.

Both the previous examples are event-consuming; we turn now to an event-preserving trigger, which monitors change of salaries and inserts employees whose salary has a rise of 5000 into a class SpecialEmp.

Example 2.8 Special employee rule

```
define immediate event-preserving trigger SpecialEmp for Employee
  events      modify(Salary)
  condition   occurred(modify(Salary),Self)
              Self.Salary - old(Self.Salary) > 5000
  actions     create(specialEmp, [Self.name], Z)
  before      AdjustSalary
end;
```

Consider an active database consisting of triggers AdjustSalary and SpecialEmp and the following transaction:

```
T2: begin transaction;
    select(X where Employee(X), X.Name='john'),
      modify(Employee.Salary,X,X.Salary+3000);
    select(X where Employee(X), X.Name='john'),
      modify(Employee.Salary,X,X.Salary+4500);
    commit;
```

Trigger SpecialEmp becomes triggered after the first modification and is immediately considered, but its condition is false. Next, SpecialEmp is reconsidered after the second modification, which affects the same object o1. This time, the condition is true, since **old** refers to the initial database state, and o1's salary has increased by more than 5000 since the transaction's start. Thus, the object <o5, john> is created in the class SpecialEmp. Finally, on commit, o1's salary is found to be greater than the salary of o1's manager, and thus trigger AdjustSalary sets o1's salary back to 5000.

Consider now what happens if SpecialEmp is turned into an event-consuming trigger. In this case, each event is independently monitored, and thus even at the second consideration the condition of the trigger is false; thus, no object is inserted into the SpecialEmp class. However, if SpecialEmp is turned into an event-consuming, deferred trigger, then it is triggered only once by the transaction, on commit; **old** refers this time to the initial state, and thus the condition is again true.

Consider next the case when both AdjustSalary and SpecialEmp are deferred and assume now that there is no partial order between them; then,

the outcome of the transaction depends on the internal ordering between the two triggers, which is defined by the system. Although the old salary does not change, the current salary is different depending on whether AdjustSalary is considered before or after SpecialEmp, and this changes the truth value of the comparison predicate in the condition of the trigger SpecialEmp. Thus, we cannot determine a priori whether the class SpecialEmp will contain one object or instead be empty. We say that their execution is not *confluent*; confluence will be discussed in Chapter 4. This example shows that imposing a partial order between triggers is essential to control their behavior in case of conflict.

Finally, we show an untargeted trigger monitoring events on more than one class. The trigger forces a rollback if the sum of salaries of employees of the "toys" department exceeds 500,000.

Example 2.9 Excessive salary rule

```
define trigger ExcessiveSalary
  events      create(Employee), modify(Employee.Salary),
              modify(Department.Employees)
  condition   Department(Y), Y.Name='toys', Employee(X),
              X in Y.Employees, sum(X.Salary) > 500000
  action      rollback
end;
```

2.5 Taxonomy of Active Database Concepts

We summarize the main features of active databases that were progressively introduced in the preceding sections.

We start with the fundamental features of active databases: events, conditions, and actions:

- The typical *events* considered by active rules are primitives for database state changes; several systems can also monitor retrievals, some systems monitor time-related events (e.g., at 5 p.m., every Friday), and some systems monitor application-defined, external events explicitly raised by applications.

- The *condition* is either a database predicate or a query. A condition returns a truth value: a query is interpreted as TRUE if it contains some tuples and as FALSE if it is empty.

- The *action* is an arbitrary data manipulation program; it may include transactional commands (such as **ROLLBACK WORK**) or rule manipulation commands (such as the activation or deactivation of active rules or of groups of active rules), and sometimes may activate externally defined procedures.

Some rules are limited to monitoring a single event. However, in most cases, rules monitor a collection of events, with an implicit disjunctive semantics: they become triggered if any one of the monitored events occurs. In some systems, it is possible from within the condition to test on the specific event occurrence that has caused the triggering of the rule. Some systems support rich *event languages*, in which complex events are built from simple events by means of an event calculus supporting conjunction, disjunction, negation, or precedence between events. With complex events, the rule processing system must include a sophisticated mechanism in order to check whether a given rule is triggered at a given point of the transaction's execution.

Consideration of active rules relative to the event being monitored can be immediate, deferred, or detached:

- *Immediate consideration* can occur **BEFORE** the event being monitored, **AFTER** the event, or **INSTEAD OF** the event; in this last option, the event primitive is not executed, and the rule's action is executed instead.
- *Deferred consideration* can occur at the end of a transaction (started by the **COMMIT WORK** transactional command), after user-defined commands, or following a user-defined command starting rule processing (e.g., the **PROCESS RULES** command).
- *Detached consideration* occurs in the context of a separate transaction that is spawned from the initial transaction after the event occurrence. There can be *causal dependencies* between the initial transaction and the detached transaction, for instance, postponing the commit of the detached transaction until after the commit of the initial transaction, or forcing the detached transaction to roll back if the initial transaction rolls back.

Execution of the action relative to the condition's consideration can be immediate, deferred, or detached as well:

- *Immediate execution*, where the action's execution immediately follows the condition's consideration, is used in almost all cases.

- *Deferred execution* postpones the action's execution until the end of the transaction or a user-defined command starting rule processing.
- *Detached execution* occurs in the context of a separate transaction that is spawned from the initial transaction after the rule consideration. In this case, there can be causal dependencies as well between the initial transaction and the detached transaction.

In most cases, events being monitored are changes of the database state; active rules may monitor state changes at different *levels of granularity*:

- State changes can be monitored at the *instance level* (i.e., changes affecting individual rows within tables or individual objects within classes).
- Alternatively, state changes can be monitored at the *statement level*, considering data manipulation statements as events.

Transition values are temporary data describing the state changes performed by a transaction, whose content is affected by the rule's level of granularity:

- When changes are monitored at the instance level, transition values are changes affecting a single tuple or object, typically accessed by means of the **OLD** and **NEW** correlation variables.
- When changes are monitored at the statement level, transition values are collective changes, collected into temporary structures such as the tables **INSERTED** and **DELETED**. Updates can be treated explicitly (e.g., by means of two tables **OLD-UPDATED** and **NEW-UPDATED**) or implicitly (e.g., by adding their before state to the **DELETED** table and their after state to the **INSERTED** table).

In many systems, multiple rules can be triggered at the same time; we say that these rules form a *conflict set*, and the rule processing algorithm must have a given policy for selecting rules out of the conflict set. In active databases, the selection of the next rule to be processed typically occurs after each rule consideration and possible execution. In this way, the next rule to be processed at a generic step of the processing could be triggered by the action of the last rule that was executed. Alternatively, rule selection can be done at given points of the rule processing algorithm, thus building a list of triggered rules that are next considered one after the other, until the list is emptied. This selection alternative is typically used by expert systems in the context of applications of artificial intelligence.

The selection of rules from the conflict set is influenced by *priorities*. The following alternatives are offered:

- Rules can be specified together with a numeric priority that defines their *total ordering*.
- Rules can be specified together with numeric or relative priorities that define a *partial ordering*. In this case, either the system keeps a system-defined total order that is consistent with the user-defined partial order, or else it selects top-level priority rules nondeterministically.
- Finally, rules could have no explicit prioritization mechanism. In this case also, either the system keeps a system-defined total order, or else it selects all rules nondeterministically.

Normally, systems have a system-defined prioritization. In such cases, executions are *repeatable*: two executions of the same transaction on the same database state with the same set of triggers yield the same execution sequence (and the same quiescent state, if it exists). However, some systems indicate their ordering criterion explicitly (i.e., based on rule creation time), while other systems do not indicate their ordering criterion.

In some systems, rules can be dynamically *deactivated* and then *activated*. Rule deactivation, however, is extremely dangerous, since one of the typical applications of active rules is integrity maintenance; in this case, rules preserving database integrity should not be arbitrarily deactivated by any database user. For this reason, active rules (as well as all database objects) are subject to the authorization mechanisms of the database; in particular, privileges of rule creation, altering, deletion, activation, and deactivation can be given only to database administrators, or else can be delegated to other users (e.g., through explicit **GRANT PRIVILEGE** commands).

Rules can be clustered into *groups*. Once a group is formed, it can be separately processed, activated, and deactivated; groups are also useful devices for assigning creation or execution privileges to users in a systematic way.

2.6 Bibliographic Notes

Reference [455] provides a complete and up-to-date treatment of active rules; the book includes chapters dedicated to the systems Postgres, Ariel, Starburst, A-RDL, Chimera, Hipac, and Ode, and then chapters on the description of general active rule features (including their implementation), on commercial systems, and on applications.

The four systems selected in this chapter for illustrating the syntax and semantics of active rules are described in the following research papers and manuals:

- The Starburst language is described in [457]. The implementation techniques used in the Starburst prototype are described in [456]. Priority management is discussed in [5].
- Oracle is defined in the manuals [320] and [319], while the problem of mutating tables is described in [327].
- Triggers for DB2 are illustrated in [114] and in the manual [220].
- Chimera is described in [94] and [95].

Among other relational systems supporting active rules, we mention Informix [222], Sybase [426], and Illustra [221]; among object-oriented research prototypes, we mention Reach [72], Ode [185], and NAOS [119].

Overviews of the various features of active databases, with a description of the concepts and terminology, are given in [332] (where various dimensions are given for an understanding of active behavior) and in [131] (where active database features are classified as either essential or optional, and an applicative classification of active database systems is proposed).

2.7 Exercises

2.1. Given the relational database schema

Employee(<u>Name</u>, Salary, Department)
Department(<u>Dept-No</u>, Manager)

define the following active rules in Starburst, Oracle, and DB2:

a. A rule that, whenever a department is deleted from the database, sets to null the value of the Department attribute for those tuples in relation Employee having the number of the deleted department.

b. A rule that, whenever a department is deleted from the database, deletes all employees in the deleted department.

c. A rule that, whenever the salary of an employee exceeds the salary of its manager, sets the salary of the employee to the salary of the manager.

d. A rule that, whenever salaries are updated, if the total of the updated salaries exceeds their total before the updates, then gives all the employees of the 'Research' department a 5% salary cut.

Complete this exercise by writing the same triggers in Chimera on the following object-oriented schema (which is equivalent to the previous relational schema):

```
create object class Employee
attributes Name: string,
           Salary: integer,
           Department: Dept
end;

create object class Dept
attributes Manager: Employee
end;
```

2.2. Referring to the relational schema above, define in Starburst or Chimera a deferred trigger R_1 that, whenever an employee who is a manager is deleted, also deletes all employees in the department managed by the deleted employee, along with the department itself.

Then define another deferred trigger R_2 that, whenever salaries are updated, checks the average of the updated salaries; if it exceeds 50,000, then it deletes all employees whose salary was updated and now exceeds 80,000.

Consider next a database state containing six employees: Jane, Mary, Bill, Jim, Sam, and Sue, with the following management structure:

- Jane manages Mary and Jim.
- Mary manages Bill.
- Jim manages Sam and Sue.

Now suppose that a user transaction deletes employee Jane and updates salaries in a way such that the average updated salary exceeds 50,000 and Mary's updated salary exceeds 80,000. Describe the trigger processing started at the end of this transaction.

2.3. Given the Chimera class Employee, with an attribute Salary and a class RichEmployee with the same schema, define in Chimera a set of triggers ensuring that in any database state the set of instances of the class

RichEmployee coincides with the set of instances of the class Employee whose value for attribute Salary is greater than 50,000.

ing a message, without changing the database content. Business rules are most advantageous when they support policies that are shared by all applications, so that these policies are centralized once and for all. *Knowledge independence* is the extraction of knowledge from applications and the encoding of it in rules; it will be further discussed in Section 4.4.

The examples in this chapter are written either in Starburst or in Chimera and use set-oriented triggers.

3.1 Integrity Management

Integrity maintenance is the most significant internal application. Constraints are expressed declaratively through predicates, also called *integrity rules*, that give a specification of the conditions that must hold on the database. In particular, constraints may be classified as static or dynamic:

- *Static constraints* are predicates evaluated on database states.
- *Dynamic constraints* are predicates on state transitions, which compare two subsequent database states produced by a transaction.

Integrity constraints are further classified as built-in or generic:

- *Built-in constraints* are fixed; they are specified by special language constructs of the data definition language. In relational databases, these include keys, unique attributes, notnull attributes, and referential integrity, all of which are allowed, for example, in SQL-92.
- *Generic constraints* are arbitrary, specified by generic predicates or queries. Examples are the check clauses and the assertions supported by the SQL-92 standard.

Integrity is enforced by the system, either during transaction execution (immediate constraints) or at the commit time of the transaction (deferred constraints). Many systems, however, check the constraints with internally generated triggers. For instance, Oracle uses this means to check referential integrity constraints. Moreover, many systems support only limited forms of generic constraints, as they do not support assertions and provide only check clauses whose scope is limited to a single row. In this case, the only possibility for specifying generic integrity constraints consists in encoding them by means of active rules.

3.1.1 Rule Generation

Rules for integrity constraints can be generated in a semiautomatic manner. The designer is initially concerned with the understanding of the declarative formula of the constraint, which corresponds to the *condition* part of the rule. Next, he must concentrate on the causes that may violate the constraint; these contribute to the *event* part of the rule and can be syntactically determined by analyzing the condition. Finally, he is concerned about the actions that ensure the constraint is no longer violated, which contribute to the *action* part of the rule. This last step normally requires a good understanding of the application, since repair actions encompass a critical part of the application's semantics; however, a standard repair action can be automatically generated that consists of forcing a transactional or statement-level rollback.

Integrity constraints can be managed by active rules with two different approaches:

- At the simplest level, constraints can be encoded as simple triggers, called *abort rules*. They are activated by any event that can violate the constraint. Their precondition tests for the actual occurrence of violations; if a violation is detected, then the rule issues a rollback command. Thus, abort rules can be syntactically generated from declarative constraints and faithfully implement the behavior of declarative constraints. The disadvantage of this approach is that it causes frequent transactional or statement-level rollbacks.
- At the most sophisticated level, constraints are specified together with a maintenance policy that indicates an action to be performed on the database state when the constraint is violated, so that database integrity is restored. Typical examples are the referential integrity constraints of SQL-92, which are specified together with a repair action. In this case, constraints can be encoded as *repair rules*; these have the same precondition and triggering events as abort rules, but their action contains database manipulations for repairing constraint violations. For instance, repair actions for referential integrity reflect the semantics of the SQL-92 clauses **CASCADE**, **RESTRICT**, **SET NULL**, or **SET DEFAULT**, which represent four policies for correcting violations to referential integrity.

3.1.2 Example

Let us consider a simple referential integrity constraint expressed in Starburst. Consider the "classical" tables Emp (employee) and Dept (department) with an attribute DeptNo of Emp, which expresses a reference to the attribute Dno of Dept; the referential integrity constraint states that all employees must be assigned to a department. The operations that can violate the constraint are

1. insertions into Emp
2. deletions from Dept
3. updates to Emp.DeptNo
4. updates to Dept.Dno

The condition is expressed as a predicate on tuples of the Emp table, stating that each tuple must be associated with a department:

```
Emp: EXISTS (SELECT * FROM Dept WHERE Dno = Emp.DeptNo)
```

Note that this predicate indicates a property that must be TRUE for all legal employees; however, in an active rule we are interested in determining those employees who violate the constraint; for them, the predicate must be FALSE. Thus, we really consider the following condition, obtained by negating the previous formula:

```
Emp: NOT EXISTS (SELECT * FROM Dept WHERE Dno = Emp.DeptNo)
```

The above formula is called the *denial form* of the integrity constraint.

In order to be compliant with the Starburst syntax, where all rules need to be targeted to one specific table, we need to introduce two rules, one for Emp and one for Dept; each rule monitors the events relevant to its target table. Thus, we can automatically produce the following abort rules:

Example 3.1 Abort rules

```
CREATE RULE DeptEmp1 ON Emp
WHEN INSERTED, UPDATED(DeptNo)
IF      EXISTS (SELECT * FROM Emp WHERE NOT EXISTS
                (SELECT * FROM Dept
                WHERE Dno = Emp.DeptNo))
```



```
CREATE RULE DeptEmp2 ON Dept
WHEN DELETED, UPDATED(Dno)
IF      EXISTS (SELECT * FROM Emp
WHERE NOT EXISTS
                (SELECT * FROM Dept
                WHERE Dno = Emp.DeptNo))
THEN  ROLLBACK
```

Note that these rules are not efficient. Whenever an event occurs that could violate the constraint, they compute the condition relative to the entire database, without taking into account transition values. Instead, rules could be coded by assuming that the constraint holds on the initial state; in that case, it is sufficient to compute the condition relative to transition values.

Next we write four optimized maintenance rules for the same example, which use transition values for achieving efficiency in the computation. Events and conditions do not change, but the action is generated according to the referential integrity policy. Note that, in order to implement these policies, each of the events producing a violation is associated with a different active rule.

We start with the two rules targeted on Emp; we choose to set the department number of new employees to **NULL**, and to set the department number of employees who violate the constraint because of a change of department number to a default value of 99. Note that strict referential integrity as in SQL-92 forces the partial rollback of these operations, but the trigger designer can override this policy.

Example 3.2 Repair rules on table Emp

```
CREATE RULE DeptEmp1 ON Emp
WHEN INSERTED
IF      EXISTS ( SELECT * FROM INSERTED
                 WHERE NOT EXISTS
                        (SELECT * FROM Dept
                         WHERE Dno = Emp.DeptNo))
THEN  UPDATE Emp
      SET DeptNo = NULL
      WHERE EmpNo IN
                (SELECT EmpNo FROM INSERTED)
      AND NOT EXISTS
                (SELECT * FROM Dept
                 WHERE Dno = Emp.DeptNo))
```

```
CREATE RULE DeptEmp2 ON Emp
WHEN UPDATED(DeptNo)
IF      EXISTS (SELECT * FROM NEW-UPDATED
                WHERE NOT EXISTS
                 (SELECT * FROM Dept
                        WHERE Dno = Emp.DeptNo))
THEN  UPDATE Emp
      SET DeptNo = 99
      WHERE EmpNo IN
                (SELECT EmpNo FROM NEW-UPDATED)
      AND NOT EXISTS
                (SELECT * FROM Dept
                 WHERE Dno = Emp.DeptNo))
```

For the rules targeted on the Dept table, we opt for a *cascade delete* policy, which consists in deleting the employees who remain "orphans" after a deletion of their department. The condition is optimized by introducing a test on the existence of deleted or updated departments, under the assumption that attribute DeptNo is a key of table Dept; thus, if a given department number is deleted or updated, employees with that department number certainly remain orphans, and therefore they must be deleted as well.

Example 3.3 Repair rules on table Dept

```
CREATE RULE DeptEmp3 ON Dept
WHEN DELETED
IF      EXISTS (SELECT * FROM Emp WHERE EXISTS
                (SELECT * FROM DELETED
                 WHERE Dno = Emp.DeptNo))
THEN  DELETE FROM Emp
      WHERE EXISTS
                (SELECT * FROM DELETED
                WHERE Dno = Emp.DeptNo)

CREATE RULE DeptEmp4 ON Dept
WHEN UPDATED(Dno)
IF      EXISTS (SELECT * FROM Emp WHERE EXISTS
                (SELECT * FROM OLD-UPDATED
                 WHERE Dno = Emp.DeptNo))
THEN  DELETE FROM Emp
      WHERE EXISTS
```



```
                (SELECT * FROM OLD-UPDATED
                WHERE Dno = Emp.DeptNo)
```

3.2 Derived Data Maintenance

Another important internal application of rules is view and derived data maintenance. A *view* is a query on the database; the query returns a table (in the relational model) or class (in the object-oriented model) whose content is derived from the content of the database; views, however, can be transparently used by applications as any other relation or class of the database. Similarly, a *derived attribute* is an attribute of a relation or class whose content can be derived by means of a derivation formula, which can be transparently used by applications as any other attribute of the database. Data derivations for views or derived attributes can be expressed by declarative query languages such as SQL; alternatively, they can be expressed by *deductive rules* (i.e., declarative rules expressing how derived data should be deduced from source data). Deductive rules enhance the power of SQL by adding various features, the most important of which is *recursion*, extensively described in Part III.

In order to support views and derived data, a database management system can use two strategies:

- Derived data can be *virtually supported*; in this case, their content is computed on demand, whenever the application issues a query that uses either a view or a derived attribute.
- Alternatively, derived data can be *materialized*; their content is persistently stored in the database. Materialized data can be accessed as any other data, but their content must be recomputed whenever the source data from which they are derived is changed.

3.2.1 Rule Generation

Active rules can be used both for data derivation and for data materialization; virtual data derivation by active rules is, however, currently supported only for views in the Illustra database system. Illustra supports two features that are very useful for this purpose: the ability of rules to be triggered by **SELECT** queries, and the clause **INSTEAD OF** for indicating that the action should be executed instead of the triggering event. Thus, whenever an application queries a view, a rule is triggered on the **SELECT** and provides, instead of access to the view, access to the underlying source tables.

In general, however, active rules are most used for the materialization of derived data. Data materialization is either through refresh or through incremental maintenance.

- The *refresh* approach consists in recomputing from scratch derived data from the source tables after each update of the source data.
- The *incremental* approach consists in computing the change to the view from the change to the source table. This approach requires computing the positive and negative *deltas*, consisting of the tuples that should be, respectively, inserted into or deleted from the view.

Rule generation for the materialization of derived data can be fully automatic: once the derivation rule is specified and the causes that make the rule invalid are syntactically determined, actions that make the view up-to-date can also be generated automatically. Refresh rules are very simple, while incremental rules are more complex and must be generated specifically for given structures of derivation rules; difficulties arise mainly from recursion and from the existence of duplicates and/or of multiple derivations for the same tuple of a view. A theory of incremental rules for derived data maintenance is outside the scope of this book.

These techniques give us devices for transforming deductive rules expressing data derivations into active rules; such transformation is particularly useful when deductive rules are not directly supported by a target database system.

3.2.2 Example

Let us consider the relational view selecting those departments that have at least one "rich" employee (who earns more than 50,000):

```
DEFINE VIEW HighPaidDept AS
        (SELECT DISTINCT Dept.Name
         FROM Dept, Emp
         WHERE Dept.Dno = Emp.DeptNo
         AND Emp.Sal > 50K)
```

The events that may cause the recomputation of the view are

1. insertions into Emp
2. insertions into Dept
3. deletions from Emp
4. deletions from Dept
5. updates to Emp.DeptNo

6. updates to Emp.Sal
7. updates to Dept.Dno

The refresh rules for this view, written in Starburst, are the following:

Example 3.4 Refresh rules

```
CREATE RULE RefreshHighPaidDept1 ON Dept
WHEN INSERTED, DELETED,
        UPDATED(DeptNo), UPDATED(Sal)
THEN  DELETE * FROM HighPaidDept;
        INSERT INTO HighPaidDept:
          (SELECT DISTINCT Dept.Name
           FROM Dept, Emp
           WHERE Dept.Dno = Emp.DeptNo
           AND Emp.Sal > 50K)

CREATE RULE RefreshHighPaidDept2 ON Emp
WHEN INSERTED, DELETED, UPDATED(Dno)
THEN  DELETE * FROM HighPaidDept;
        INSERT INTO HighPaidDept:
          (SELECT DISTINCT Dept.Name
           FROM Dept, Emp
           WHERE Dept.Dno = Emp.DeptNo
           AND Emp.Sal > 50K)
```

An incremental rule is easily written for the **INSERT** event on Dept, shown below, but it is more complex for other events.

Example 3.5 Incremental refresh rule

```
CREATE RULE IncrRefreshHighPaidDept1 ON Dept
WHEN INSERTED
THEN  INSERT INTO HighPaidDept:
        (SELECT DISTINCT Dept.Name
         FROM INSERTED, Emp
         WHERE INSERTED.Dno = Emp.DeptNo
         AND Emp.Sal > 50K)
```

3.3 Replication

Replication is a particular form of data derivation between several copies of the same information, which must be kept aligned. In general, replication

is used in distributed systems: two or more copies of the same information are kept on different database servers. In a distributed system, keeping copies synchronized at all times is too expensive, and typically not required from an applicative viewpoint, since copies do not need to be identical at all times. Therefore, asynchronous techniques are used for propagating changes from one copy to another, rather than synchronizing them in the context of distributed transactions. Two replication schemes are used:

- The most widely used approach to replication distinguishes between a *primary copy* and one or more *secondary copies*. All changes are performed on the primary copy, while the secondary copy is asynchronously updated by the replication manager and is read-only for all applications. Changes to the primary copy are first monitored by a *capture module* installed at the primary copy site and then propagated to the copy by an *apply module* installed on the sites storing the secondary copies. Typically, the change capture occurs in the context of applications that modify the primary copies, while the change application is performed asynchronously.

- A more recent approach is *symmetric replication*, in which two copies can asynchronously accept changes. Thus, each of them in turn plays the role of both primary and secondary copy, and each has both a capture and an apply module. Concurrent updates can be performed on the copies without requiring their synchronization; obviously, this can cause losses of consistency between the copies, which must be recognized and reported to applications.

Active rules provide an efficient mechanism for implementing the capture module; they run in the context of transactions that perform the changes to source data, and construct the positive and negative deltas (i.e., the tuples that should be added to or dropped from the secondary copy in order to make it consistent with the primary). Deltas are next applied to secondary copies. The following active rules, written in Starburst, perform the capture function:

Example 3.6 Capture rules

```
CREATE RULE Capture1 ON Primary
WHEN INSERTED
THEN  INSERT INTO PosDelta
          (SELECT * FROM INSERTED)
```



```
CREATE RULE Capture2 ON Primary
WHEN DELETED
THEN  INSERT INTO NegDelta
          (SELECT * FROM DELETED)

CREATE RULE Capture3 ON Primary
WHEN UPDATED
THEN INSERT INTO PosDelta
          (SELECT * FROM NEW-UPDATED);
         INSERT INTO NegDelta
          (SELECT * FROM OLD-UPDATED)
```

3.4 Workflow Management

Workflow managers are emerging software systems for organizing the working activities within enterprises. They support a description of the process, and they help its execution and management by monitoring the assignment of tasks to users and by assisting their coordination through document and message exchange.

The workflow management paradigm is a reactive one: workflow managers monitor the events that occur in the system and then perform the required event management activities. Events can be *internal* (i.e., generated from within the workflow manager while workflows are progressing) or *external* (i.e., representing the interaction of the workflow manager with the external world). In particular, events may represent exceptions to the normal execution of workflows. For the features described above, workflow management is a very promising field of application for active databases.

An example of an internal event representing an exceptional situation is the sudden unavailability of an agent who is performing an urgent task; in such a case, the task is assigned to the agent's substitute, provided that he is in turn available. With suitable Chimera classes describing agents and tasks, the following trigger performs the required exception management:

```
define trigger WF1 for Agent
  events      modify(Agent.Availability)
  condition   Agent(A), occurred(modify(Agent.Availability),A),
              A.Availability=FALSE, task(T), T.Responsible=A,
              T.Type='Urgent', Agent(B), A.Substitute=B,
              B.Availability=TRUE
```

```
  actions     modify(Task.Responsible, T, B)
end;
```

An exceptional situation in a car rental workflow occurs when one of the company's cars is damaged in an accident and that car is booked for a car rental. In such a case, the workflow system creates a warning message that includes the booking number and responsible agent; based on that message, the agent will find a substitute for the car or inform the client. The following trigger performs the required exception management:

```
define trigger WF2 for Accident
  events      create(Accident)
  condition   Accident(A), occurred(create, A),
              Booking(B), B.Car=A.DamagedCar,
  actions     create(Warning,[B.Number,B.Agent],X)
end;
```

3.5 Business Rules

Business rules model the reaction to events that occur in the real world, with tangible side effects on the database content, so as to encapsulate the application's reactive behavior to such events. Their organization and content are not regular and cannot be anticipated or declaratively specified. Representative examples of business rules, from real-life applications, include rules for stock and bond trading in financial applications, for airway assignment to flights in air traffic control systems, for order management in inventory control systems, and so on. The common feature of these rules is to respond to external events by pursuing certain objectives: profit maximization, flight safety, optimal warehouse logistics, and so on.

Generation of business rules is a critical process; the designer should be concerned with the portion of application-specific reactive processing, modeled by active rules rather than supported by the applications. We will come back to the design of business rules in the next chapter; in particular, Section 4.4 presents a methodology for the generation of business rules.

3.5.1 A Case Study: Energy Management System (EMS)

We conclude this chapter with an example of an application that is modeled with active rules covering the business process. We will look at a fragment of the Energy Management System (EMS), a system managing the Italian electrical power distribution network. The EMS network is composed

of *sites* and *branches* connecting site pairs. Sites can be power stations, where power is generated; intermediate nodes, used for power distribution throughout the network; or final users, which make use of electrical power. The operating conditions of the network are monitored daily, with frequent reconfigurations; the topology is modified only once per month. An "operational network" has a radial topology (a forest of trees connecting power stations to users), although many other branches exist that can be dynamically added to or dropped from the network. The purpose of the network is to transfer the exact power from distributors to users through nodes and (directed) branches connecting node pairs. To this purpose, several wires are placed along the branches; wires are of given types, and each type carries a maximum power. Active rules are used to respond to input transactions asking for reconfigurations due to new users, or to changes in their required power, or to changes in the assignments of wires and tubes; note that in this example the same rules serve several different input transactions.

3.5.2 Database Schema for the EMS Case Study

We give a flat representation of the schema, without using generalization hierarchies. Nodes of the network are described by the classes Distributor, User, and Node. The class Distributor describes electrical power generators that provide power to the network, while the class User describes consumers of the network power. The class Node describes intermediate sites, where power is transferred in order to be distributed.

```
create object class IntNode
attributes NodeId: Site,
           BranchIn: Branch,
           Power: real
end;

create object class User
attributes UserId: Site,
           BranchIn: Branch,
           Power: real
end;

create object class Distributor
attributes NodeId: Site,
           Power: real,
           MaxPower: real
end;
```

The topology of the network is described in the class Branch, which contains all the connections between site pairs. The connections are described in the class Wire and can have a type described in the class WireType.

```
create object class Branch
attributes From: Site,
           To: Site,
           Power: real
end;

create object class Wire
attributes From: Site,
           To: Site,
           Type: WireType,
           Power: real
end;

Create object type WireType
attributes MaxPower: real
end;
```

3.5.3 Business Rules for the EMS Case Study

The application gathers several users' requests and then applies all of them within a transaction; business rules propagate changes in the users' requirements up in the trees connecting users to distributors. The transaction fails if the maximum power of some distributors is exceeded because this situation forces the network's redesign, which is done manually. Otherwise, rules adapt the network to the new needs. In order to avoid an early propagation of increased requests that may lead to unnecessary rollbacks, rules propagate the reductions of power first and the increases of power next. If the requested power on wires exceeds the maximum power that can be carried by them, then rules automatically change or add wires in the relevant branches. These specifications yield to the following 11 rules, described first in natural language (for ease of understanding) and next in Chimera. Priorities require that rules R_2, R_3, and R_4 precede rules R_5, R_6, and R_7.

R_1: *If a new user node requires power, connect it to the closest node.*
R_2: *If a user or a node requires less power, change the power of the user or node and propagate the change to its input branch.*

R_3: *If a branch requires less power, change the power of the branch and propagate the change to its input node.*
R_4: *If the power required from a distributor is decreased, change its output power accordingly.*
R_5: *If a user or node requires more power, change the power of the user or node and propagate the change to its input branch.*
R_6: *If a branch requires more power, change the power of the branch and propagate the change to its input node.*
R_7: *If the power required from a distributor is increased, change its output power accordingly.*
R_8: *If a distributor's power exceeds its maximum, rollback the entire transaction.*
R_9: *If a branch's power is changed, change the power of some of its wires accordingly.*
R_{10}: *If a wire's power is above its threshold, change wire type.*
R_{11}: *If there's no suitable wire type, add another wire to the branch.*

We next show these rules written in Chimera. This is a nontrivial rule set that serves the purpose of showing a complex application of active rules; it can be skipped by the uninterested reader. Several user-defined functions and procedures are not further defined: function closest(U,N) finds the node N closest to a given user U; function one(set) returns an element randomly selected among the set of elements passed as arguments. Rules R_2 and R_5 have two versions, each targeted on a different class.

Example 3.7 Business rules for the EMS case study

```
define trigger R1 for User
  events      create
  condition   User(U), occurred(create,U),
              Node(N), closest(U,N),
  actions     create(Branch,[U.UserId,N.NodeId,U.Power],X),
              create(Wire,[U.UserId,N.UserId,'default',U.Power],Y)
end;
```

Trigger R1 reacts to the creation of a new user; it determines the node N in the network that is closest to the new user and connects the new user to N with a branch and a wire.

```
define trigger R2.1 for User
  events      modify(Power)
  condition   User(U), occurred(modify(Power),U),
              real(R), R=old(U.Power)-U.Power, R > 0,
              Branch(B), U.BranchIn=B
  actions     modify(Branch.Power,B,B.Power-R)
before R5.1,R5.2,R6,R7
end;
```

Trigger R2.1 reacts to a decrease in the power requested by a user; it selects the branch incoming to the user and propagates the power decrease to the branch.

```
define trigger R2.2 for IntNode
  events      modify(Power)
  condition   IntNode(N), occurred(modify(Power),N),
              real(R), R=old(N.Power)-N.Power, R > 0,
              Branch(B), N.BranchIn=B
  actions     modify(Branch.Power,B,B.Power-R)
before R5.1,R5.2,R6,R7
end;
```

Trigger R2.2 reacts to a decrease in power requested by a node; it selects the branch incoming to the node and propagates the power decrease to the branch.

```
define trigger R3 for Branch
  events      modify(Power)
  condition   Branch(B), occurred(modify(Power),B),
              real(R), R=old(B.Power)-B.Power, R > 0,
              IntNode(N), N.From=B
  actions     modify(IntNode.Power,N,N.Power-R)
before R5.1,R5.2,R6,R7
end;
```

Trigger R3 reacts to a decrease in power transmitted along a branch; it selects the source node of the branch and propagates the power decrease to the node.


```
define trigger R4 for Branch
  events      modify(Power)
  condition   Branch(B), occurred(modify(Power),B),
              real(R), R=old(B.Power)-B.Power, R > 0,
              Distributor(D), D.From=B
  actions     modify(Distributor.Power,D,D.Power-R)
before R5.1,R5.2,R6,R7
end;
```

Trigger R4 reacts to a decrease in power transmitted along a branch directly connected to a distributor; it selects the distributor and propagates the power decrease to it.

```
define trigger R5.1 for User
  events      create, modify(Power)
  condition   User(U), occurred(modify(Power),U),
              real(R), R=old(U.Power)-U.Power, R < 0,
              Branch(B), U.BranchIn=B
  actions     modify(Branch.Power,B,B.Power-R)
before R5.1,R5.2,R6,R7
end;
```

Trigger R5.1 reacts to the creation of a new user or to the increase in power requested by a user; it behaves like trigger R2.1.

```
define trigger R5.2 for IntNode
  events      modify(Power)
  condition   IntNode(N), occurred(modify(Power),N),
              real(R), R=old(N.Power)-N.Power, R < 0,
              Branch(B), N.BranchIn=B
  actions     modify(Branch.Power,B,B.Power-R)
end;
```

Trigger R5.2 reacts to a decrease in power requested at a node; it behaves like trigger R2.2.

```
define trigger R6 for Branch
  events      modify(Power)
  condition   Branch(B), occurred(modify(Power),B),
              real(R), R=old(B.Power)-B.Power, R < 0,
              IntNode(N), N.From=B
```

```
  actions     modify(IntNode.Power,N,N.Power-R)
end;
```

Trigger R6 reacts to an increase in power transmitted along a branch; it behaves like trigger R3.

```
define trigger R7 for Branch
  events      modify(Power)
  condition   Branch(B), occurred(modify(Power),B),
              real(R), R=old(B.Power)-B.Power, R < 0,
              Distributor(D), D.From=B
  actions     modify(Distributor.Power,D,D.Power-R)
end;
```

Trigger R7 reacts to a decrease in power transmitted along a branch directly connected to a distributor; it behaves like trigger R4.

```
define trigger R8 for Distributor
  events      modify(Power)
  condition   Distributor(D), occurred(modify(Power),D),
              D.Power > D.MaxPower
  actions     rollback
end;
```

Trigger R8 reacts to a request for an increase in power at a distributor that exceeds the distributor's maximum power; it rolls the transaction back.

```
define trigger R9 for Branch
  events      modify(Power)
  condition   Branch(B), occurred(modify(Power),B),
              real(R), R=old(B.Power)-B.Power,
              Wire(W), W.From=B.From, W.To=B.To,
              integer(C), C=count(W)
  actions     modify(Wire.Power,W,W.Power-(R/C))
end;
```

Trigger R9 reacts to the change in power transferred along a given branch by propagating the change to the wires placed along the branch; the change is equally distributed to all the wires of the branch.

```
define trigger R10 for Wire
  events      create, modify(Power), modify(Type)
```



```
  condition   Wire(W), occurred(create, modify(Power),
                                modify(Type), W),
              W.Power > W.Type.MaxPower, WireType(WT1),
              WT1.MaxPower = min(R where real(R), WireType(WT2),
                                 R = WT2.MaxPower, R > W.Power),
  actions     modify(Wire.Type,W,one(WT1))
end;
```

Trigger R10 reacts to the creation of wires or to the change in power carried by a wire when the power exceeds the maximum for its wire type and a wire type exists that can support the new power request. In that case, one suitable wire type is selected, and then the wire type of the selected wire is modified.

```
define trigger R11 for Wire
  events      create, modify(Power), modify(Type)
  condition   Wire(W), occurred(create, modify(Power),
                                modify(Type), W),
              W.Power > max(R where real(R), WireType(WT),
                            R=WT.MaxPower)
  actions     modify(Wire.Power,W,0.8×W.Type.MaxPower),
              create(Wire,[W.From, W.To, W.Type,
                   (W.Power-0.8×W.Type.MaxPower)],X)
end;
```

Trigger R11 reacts to the change in power carried by a given wire when the power exceeds the maximum for its wire type and no wire type exists that can support the new power request. In that case, that wire is assigned to 80 percent of its maximum power, and the remaining power is carried by another wire that is created and placed on the given branch. This change triggers R10, which checks that the power carried by the newly introduced wire does not exceed its maximum power.

3.6 Bibliographic Notes

The "declarative approach" to active rule generation, which consists in giving a declarative specification of active rules and then automatically or semi-automatically generating them by appropriate translation, is described in [96] (relative to integrity constraint management) and [97] (relative to view maintenance). Incremental view maintenance is discussed in [201] and [98]; problems due to the existence of duplicates in views are focused on in [200]

and [195]. A technique for the automatic generation of repair actions for integrity constraints, which takes into account the user's informal indications on the semantics of desired repair rules, is discussed in [93].

The use of active rules for workflow enactment is suggested in [125] and carefully specified in [86]; this paper is developed in the context of the WIDE Esprit project, whose aim is to use active rules and enhanced transactions in support of workflow management systems. The use of active rules for implementing active agents is discussed in [188], which uses workflow management as an example of application of the active agent paradigm.

The EMS application, summarized in this chapter, is further described in [292] and [36]. A significant application of active rules to building design is presented in [421]; the peculiarity of that approach is that rulesets can be dynamically activated and deactivated in order to adapt to the progression of the design.

3.7 Exercises

3.1. Given the relational database schema

Ph.D. Student(Email, Name, Area, Supervisor)
Prof(Email, Name, Area)
Course(Title, Prof)
CoursesTaken(PhDSt, Course)

derive the triggers for maintaining the following integrity constraints:

a. Each Ph.D. student must work in the same area as their supervisor.
b. Each Ph.D. student must take at least one course.
c. Each Ph.D. student must take the courses taught by their supervisor.

3.2. Given the relational database schema

Employee(Name, DeptCode)
Department(DeptCode, DeptName, City, Budget)

and the view MilanEmp, defined as

```
SELECT Name
FROM Employee, Department
WHERE Employee.DeptCode = Department.DeptCode
    AND Department.City = 'Milano'
```


a. define the triggers for incrementally maintaining the view
b. define the triggers for handling updates (insertions and deletions) through the view

3.3. Consider the set of rules specified in Section 3.5.3. Draw a simple energy management network consisting of a few nodes and connections, then populate the corresponding classes in Chimera. Think of simple update operations and generate the corresponding execution trace, listing triggers that are subsequently considered and executed; show the final quiescent state obtained at the end of rule processing. Think of one update operation that causes the execution of the abort trigger R8.