Chapter 1

Specifying Database Transactions and Active Rules in the Situation Calculus

LEOPOLDO BERTOSSI
JAVIER PINTO
RICARDO VALDIVIA[1]

Departamento de Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica de Chile
Santiago 6904411, CHILE.

Abstract

This chapter provides a predicate logic based semantics for active rules in active databases [18, 9]. Our main contribution is a proposal to integrate the specification of active rules with the specification of the dynamics of transaction based change in relational databases. Our approach extends previous work in which the situation calculus, a language of many sorted predicate logic, is used to specify updates in databases [14]. To achieve this, we first specify database transactions in the situation calculus, and then active rules using the notion of occurrence as proposed in [13].

1.1 Introduction

This chapter provides a predicate logic based semantics for active rules in active databases [18, 9]. Our work is based on a dialect of the situation calculus (*Sitcalc*), a language

[1] {bertossi,jpinto,revaldiv}@ing.puc.cl

of many sorted predicate logic that was originally proposed to represent knowledge about dynamic domains [12]. An important characteristic of *Sitcalc* is its treatment of actions, primitive transactions in a database setting[2], at the object level. Furthermore, the language has sorts for domain individuals and situations. In [14], Reiter proposed a logical framework based on the *Sitcalc* to specify transaction based database updates. The only transactions involved in the original proposal are primitive in the sense that they are not decomposable. Logical specifications, in Reiter's style, include axioms to specify preconditions for actions and axioms to specify the direct effects of actions. A completeness assumption is used in order to solve the *frame problem*, allowing to succinctly represent the changes and non-changes produced by the execution of actions. With this kind of specification all possible and legal courses of evolution of the database are open to hypothetical reasoning; therefore, virtual updates can be easily considered.

In database management systems (DBMSs), changes in the state of the database result from the execution of complex transactions, which are composed of sequences of update operations. Thus, the database changes correspond to the net effects of all the operations involved in the transaction. Once the sequence of updates is successfully executed, they are committed, with a $Commit$ operation. That is, the changes on the database are made visible to other transactions.

Transactions can be used as mechanisms to support recovery from failures, concurrency management and consistency maintenance. For instance, to implement consistency maintenance, one can verify the satisfaction of the integrity constraints before committing the transaction. If the new database state is consistent, the $Commit$ action is triggered, otherwise the transaction is aborted by triggering a $Rollback$ operation. In this chapter, we extend Reiter's work on the formalization of database updates by formalizing the notions of complex transaction, commit, rollback, etc. as they appear in relational databases.

Traditional DBMSs are passive, that is, transactions are executed only when they are invoked explicitly. In contrast, an active DBMS (ADBMS) continuously monitors the state of the database and reacts as specified by *active rules*. These rules have also been called *Event-Condition-Action* or ECA rules [18, 19]. In this context, an ECA rule's action corresponds to a complex database transaction; to avoid confusion we will refer to a these actions as rule actions. On the other hand, events in an ECA rule correspond to changes in the database tables. In our formalization, the fact that a change in

a database table has occurred will be stored in auxiliary tables; then, events are modeled as simple formulas that refer to the values stored in these auxiliary tables. Therefore, from a logical viewpoint, events and conditions will be modeled as conditions on the state of the database.

In order to formalize the notion of active rule, we use the notion of occurrence which allows us to specify that some actions must occur given specific conditions. These conditions might refer to the occurrence of some event(s), or to the actual state of the database. The applicability conditions for active rules are modeled as conditions for action occurrences. Since events and actions in active rules are expressed in terms

[2]The terms *update operation*, *primitive transaction*, and *action* are used interchangeably.

of database transactions, it is also necessary to provide an account for (complex) transactions a s they appear in real DBMSs. In part, this work has been directed towards providing a unifying criteria regarding the meaning of active rules, and to provide a framework for comparing different proposals for active database systems.

This chapter is structured as follows. Section 1.2 describes the execution modes of active rules, in particular of rules in Starburst [17]. Section 1.3 introduces *Sitcalc* and specifications of database updates. It continues with a more precise description of the task of specifying transactions and active rules. The section ends with the introduction of occurrences in *Sitcalc* [13]. Section 1.4 shows how to represent and specify database transactions and rollbacks. Active rules in the *Sitcalc* are specified in section 1.5. Section 1.6 shows an example of reasoning from an specification containing active rules. Finally, section 1.7 presents conclusions and future work.

1.2 Active Rules

The syntactic structure of active rules is similar in many DBMSs. Nevertheless, some fundamental aspects (for instance, the exact instant of execution of rules and the way events are considered) may differ radically from one system to another. In this section we review some issues regarding the different ways in which active rules can be handled (for more on these issues, consult [7, 18]).

An active rule is composed of three basic elements: *Events*, *Conditions* and *Actions*. The informal meaning is "Whenever the *Event* occurs, evaluate the *Condition*; if it holds, then execute the *Action*". An event is some change in the extension of database relations. In our proposal, these changes are reflected on the contents of auxiliary tables that record these changes. Thus, from a logical point of view, an event is a truth condition in a state. This truth condition refers to the auxiliary tables.

There is no standard terminology to refer to the process of rule triggering and execution. We choose the following (inspired by [8]): An ECA rule is said to be fired if its Event part is satisfied. Among all the rules that are fired, one or more may be considered for execution. If a rule is considered for execution, and its Condition part is satisfied, then its Action part is executed.

There are several possible semantics for the behavior of an active rule. In this regard, some important issues must be considered:

Granularity. Given that many DBMS operate on sets of tuples, there are several possible choices for rule firing. First, rules can be processed after the modification of individual tuples. Second, rules can be processed after all modifications specified by an update operation have been performed. Finally, rules can also be processed at the end of the execution of the transaction.

Simultaneous firing of rules. In some languages, the simultaneous firing of rules are disallowed. If this is not the case, then the execution may be sequential (using some form of conflict resolution) or concurrent. We consider the case in which rules have priorities for conflict resolution.

Rule chaining. When a rule is executed, it can lead to a state where other rules might also be applicable, chaining their execution.

This possibility opens the problem of termination of the execution [1].

Scope of the transaction. In general terms, active DBMSs consider the execution of a rule inside the range of execution of the transaction where the event occurred, but there are systems with higher flexibility in the relationship between rules and transactions.

We use Starburst as a model in order to describe our approach to specify the semantics of active rules. In the next section we describe how Starburst addresses these issues.

1.2.1 Starburst

Active rules in Starburst [17] are associated to the notion of a *transition*. The action in a rule may be a sequence of SQL operations. When this sequence is executed, it gives rise to a transition. A transaction may involve several transitions; the first transition results from the execution of the operations explicitly specified in the user transaction. Transitions also arise from the execution of the actions that are associated to rules.

A rule may mention several alternative events. If at least one of them occurs, the rule fires. In Starburst, all rules are assigned a priority. The conditions of the fired rules are evaluated in order of priority. The first rule whose *condition* holds, is *executed*. If a rule has been fired, and its condition evaluated, it is said to have been *considered* (regardless of whether or not its *condition* holds). After a rule is executed, the process starts again and stops when no rules can be executed.

To fire an active rule in Starburst, the net effects of the transitions are considered, that is, changes caused on the tuples as a consequence of a set of SQL operations are seen as effects of a single operation. For example, if a tuple is updated and then deleted, only the deletion of the original tuple is considered.

Notice that an original user transaction can be extended by the execution of active rules. At the end of the original transaction and at the end of a rule execution, rules are evaluated for firing. A rule is fired if one or more of the events that fire it has occurred since the last time the rule was considered (see above for the difference between firing and considering a rule), or since the beginning of the original transaction if the rule has not been considered yet. The events are always checked with respect to the net effects, and only at the end of the user transaction (but before committing) or at the end of the transaction associated to an executed active rule, and not inside a transaction or rule execution. Starburst has a natural execution semantics, each rule considers each event exactly once. Thus, events that make a rule to be considered, are forgotten when the rule is evaluated for firing at a later time. According to this semantics, rules can be fired at the end of the states resulting from execution of user transactions (transitions) or after transitions generated by the execution of other rules.

1.3 Specifying Database Updates

In this section we briefly describe *Sitcalc* and specifications of databases updates by primitive, non decomposable transactions. In *Sitcalc* there are sorts $\mathcal{I}$, $\mathcal{A}$ and $\mathcal{S}$ for domain individuals, actions (or primitive transactions) and situations, respectively. First order variables and quantification are used for each sort. In formulas, variables are implicitly assumed to be universally quantified with maximum scope. Basic dynamic properties, called *fluents* elsewhere, are represented with predicates that take one situation argument (its last), and are of the form $F(\bar{x}, s)$. Their extensions correspond to the contents of a database table at situation s.

There is distinguished constant, S_0, to denote the initial situation, along with a successor function, do, such that the term $do(a, s)$ denotes the situation that results from executing a in situation s. The following axioms characterize the space of situations:

$$S_0 \neq do(a, s), \tag{1.1}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2), \tag{1.2}$$

$$(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \supset P(do(a, s))) \supset (\forall s)P(s)]. \tag{1.3}$$

The last axiom is a second order induction axiom on situations. With these axioms, situations form a tree whose root is the initial situation.

There is also a predicate, $Poss(a, s)$, with the intended meaning that (the execution of) a is possible at situation s. This predicate has to be axiomatized giving preconditions for all the named actions in the language. A binary predicate, $<$, between situations can be defined with the intended meaning that $s < s'$ holds when s' can be reached from s by executing a finite sequence of possible actions. This definition is based on the induction axiom plus the following constraints, where $s \leq s'$ is an abbreviation for $s < s' \vee s = s'$:

$$\neg s < S_0, \;\; s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s'). \tag{1.4}$$

A formal specification of the dynamics of a database for primitive transactions is expressed by the axioms [10]: $\Sigma = \mathcal{D}_{found} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ssa}$, where:

1. $\mathcal{D}_{found}$ contains the foundational axioms of the situation calculus (1.1) - (1.4) .

2. $\mathcal{D}_{S_0}$ is the initial database as a theory; it describes the world at the initial situation S_0.

3. $\mathcal{D}_{una}$ contains unique names axioms for actions.

4. $\mathcal{D}_{ap}$ contains the action precondition axioms of the form

   $$\forall s \forall \bar{x} \; Poss(A(\bar{x}), s) \equiv \Pi(\bar{x}, s)$$

   , for each named parameterized action A. Here, formula $\Pi(\bar{x}, s)$ is simple in s[3], which means that its only situation term is the free variable s, and it does not mention $Poss$.

[3]This means that s the only situation term appearing in the formula.

5. $\mathcal{D}_{ssa}$ contains the "successor state axioms" (SSAs). They characterize the evolution of the database by describing, for each table F, when $F(\bar{x}, do(a,s))$ becomes true, that is when the property F holds at a successor state that results from executing a possible action a in situation s[4]. They have the form:

$$\begin{aligned} &Poss(a,s) \supset F(\bar{x}, do(a,s)) \equiv \\ &\quad [\Psi_F(\bar{x}, a, s) \vee (F(\bar{x}, s) \wedge \neg\Psi_{\neg F}(\bar{x}, a, s))], \end{aligned} \tag{1.5}$$

   where Ψ_F and $\Psi_{\neg F}$ are formulas simple in s. They describe the only actions and conditions that can make F respectively true or false at the situation $do(a,s)$.

In addition, we may have integrity constraints which should be satisfied after transactions are executed. These integrity constraint do not play the role of *ramification* or *qualification* constraints [11], rather they specify conditions that should be satisfied after transactions are finished. If these constraints are not satisfied, then a rollback should be performed, otherwise, the transactions are committed. Logically, these constraints could be written as:

$$(\forall\, s, s')\ S_0 \leq s \wedge s = do(Commit, s') \supset \gamma(s),$$

where $Commit$ is a special action, introduce later, used to perform the actual commitment of a transaction. As specified in the axiomatization of $Commit$, one of its preconditions is that the γ constraints be satisfied.

Example: We consider a simple database of a company with the table $Emp(x,s)$, meaning that person x is an employee of the company when the database is in situation s, and with the primitive transactions $hire(x)$ and $fire(x)$. In this case there is one successor state axiom:

$$\begin{aligned} &\forall(a,s) Poss(a,s) \supset \forall x[Emp(x, do(a,s)) \equiv \\ &\qquad\qquad a = hire(x) \vee Emp(x,s) \wedge a \neq fire(x)], \end{aligned}$$

and two action precondition axioms:

$$\begin{aligned} &\forall(x,s)[Poss(hire(x), s) \equiv \neg Emp(x,s)], \\ &\forall(x,s)[Poss(fire(x), s) \equiv Emp(x,s)]. \end{aligned}$$

1.3.1 Occurrences in the Situation Calculus

¿From the logical specification of the dynamics of change of a traditional database, it is possible to reason about all its possible legal evolutions. However, these specifications do not consider transactions that *must occur* given certain environmental conditions (i.e., the database state). In order to extend this style of specification to ADBMSs, it is necessary to extend the situation calculus with *occurrences*. This is necessary, given that the actions specified by active rules must be *forced to occur* when the associated

[4] These axioms solve the problem of specifying both changes and persistencies in a succinct way.

event happens and the corresponding condition is satisfied. That is, the future is not open to all possible evolutions, but constrained by the necessary execution of actions mentioned in the rules that fire, given that their related conditions hold.

The notion of occurrence in *Sitcalc* we use here is based on [13]. The starting point is the observation that every situation s identifies a unique sequence of actions. That is, situations can be identified with histories of action executions: $s = do(a_n, \ldots (do(a_2, do(a_1, S_0)) \ldots)$. We say that the actions $a_1, a_2, \ldots, a_n$ have occurred before s. Next, the predicate $proper$ is introduced to characterize those situations whose histories are considered to be proper with respect to the domain that is being axiomatized. In particular, when modeling active databases, we consider that situations are proper when their histories are consistent with the semantics given to the rule execution.

Thus, the predicate $occurs$ is introduced, such that $occurs(a, s)$ represents the fact that action a occurs in situation s. If this holds, then the situation $do(a, s)$ must be in the history of every proper situation that is accessible from s:

$$\begin{aligned}&proper(s_h) \equiv \\ &\quad \forall a, s\ (s < s_h \wedge occurs(a, s) \supset do(a, s) \leq s_h).\end{aligned} \tag{1.6}$$

Finally, we want to consider only those situations that are contained in a proper history. For this purpose, the predicate *legal* is introduced, and defined in order to axiomatize the intuition that a situation or history s is legal if for every (posterior) situation accessible from s, there is a proper future with s in its history:

$$legal(s) \equiv \forall s'\ s < s' \supset \exists s_h\ s' \leq s_h \wedge proper(s_h). \tag{1.7}$$

In summary, a situation is legal, if it results from the execution of possible actions, and all occurrence constraints are satisfied by its history. Notice that *legal* is defined in terms of $proper$, in consequence, with *legal* we can consider a sub-tree of the situation structure, whose branches contain proper future situations.

Whenever the condition of an active rule in consideration is satisfied, the action mentioned in the rule must occur (be executed). In a database whose situations are all legal there will be only proper futures. Thus,

in a situation calculus tree we only consider branches in which the given sequence of operations has occurred. Therefore, the specification of an active rule must include the presence of the predicate $occurs$ associated to sequence of operations.

1.4 Specifying Transactions

In *Sitcalc*, transactions will be represented as finite lists of primitive actions, that is of elements of the sort $\mathcal{A}$. We introduce a new sort, $\mathcal{L}$, constructed from $\mathcal{A}$. We denote a generic variable for lists with l. We consider (a) The function **;** : $\mathcal{L} \times \mathcal{A} \rightarrow \mathcal{L}$ as a list constructor; (b) An extended function do so that it includes the execution of lists of primitive transactions, that is, $do\ :\ \mathcal{L} \times \mathcal{S} \rightarrow \mathcal{S}$; (c) The empty list, [], with no effects on a situation. The basic axiomatization for lists executions is:

$$\begin{aligned}&do([], s) = s, \\ &do(l; a, s) = do(a, do(l, s)).\end{aligned}$$

Also, the usual predicates on lists can be introduced in an obvious manner: $member$, to represent the membership of an action to a list, *first*, *last*, and *tail* are predicates with their usual meanings.

Finally, the $Poss$ predicate can be extended to finite sequences of transactions:

$$Poss([], s) \equiv True,$$
$$Poss([a_1, \ldots, a_n], s) \equiv \bigwedge_{k=1}^{n} Poss(a_k, do([a_1, \ldots, a_{k-1}], s)).$$

Since transactions will start with new actions $Begin$ and $Commit$ of the sort $\mathcal{A}$, we need to distinguish the transaction body that does not include these limit actions:

$$transaction\_body(l) \equiv$$
$$\forall a\ (member(a, l) \supset a \neq Begin \wedge a \neq Commit).$$

A transaction is a list of actions delimited by the actions $Begin$ and $Commit$:

$$transaction(l) \equiv$$
$$(\exists\, l', l'')\ l = (l'; End) \wedge first(Begin, l') \wedge tail(l'', l') \wedge transaction\_body(l'').$$

Initially, we consider the following preconditions for the primitive transactions $Begin$ and $Commit$ (later on we will modify these preconditions):

$$Poss(Begin, s) \equiv True,$$
$$Poss(Commit, s) \equiv \gamma(s),$$

where $\gamma(s)$ is a condition that must hold for the transaction to be committed. Formula $\gamma(s)$ comes from a (finite conjunction of) static integrity constraint(s) of the form $\forall s\ \gamma(s)$.

In [4] a technique for treating dynamic integrity constraints as static integrity constraints is presented.

Notice that the actions $Commit$ and $Rollback$ do not produce any changes on the tables of the database by themselves. This is because they do not appear in the SSAs of the tables presented in section 1.3. Actions $Commit$ and $Rollback$ have a meaning only at the end of a transaction, whose effects are made visible by the $Commit$ and forgotten by the $Rollback$. This is the case when, before attempting a commit, the integrity constraints are not satisfied.

During the execution of a transaction, a $Rollback$ operation cancels the transaction and returns the database to the state of the database at the beginning of the transaction. Thus, a rollback produces a new situation whose state is equivalent to the state of the database at the time the transaction started.

In a first attempt at formalizing rollback, we introduce new fluents that would *copy* the fluents associated to the database relations at the situation where the transaction is started (i.e., at the time of the execution of a *Begin* action). I.e., for each fluent *F*, we introduce a new fluent B_F, in the following manner:

$$
\begin{aligned}
&Poss(a,s) \supset \\
&\quad B_F(\bar{x}, do(a,s)) \equiv (F(\bar{x},s) \wedge a = Begin) \vee \\
&\qquad\qquad\qquad\qquad (B_F(\bar{x},s) \wedge a \neq Begin).
\end{aligned}
$$

Thus, every time a transaction is started, with a *Begin* action, all fluents are *backed up* in auxiliary fluents B_F. These fluents are utilized in order to restore the fluent values in the event of a rollback. Therefore, to account for this behavior, the successor state axioms for a fluent F would be rewritten as:

$$
\begin{aligned}
&Poss(a,s) \supset \\
&\quad F(\bar{x}, do(a,s)) \equiv [\Psi_F(\bar{x},a,s) \vee \\
&\qquad\qquad\qquad (F(\bar{x},s) \wedge \neg\Psi_{\neg F}(\bar{x},a,s))] \wedge \\
&\qquad\qquad\qquad a \neq Rollback \vee \\
&\qquad\qquad\qquad B_F(\bar{x},s) \wedge a = Rollback.
\end{aligned}
$$

These SSAs can be easily rewritten in the form (1.5).

1.5 Representing Active Rules

Now, we introduce a new sort, $\mathcal{R}$, for rules (and variables for this sort $r, \ldots$). We also introduce a rule name of sort $\mathcal{R}$, a constant, $\mathtt{rule}_{R_i}$, for each of the rules R_i $(i = 1, \ldots, m)$ specified in the active database. We add unique names axioms for rules: $\mathtt{rule}_{R_i} \neq \mathtt{rule}_{R_j}$ for $i \neq j$; and domain closure for rules: $\forall r \ \bigvee_{i=1}^{m} r = \mathtt{rule}_{R_i}$.

These new elements, allow us to distinguish rules from their associated sequences of actions. Furthermore, we introduce new, auxiliary predicates that play a role similar to the tables associated to update events in Starburst [17]. A dynamic predicate, $V_{F,R}$, which holds for the instances of table F that become true (and were false before) during a transition. Analogously, $V_{\neg F,R}$, which holds for the instances that become false (and were true before). These predicates are used to detect the occurrences of events associated to rules R; that is, events are identified with specific updates in tables.

The action $Begin$, being the beginning of the user transaction, initializes all the auxiliary predicates as empty. Furthermore, a $Considered$ predicate is required. It also initializes the auxiliary predicates as empty for those rules *R* considered according to Starburst's specification. The fluent $Considered$ is specified below.

We need successor state axioms for the auxiliary predicates below.

$$
\begin{aligned}
&Poss(a,s) \supset \\
&\quad V_{F,R}(\bar{x}, do(a,s)) \ \equiv \ [\Psi_F(\bar{x},a,s) \vee \\
&\qquad\qquad\qquad (V_{F,R}(\bar{x},s) \wedge \\
&\qquad\qquad\qquad \neg(a = Begin \vee Considered(\mathtt{rule}_R, s)))].
\end{aligned}
\tag{1.8}
$$

$$
\begin{aligned}
&Poss(a,s) \supset \\
&\quad V_{\neg F,R}(\bar{x}, do(a,s)) \ \equiv \ [F(\bar{x},s) \wedge \Psi_{\neg F}(\bar{x},a,s) \vee \\
&\qquad\qquad\qquad (V_{\neg F,R}(\bar{x},s) \wedge \\
&\qquad\qquad\qquad \neg(a = Begin \vee Considered(\mathtt{rule}_R, s)))].
\end{aligned}
\tag{1.9}
$$

Here, Ψ_F and $\Psi_{\neg F}$ are the formulas appearing in (1.5), the SSA for base table F.

Now, we specify the basic transitions, that is, the initial list of actions corresponding to the user transaction, and the lists of actions associated to rules. The initial list is a sequence of actions delimited by actions $Begin$ and End. The list associated to a rule R always ends with an End action. We do not use $Commit$ instead of End, because it is reserved for the end of the (user) transaction. We will also use a predicate $transition\_body$ defined in terms of $transaction\_body$, that allow us to prevent lists of actions from containing any of the distinguished auxiliary actions we introduced before:

$$transition\_body(l) \equiv transaction\_body(l) \wedge \neg member(End, l).$$

$$\begin{aligned} initial\_list(l) \equiv & first(Begin, l) \wedge last(End, l) \wedge \\ & (\forall a)\ (member(a, l) \wedge \neg first(a, l) \wedge \neg last(a, l) \supset \\ & \qquad a \neq Begin \wedge a \neq End) \end{aligned}$$

Also, we define the predicate *rule_list* to specify the action list associated to a rule.

$$\begin{aligned} rule\_list(\mathtt{rule}_i, l) \ \supset \ (\exists\, l')\ l = l'; end \wedge \\ transition\_body(l'), \end{aligned} \tag{1.10}$$

Furthermore, the list of actions must be unique for each rule, that is, we need the axiom $rule\_list(r, l) \wedge rule\_list(r, l') \supset l = l'$.

In a particular application, the predicate $rule\_list$ should be axiomatized directly, by giving the lists of specific actions associated to each rule. For instance, if rule R_i is $(Event; Condition; a_1, \ldots, a_n)$, where the a_i's are the primitive actions to be executed when $Event$ and $Condition$ apply, then we introduce the axiom:

$$rule\_list(R_i, [a_1, \ldots, a_n, End]).$$

Now we want to specify a predicate $Poss$ relative to rules, in such a way that *if a rule is possible, then the list of actions associated to it will occur*. A rule is possible if it has been considered by an event and its condition holds.

First, we specify that a rule is fired in a situation as:

$$Fired(R, s) \equiv Event_R(s) \wedge AssertionPoint(s). \tag{1.11}$$

where $Event_R(s)$ is a formula that describes the event that fires the rule (represented by entries in the auxiliary tables). [5] $AssertionPoint(s)$ states that a rule can be executed at the end of the execution of the user transition or at the end of the sequence associated to a rule [1], i.e.:

$$AssertionPoint(s) \equiv \exists s'\ s = do(End, s'). \tag{1.12}$$

[5] That is, $Event_R(s)$ is expressed in terms of $V_{F,R}$ and $V_{\neg F,R}$. It could be expressed in terms of views defined on these predicates. As shown in [3], it is possible to derive successor state axioms for views, which can be used in the same way as successor state axioms for the base predicates.

Second, a rule R is considered if, after it has been fired, its conditions have been evaluated. The rule is taken to be considered whether or not its condition is satisfied. Notice that for a rule to be considered there can be no other rule of higher priority that is at the same time considered and with its condition satisfied. Thus, we may write:

$$\begin{array}{l} Considered(R,s) \equiv Fired(R,s) \wedge \\ \quad \bigwedge_{i=1}^{m}((Fired(rule_{R_i},s) \wedge \\ \quad priority(\mathtt{rule_{R_i}}) > priority(R)) \supset \neg Cond_{R_i}(s)). \end{array} \tag{1.13}$$

Where, $Cond_R(s)$ refers to the additional conditions, expressed as a formula, which may involve both the auxiliary tables and the base tables. Also, $priority$ is a function: $\mathcal{R} \to I\!N$, (we may consider $I\!N$ as a new sort with an underlying specification) which introduces a total order between rules. For this function it must hold:

$$\forall r, r' \; r \neq r' \supset priority(r) \neq priority(r'). \tag{1.14}$$

This sentence establishes a total order between the elements of $\mathcal{R}$.

Third, a rule R is possible if:

$$\begin{array}{rl} Poss(R,s) \equiv & Considered(R,s) \wedge Cond_R(s) \\ & \wedge\, (rule\_list(\mathtt{rule}_R, l) \supset Poss(l,s). \end{array} \tag{1.15}$$

Notice that only one rule can be possible at each assertion point. This fact follows from the formalization presented in this section (in particular, the use of the priority assignment function).

Even when the conditions for an active rule are satisfied, we have to make sure that it will be executed, that is, the associated sequence of actions must occur. Therefore, we extend the predicate $occurs$ to the case of sequences of actions.

$$occurs(l;a,s) \equiv occurs(a, do(l,s)) \wedge occurs(l,s). \tag{1.16}$$

In our specification we have the axiom:

$$Poss(r,s) \wedge rule\_list(r,l) \supset occurs(l,s). \tag{1.17}$$

Predicate $occurs$ will force the execution of the sequence of actions associated to the a rule when the conditions for the execution of the rule are satisfied.

Finally, notice that for executing the $Commit$, that ends the transaction, there should be no pending rules. That is, we modify its precondition to:

$$\begin{array}{l} Poss(Commit,s) \equiv \gamma(s) \wedge AssertionPoint(s) \wedge \\ \qquad \forall r\, \neg Poss(r,s), \end{array} \tag{1.18}$$

where $\gamma(s)$ are the integrity constraints to be satisfied.

Now, we have to state that $Commit$ and $Rollback$ have to be executed as appropriate. In order to trigger a $Commit$, we add:

$$(\forall\, s)\; s \geq S_0 \wedge Poss(Commit,s) \supset occurs(Commit,s).$$

For triggering the $Rollback$ action, we add:

$$(\forall\, s)\; s \geq S_0 \wedge AssertionPoint(s) \wedge [(\forall\, r)\; \neg Poss(r,s)] \wedge \neg\gamma(s) \supset occurs(Rollback,s).$$

1.6 Reasoning from the Specification

Given the specification of the dynamics of the relational database, including a specification of the active rules in terms of the way their checking and triggering interact with the database evolution, one can perform reasoning from the specification. Among others, we may find the following reasoning problems that can be formulated as obtaining corresponding sentences as logical consequences of the specification[6].

1. There are well known problems regarding the behavior of transactions; according to [1], the following are particularly relevant:

   Termination: i.e., whether one can guarantee that the transactions will end.

   Confluence: In some systems it is possible to arrive at a situation in which several rules can be executed at an assertion point. In this case, one would be interested in proving that all possible executions lead to the same state of the database.

   Given a ground user transition l_U, the termination problem can be posed in the following form:

   $$\begin{array}{l}\Sigma \cup \Omega \models \exists s\; initial\_list(l_U) \wedge s = do(l_U, S_0) \supset \\ \qquad \exists s'\; s \leq s' \wedge legal(s') \wedge Poss(Commit, s').\end{array} \tag{1.19}$$

   Here Σ is the specification of the dynamics of the database as proposed by Reiter (without the specification of the active rules), and Ω is the specification of the active rules as presented before. The condition on the possibility of executing $Commit$ can be replaced by its definition (1.18).

   The confluence problem arises in systems in which there is no complete conflict resolution strategy. To model this situation, one would need to modify or eliminate the priorization of rules, leading to a modification of the specification of $Considered$ (1.13). To deal with this problem, one needs to consider all possible execution sequences, and compare the states that result from each execution sequence.

2. Given a ground user transition l_U, determine if the active rules preserve the integrity constraints at the end of the corresponding user transaction:

   $$\begin{array}{l}\Sigma \cup \Omega \models \exists s\; initial\_list(l_U) \wedge s = do(l_U, S_0) \supset \\ \qquad (\exists s'\; s \leq s' \wedge legal(s') \wedge AssertionPoint(s') \wedge \\ \qquad \forall r\; \neg Poss(r, s') \supset \gamma(s')).\end{array} \tag{1.20}$$

3. Given a ground user transition l_U and a rule R, determine if the rule will be executed along the execution of the corresponding transaction:

   $$\begin{array}{l}\Sigma \cup \Omega \models \exists s\; initial\_list(l_U) \wedge s = do(l_U, S_0) \supset \\ \qquad \exists s'\; s \leq s' \wedge legal(s') \wedge Poss(\mathtt{rule_R}, \mathtt{s'}) \\ \qquad \wedge \neg \exists s''\; s \leq s'' \leq s' \wedge Poss(Commit, s'').\end{array} \tag{1.21}$$

[6] In an extended version of this chapter we will deal with these reasoning problems. Here we restrict ourselves to their formulation in the context of the specification

Example: The following example, from [17], allows us to illustrate statement (1.20). It is based on the tables:

$$Emp(emp\_no, name, salary, dept\_no, s),$$
$$Dept(dept\_no, mgr\_no, s).$$

We want to specify a variation of the cascade elimination methodology for maintaining the referential integrity constraint.

In this case, the only existing rule is R: "If an employee is eliminated from Emp (the Event), then the department managed by that employee must be eliminated with all its employees (the Action)". There is no condition to be evaluated here. We expect that after executing this rule, and at the end of the user transaction, (1.20) will hold, that is the referential consistency will be restored.

In this particular example, the Σ specifies actions, del_emp and del_dept, that eliminate employees and departments, respectively., and are always possible. Successor state axioms for Emp and $Dept$ are also included in Σ. The specification is extended by Ω, which includes axioms for sequences, successor state axioms for the auxiliary tables, and axioms for active rules.

The event associated to rule R (the other elements of the specification given in previous sections are general) is specified by:

$$\begin{aligned} Event_R(s) \equiv\ & (\exists\, emp\_no, name, salary, dept\_no) \\ & \quad V_{\neg Emp,R}(emp\_no, name, salary, dept\_no, s). \end{aligned}$$

The transition, l_R, associated to the rule is:

$$del\_emp_R(emp\_no, name, salary, dept\_no); del\_dept_R(dept\_no, mng); End.$$

Here del_emp_R and del_dept_R are new primitive transactions associated to R. They are always possible; they have the intended effects, but only on the right conditions: they eliminate employees and departments which are associated to managers already eliminated by means of the action del_emp (what is recorded in the auxiliary tables). These new actions with these conditions have to be inserted in the SSAs for Emp and $Dept$.

If the user transaction is such that its associated transition, l_U, is:

$$Begin; del\_emp(E, N, S, D); End,$$

then this is the transition term that must be introduced in (1.20). The integrity constraints, $\gamma(s)$, is the conjunction of

$$Emp(emp\_no, name, salary, dept\_no, s) \supset \exists mng\ Dept(dept\_no, mng, s)$$

and

$$\begin{aligned} Dept(dept\_m, mng, s) \supset\ & (\exists\, name\_m, salary\_m, dept\_no) \\ & \quad Emp(mng, name\_m, salary\_m, dept\_no, s). \end{aligned}$$

We could prove (1.20) by means of derived induction principle on states that allows jumping from states at ends of user and rules transactions to new states of the same kind, because intermediate states, along transitions, may violate the integrity constraint momentarily.

1.7 Conclusions and Further Work

In this chapter, we have shown that it is possible to introduce the notion of active rule in the specification of the dynamics of a relational database. The specification ensures that rules will be executed in appropriate conditions and with the expected effects. In this way, the specification of the dynamics of active rules is integrated in a homogeneous and common formalism with the specification of the dynamics of the underlying tables. All the advantages of the original specifications are inherited by the extended specification, in particular, the possibility of integrating in a single formalism the usual extensional aspects of databases and their dynamic aspects; and the possibility of doing hypothetical reasoning with respect to to a virtual development of the database [4, 2], and the possibility of proving properties of the evolution of the database. For instance, satisfaction of integrity constraints [4, 5], or the satisfaction of a particular goal when active rules are executed (for example, termination).

We can see that in the context of our *Sitcalc* based specifications of database updates and active rules, these rules inherit a clear and well understood semantics. Namely, the semantics of predicate logic. New semantics are not needed.

Apart from the specification of active rules, in this chapter we specified some useful and common notions of databases, in particular, of the rollback operation.

Most of the specification we gave is domain independent and can be used with different specific cases of active rules. In this sense, the specification is modular.

Also, we considered the case of conditions in rules that are evaluated statically. Nevertheless, temporal conditions in active rules are also interesting [15, 16]. For example, a usual condition like non decreasing salaries for employees falls in this category. We can handle this case in our specification by accommodating history encoding [6] in the context of specifications of database updates, as done in [2].

Acknowledgments:

This research has been partially financed by FONDECYT (Grants 1971304 and 1980945), ECOS/CONICYT (Grant C97E05), DAAD and the Catholic University of Chile (DIPUC).

Afterword:

(By Leopoldo Bertossi): I became interested in doing research in databases after reading Ray Reiter's papers on foundations of databases. There I discovered that databases could be a fascinating area for scientific research. Having coauthored a couple of papers with Ray was also an extremely stimulating scientific experience for me. In my research activity, I have always had Ray as a model of conceptual depth, clarity and excellent scientific writing. I am very glad to have had the opportunity to interact with Ray, as a great person and scientist.

(By Javier Pinto): I met Ray Reiter in 1989 when I started working towards my Ph.D. degree at the University of Toronto. I feel now, as I felt then, that I was extremely privileged for having him as my thesis supervisor. So, it is a great pleasure to be able to contribute to this volume in honour of Ray. As acknowledged by everybody in the field, Ray is an outstanding researcher who has made central contributions to the foundations of AI. He has also contributed to the understanding of the close relationship between the fields of Knowledge Representation and Databases. I am forever in debt with Ray, and the more I know him, the more I admire him, not only as a scientist, but also as a human being.

Bibliography

[1] A. Aiken, J. Widom, and J. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proc. ACM-SIGMOD Conference*, pages 59–68, 1992.

[2] M. Arenas and L. Bertossi. Hypothetical Temporal Queries in Databases. In A. Borgida, V. Chaudhuri, and M. Staudt, editors, *Proceedings "ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98): Innovative Application Programming and Query Interfaces*, 1998. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.

[3] M. Arenas and L. Bertossi. The Dynamics of Database Views. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 197–226. Springer-Verlag, 1998.

[4] L. Bertossi, M. Arenas, and C. Ferretti. SCDBR: An Automated Reasoner for Specifications of Database Updates. *Journal of Intelligent Information Systems*, 10(3), 1998.

[5] L. Bertossi, J. Pinto, P. Saez, D. Kapur, and M. Subramaniam. Automating Proofs of Integrity Constraints in the Situation Calculus. In Z. W. Ras and M. Michalewicz, editors, *Foundations of Intelligent Systems*, pages 212–222. Springer, LNAI 1079, 1996.

[6] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, 20(2):149–186, June 1995.

[7] U. Dayal, E. Hanson, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database System: The Object Model, Interoperability, and Beyond*. ACM Press, 1995.

[8] K. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of adbms fe atures. In *2nd Workshop on Rules in Databases (RIDS)*, Athens, Greece, 1995. Lecture Notes in Compure Science, Springer.

[9] U. Jaeger and J.C. Freytag. An Annotated Bibliography on Active Databases. *SIGMOD Record*, 24(1):58–69, 1995.

[10] F. Lin and R. Reiter. State Constraints Revisited. *Journal of Logic and Computation. Special Issue on Actions and Processes*, 4(5):655–678, 1994.

[11] Fangzhen Lin and Raymond Reiter. State Constraints Revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994. Special Issue on Actions and Processes.

[12] John McCarthy. Situation, actions, and causal laws. Technical Report Memo 2, Stanford Artificial Intelligence Project, 1963.

[13] Javier Pinto. Occurrences and Narratives as Constraints in the Branching Structure of the Situation Calculus. To appear in Journal of Logic and Computation.

[14] R. Reiter. On Specifying Database Updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[15] A. P. Sistla and O. Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proc. ACM SIGMOD'95*, pages 269–280, 1995.

[16] A. P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Data and Knowledge Engineering*, 7(3):471–486, 1995.

[17] J. Widom. The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):583–595, 1996.

[18] Jennifer Widom and Stefano Ceri. *Active Database Systems, Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

[19] C. Zaniolo, S. Ceri, Ch. Faloutsos, R. T. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.