Formal Characterizations of Active Databases: Part II

Chitta Baral*[1] and Jorge Lobo**[2] and Goce Trajcevski[2]

[1] Univ. of Texas at El Paso, El Paso, TX, 79968, USA,chitta@cs.utep.edu
[2] Department of EECS, Univ. of Illinois at Chicago, 851 S. Morgan St, Chicago, Il 60607, USA {jorge,gtrajcev}@eecs.uic.edu

Abstract. This paper presents a formal framework for specifying active database systems. Declarative characterization of active databases allows additional flexibility in defining an implementation–independent semantics of the active rules. By making a clear distinction between actual and hypothetical execution of the actions, one can make claims and about the (possible) effects of an actions' sequence and prove them, without actually executing it. The results that we present extend the active database description language introduced in [5] with additional semantic dimensions. We demonstrate through examples how we can encode the active rules and their operational behavior from different existing systems.

1 Introduction and Motivation

The core concept which makes a database system active is the concept of an active rule. The origin of the active rules is the production rule paradigm from the field of Artificial Intelligence with the languages like OPS5 [7], used in expert systems. Typically, a production rule is of the form *condition*→*action*, where an inference engine cycles through all the rules and matches the condition part with the data in the working memory. Active rules, on the other hand, typically follow the *event-condition-action* (ECA) paradigm which specifies an action (possibly a sequence of actions) to be executed when a given event occurs, provided that certain conditions hold. The reactive capabilities of active databases are useful for many applications, such as views [9, 10], integrity constraints [36, 8], and workflows. Several active database languages [33, 22, 21] have been proposed, and many systems and prototypes have been designed and, partially or completely, implemented [25, 28, 11, 19, 34, 39] (many systems are presented in the collection [40]). Each system has some active features, expressed in its own syntax and (operational) semantics. However, it can be noticed that sometimes rules with a similar form will behave differently in different systems ([24, 13], present surveys of several systems functionalities and operational semantics).

* Support was provided by the National Science Foundation, under grant Nr. IRI-9211662 and IRI-9501577.

** Support was provided by Argonne National Laboratory under contract Nr. 963042401.

The dissimilarities arise because there are many different functional features [30] that make a database system active and the existing systems have taken different choices among appropriate alternatives. This has resulted in recognizing [14, 30, 40] that there has been very little activity on formal foundations of active behavior. Some recent results on formal characterization of active database are [16, 18, 17, 5, 31, 37, 41, 42].

This paper extends the language $\mathcal{L}_{active}$ introduced in [5] to describe the semantics of active database systems. $\mathcal{L}_{active}$ is based on the action description language $\mathcal{L}_0$ of Baral, Gelfond and Provetti [4]. One of the advantages of $\mathcal{L}_0$ is the clear distinction the language makes between actual and hypothetical occurrence of an action. This feature provides a very important tool for an active database designer. Namely, the designer is enabled to *reason* about various effects of executing a sequence of actions [16] such as:

- Is the particular sequence executable?
- Will certain data hold after executing a particular sequence?
- Will certain events occur as a result of executing a given sequence or, equivalently, will a certain active rule be triggered?

In $\mathcal{L}_{active}$ we allow for the description of the effects of executing actions, definition of events, and definition of active rules which can encode different semantic options. The semantics of $\mathcal{L}_{active}$ is based on the automata-based semantics of action description languages [20] and allows us to define an entailment relation between an active database description and queries about the state of the database after execution of a sequence of actions. A state of the database is described both by the data and the events. This is similar to [18] where the authors present a very comprehensive study of the existing active database systems and various semantic dimensions (functional features) and also offer a formalism (EECA) for encoding them. The rules from EECA format are translated into a *core* format for which the execution semantics is given by an algorithm specified in a C–like language. In contrast to [18] our formalism is completely declarative. In our approach, given the description of the active database, the key feature of the semantics is a transition function which generates the evolution of the states when a particular sequence of actions is executed. This function is implementation-independent. As we demonstrate later in the paper, the separation of the *event definition* from the *active rule* allows us to specify more complex events that the ones in [18] (although not the full class of [29]). An important aspect of our language is that it can be easily extended to incorporate additional features such as concurrent actions [3] and deductive rules [26, 2, 27] using previous results from action description languages. Furthermore, there are straightforward translations of active database descriptions into logic programs that implement the entailment relation giving us a tool to automatically reason about hypothetical executions.

The remainder of the paper is structured as follows. We introduce the syntax and semantics of $\mathcal{L}_{active}$, with some examples from the existing systems. Next, we present more examples showing how different active database semantics can be captured in our language. Next, we formally describe the semantics and show we can ask queries about the database behavior based on the entailment relation. Last, we give a brief comparison with the existing works, draw some conclusions and indicate directions for future work.

2 Syntax of $\mathcal{L}_{active}$

We assume that there are four (possibly countably infinite) pairwise disjoint sets of symbols: $\mathcal{A}$ – action names, $\mathcal{F}$ – fluents, $\mathcal{E}$ – event names, and $\mathcal{R}$ – rule names; and a set of variables. Each symbol has an *arity* associated with it and literals from each set are defined as usual. Atoms from $\mathcal{A}$, $\mathcal{F}$, $\mathcal{E}$ and $\mathcal{R}$ are called *actions, fluents, events*, and *rule_ids* respectively. We will restrict the use of literals to fluent and event literals. There is a special action symbol $\uparrow$, called the processing point action symbol (see section 2.1

Fluents are data items which can change their values as the active database evolves. They can take different forms according to the kind of database being used. For example, in an object oriented database, they could be a name of an object attribute, together with a value from the domain of the attribute. In a relational model, fluents correspond to the tuples that can appear in the relations. Variables can be used in the literals and they represent parameters that can be replaced by any value from the underlying domain of the attributes. Our examples will be loosely based on the relational model, in order to minimize the introduction of new notation.

There are three types of propositions in $\mathcal{L}_{active}$:

(i) The first kind of propositions are the *causal* or *effect laws* which are expressions of the form:

$$a(\overline{X}) \textbf{ causes } f(\overline{Y}) \textbf{ if } p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}) \tag{1}$$

where $a(\overline{X})$ is an action and $f(\overline{Y}), p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are fluent literals ($n \geq 0$). $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are called *preconditions*. The intuitive meaning of (1) is that in any state of the active database execution in which $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are true, the execution of the action $a(\overline{X})$ causes $f(\overline{Y})$ to be true in the resulting state. The preconditions $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ will be evaluated as regular queries in the database and $a(\overline{X})$ is an action that could be invoked by a user or an active rule. Thus, variables appearing in $\overline{Y}$ or in any negated fluent in the preconditions must also appear in one of the positive fluents in the precondition. If there are variables in $\overline{X}$ that do not appear in any of the positive fluents in the preconditions this arguments most be ground at the time of the invocation of the action, otherwise there will be an error in the execution.

Example 1. The actions common to most database systems are the SQL operations *insert, delete* and *update*, which we will refer to as *add, del* and *upd* respectively. Consider, for example, an update request which will change the salary of all the employees whose name is *joe* in the department $d1$ to 2000, regardless of their current salary.

$$\begin{aligned} upd(emp.salary, 2000)) \ \textbf{causes} \ & emp(SS, joe, d1, 2000) \\ \textbf{if} \ & emp(SS, joe, d1, X) \\ upd(emp.salary, 2000) \ \textbf{causes} \ & \neg emp(SS, joe, d1, X) \\ \textbf{if} \ & emp(SS, joe, d1, X) \end{aligned}$$

This correctly reflects the semantics of the update which is, if the tuple exists, substitute it with a new one with modified values for the attributes. If the tuple does not exist, the effect is null.

In general, actions are the operations provided by the systems that can be applied to the data. Some of them may have no direct effect on the database, and there will not be causal laws associated with them. An example of such action could be the retrieve operations defined in Postgres. Other actions could be commit, abort, or even application procedures defined by the user. We would like to remark that the role of the causal laws is not the define how to implement the actions but to specify what are the effects of the actions in the data stored in the database. We assume that actions are atomic.

(ii) The *event definition* proposition is an expression of the form:

$$e(\overline{X}) \ \textbf{after} \ \ a(\overline{W}) \ \textbf{if} \ \ e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m}), q_1(\overline{Z_1}), \ldots, q_n(\overline{Z_n}) \qquad (2)$$

where $e(\overline{X}), e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m})$ are event literals and $q_1(\overline{Z_1}), \ldots, q_n(\overline{Z_n})$ are fluent literals. This proposition says that the execution of the action $a(\overline{W})$ ordered in a state in which each of the fluent literals $q_i(\overline{Z_i})$ is true *and* each of the event literals $e_j(\overline{Y_j})$ is true (i.e. the event in the event literal belongs to the current set of event if the event literal is positive, or it does not belong to the set if the event literal is negative) generates the event literal $e(\overline{X})$ (i.e. it is added to the set of current events) if the event literal is positive, or removes the event from the set of current events if the event literal $e(\overline{X})$ is negative. If the execution is ordered in a state in which some of the $q_i(\overline{Z_i})$ or $e_j(\overline{Y_j})$ does not hold than (2) has no effect. Each of the variables appearing in $\overline{X}$ or in a negated event or fluent literals, has to appear either in $\overline{W}$ or in a positive event/fluent literal.

Remark. 1) It is usually the case that some of the actions in a database system are part of the events, but is not standard which action will become an event. For example, in SQL-3, the events are insertions, deletions and updates. POSTGRES adds *retrieve* to the list. In addition, there are composite events (events defined by other events or a set of actions), and possibly clock–ticks. Thus, we would like to separate actions from events, and in case an action directly defines and event we must make an explicit definition.

2) Arguments in the events have two purposes: One is obviously to past information from the actions and other events that define the event to the condition

and action part of the active rules (to be defined). The second is to distinguish between set vs. tuple at the time events. For example, when a set of tuples is inserted in a relational table a rule can be trigger for each tuple inserted or the insertion as a whole. If there are no variables in the event, there will be only one event generated for the whole insertion. If there are variables, there will an event generated for each tuple. For example the event definition

$$e\_ins(emp) \textbf{ after } ins(emp(SS, N, D, Sal)$$

will generate an event after the insertion of a set of tuples in the *emp* relation. The following event definitions defines an event for each tuple inserted in the *emp* relation.

$$e\_ins(emp(SS, N, D, Sal) \textbf{ after } ins(emp(SS, N, D, Sal)$$

We can obtain other granularities by removing arguments for the event. Let say we would like an event for each department where an insertion is made. Then we will write

$$e\_ins(emp(D)) \textbf{ after } ins(emp(SS, N, D, Sal)$$

The default assumption is that the event *persist* from one state to another, with two possible exceptions: either the event is *consumed* by an active rule (see below), or the event is removed by an action based on the specification of an event definition. For example, if we have an expression $\neg e_1$ **after** a_1, the execution of the action a_1 will cause the event e_1 not to be present in the resulting state. Hence, the meaning of "an event is true in a given state"is: the event was induced (i.e. generated) in some state prior to the given one and the event persisted, or the event was induced by an execution of an action in the previous state.

Example 2. This example shows how event definition propositions enable us to capture the concept of net effects. Many database systems allow the execution of a set of actions in bulks before the active rules are processed and events are defined in terms of the *net effects* the bulk of actions has in the database (as suppose to the individual effect of each action). The complication here is that the definition of net effect varies from system to system. For example, the premises for defining a net effect in Starburst are:

- If a tuple is inserted and then updated, it is considered an insertion of the updated tuple.
- If a tuple is updated and then deleted, it is considered as a deletion of the original tuple.
- If a tuple is updated more than once, it is considered as an update from the original value to the newest value.
- If a tuple is inserted and then deleted, it is not considered in the net effect at all.

These four premises can be encoded in $\mathcal{L}_{active}$ as:

$$\begin{aligned} e\_add(H) &\quad \textbf{after} \ \ upd(G,H)\ \textbf{if} \ \ e\_add(G) \\ e\_del(G) &\quad \textbf{after} \ \ del(G)\ \textbf{if} \ \ e\_upd(G,F) \\ e\_upd(G,I) &\quad \textbf{after} \ \ upd(H,I)\ \textbf{if} \ \ e\_upd(G,H) \end{aligned} \qquad (3)$$

In addition, due to our assumption of the persistence of events, we need the following event definition propositions to remove events from the current set of events:

$$\begin{aligned} \neg e\_add(G) &\quad \textbf{after} \ \ upd(G,H)\ \textbf{if} \ \ e\_add(G) \\ \neg e\_upd(G,F) &\quad \textbf{after} \ \ del(G)\ \textbf{if} \ \ e\_upd(G,F) \\ \neg e\_upd(G,H) &\quad \textbf{after} \ \ upd(H,I)\ \textbf{if} \ \ e\_upd(G,H) \\ \neg e\_add(G) &\quad \textbf{after} \ \ del(G)\ \textbf{if} \ \ e\_add(G) \end{aligned} \qquad (4)$$

There are few observations that we need to make. A common use of active rules (for applications such as alerting) is the ability of being triggered *before* the execution of a particular action. We can model this behavior by a simple requirement that the elements of $\mathcal{A}$ are actually pairs. Each action a is specified as a pair $(a_{begin}, a_{execute})$. While the action part of a *causal law* could have only $a_{execute}$ type of symbols, the *event definition* would be allowed to have both types. Hence, the *before* behavior can be modeled by defining a triggering event of an active rule as: e_trigg **after** a_{begin} **if** Q. Splitting the rule in *begin* and *execute* is being borrowed from transaction oriented processing techniques [23]. Observe too that the syntax of $\mathcal{L}_{active}$ enables us to define more complex events than most of the existing active database systems. For example, we can easily define the "sharp increase" event which occurs if the updated value of a particular attribute is more than 10% higher than its current value: $e\_sharp\_increase$ **after** $upd(rel_1(\overline{A},X), rel_1(\overline{A},Y))$ **if** $rel_1(\overline{A},X), X \geq 1.1 * Y$.

(iii) An *active rule* proposition is an expression of the form:

$$\begin{aligned} r(\overline{X_r}) : e_t(\overline{X_t}) &\quad \textbf{consumed}\ (C\_s) \\ &\quad \textbf{initiates}\ [\alpha]\ \textbf{at} \ \ e_a(\overline{X_a}) \\ &\quad \textbf{if} \ \ p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})\ \textbf{at} \ \ e_c(\overline{X_c}) \end{aligned} \qquad (5)$$

where $r(\overline{X_r})$ is a rule identifier, $e_t(\overline{X_t})$, $e_a(\overline{X_a})$, and $e_c(\overline{X_c})$ denote the *triggering event, action-execution event*, and *condition-evaluation event* respectively, α is a sequence of actions, called the action part of the rule, and $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are fluent literals, called the *condition of the rule.* C_s is one of the symbols *no*, *local* or *global*. Variables appearing in $\overline{X_a}$, in any of the input arguments of the actions in α or in any negative literal in the condition must also appear in $\overline{X_t}$ or in a positive literal in the condition. Variables appearing in $\overline{X_r}$ or $\overline{X_c}$ must also appear in $\overline{X_t}$.

Following is the intuitive meaning of each part in the rule:

• The triggering event $e_t(\overline{X_t})$: We assume that the rule is triggered in the very first state in which the event $e_t(\overline{X_t})$ has occurred. Due to the assumption of "persistence" of the events, the rule remains triggered for as long as $e_t(\overline{X_t})$ is in the current set of events. Notice that all the variables in the event are already instantiated during rule processing since they were instantiated by the event definition proposition.

• C_s denotes the *consumption scope* which the *active rule* has over the e_t, with respect to itself and the other active rules with the same triggering event. As we said earlier, it is an element of the set $\{no,\ local,\ global\}$. These modes were introduced in the specification model of [18]. The mode *no* consumption means that neither the condition evaluation nor the action execution of the active rule have any influence on the persistence of the triggering event $e_t(\overline{X_t})$. This mode is typical of production rule systems such OPS5. Once a rule is triggered, each time its condition is true, its action part will be executed. Before explaining the other two options, let us discuss an issue closely associated with the coupling modes – the notion of an active rule being *considered*. A rule may be treated as *considered* as soon as its condition part is to be evaluated regardless of the outcome of the query. Another option is to treat a rule as *considered* only when its action part is to be executed. By default, we assume that the rule is considered at the moment of its condition evaluation.[3] Now, if the consumption scope of the *active rule* is *local*, its consideration will cause that the particular rule is no longer triggered by e_t. However, the other rules which were triggered by e_t remain triggered. On the other hand, if the consumption scope is set to be *global*, then the consideration of a particular active rule consumes the triggering event in such a manner that all the other rules triggered by the same e_t are detriggered.

• α is a sequence of actions.

• The two other events, $e_c(\overline{X_c})$ and $e_a(\overline{X_a})$, denote the events in which the condition evaluation and the execution of the actions take place. The purpose of the explicit specification of the action and condition events is twofold: One, it lets us define the different *coupling modes*. Two, when the mode is deferred, rather than fixing the event for the condition evaluation or the action execution (such as just before commit), the events will appear in the rule. For example, if the designer wants to have the condition of the active rule evaluated immediately when the rule is triggered, than the rule will be of the form:

$$
\begin{aligned}
r(\overline{X_r}) : e_t(\overline{X_t}) \quad & \mathbf{consumed}\ (C\_s) \\
& \mathbf{initiates}\ [\alpha]\ \mathbf{at}\ \ e_t(\overline{X_t}) \\
& \mathbf{if}\ \ p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})\ \mathbf{at}\ \ e_c(\overline{X_c})
\end{aligned}
\tag{6}
$$

where the triggering event and the condition event are the same. For a deferred evaluation of the condition to just before commit and immediate execution of

[3] If needed, we can add an expression of the form **at** e_c or **at** e_a in the first line of (5), to specify that consideration of the rule will occur when the actions begin to execute.

the action part, the rule will have the form:

$$\begin{aligned} r(\overline{X_r}) : e_t(\overline{X_t}) \quad & \textbf{consumed}\,(C\_s) \\ & \textbf{initiates}\,[\alpha]\ \textbf{at}\ \ commit_{begin} \\ & \textbf{if}\ \ p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})\ \textbf{at}\ \ commit_{begin} \end{aligned} \tag{7}$$

Although in this paper we consider only rules that execute inside the same transaction, nothing preempts the use of actions from extended transaction models, such as open nested transaction models [?] to define actions or conditions that are evaluated outside the current transaction.

If all the three events in an active rule are the same, then we allow the omission of e_a and e_c. If two events in an active rule are the same, then instead of repeating we may just refer to the first event.

• The condition part of the active rule is essentially a query posed to the database and it may also contain some evaluable comparison predicates (like "$<$" or "$=$").

2.1 Rule processing points

Before we present examples of active rules we need to introduce the concept of rule processing points. When are rules considered varies from system to system. For example, Postgres considers rules after the execution of each action; Starburst considers rules only a commit time. There are systems, such as Ariel, where the user can group actions in blocks and rule processing happens at the end of each block. In $\mathcal{L}_{active}$ we will have a special action symbol $\upharpoonright$. There are no effect laws associated with $\upharpoonright$, but the occurrence of $\upharpoonright$ in a sequence of actions will indicate a processing point. Thus, if we would like to process rules after each action as in Postgres, we will add an $\upharpoonright$ after each regular action. The $\upharpoonright$ appearing in the action list of a rule, allows the recursive processing of rules.

In the next subsection we demonstrate that much of the existing behaviors described in the literature, which make reference to some pre-events states can be elegantly captured by making a smart usage of the variables and the *event definitions*.

3 Examples from the existing systems

This section presents some examples which demonstrate how our approach can be used to encode different active behaviors from the existing systems. Since [18] presents a very comprehensive comparative study of many active database systems, we borrow the next two examples from it in order to illustrate the relative power of $\mathcal{L}_{active}$. We also accompany them with comments regarding some variations.

Example 3. The rule *rule_pjs* (propagate_joe's_salary) reacts to changes of the salary of a particular employee *joe*[4] in such a manner that it causes the salaries

[4] Note that within the examples we are using PROLOG–like notation (i.e. constants begin with a small letter and variables begin with a capital letter).

of two other employees *sam* and *bob* to have same values as *joe's*. The other rule in the system *rule_iss* (increase_sam's_salary) reacts to changes of *sam's* salary in such a manner that it recursively increases his salary by 10% until it reaches 5000.

The event definitions are:

$e\_change\_js(S_{new})$ **after** $upd(emp(E\#, joe, S_1), emp(E\#, joe, S_{new}))$
e_upd_ss **after** $upd(emp(E\#, sam, S_1), emp(E\#, sam, S_2))$

The action of setting Bob's salary equal to Sam's is specified by the following effect proposition:

upd_bob_sam **causes** $emp(E\#_{bob}, bob, S_{sam})$ **if** $emp(E\#_{sam}, sam, S_{sam})$
upd_bob_sam **causes** $\neg emp(E\#_{bob}, bob, S_{old})$ **if** $emp(E\#_{bob}, bob, S_{old})$

The active rules are specified as follows:

$rule\_pjs : e\_change\_js(S_{new})$ **consumed** (*local*)
initiates $upd(emp(E\#_{sam}, sam, S_1),$
$emp(E\#_{sam}, sam, S_{new})), \uparrow$
$upd\_bob\_sam, \uparrow$

$rule\_iss : e\_upd\_ss$ **consumed** (*local*)
initiates $upd(emp(E\#, sam, S),$
$emp(E\#, sam, 1.1 * S)), \uparrow$
if $S < 5000$

Observe that, since there are processing points after each action, Sam will end up with a salary of 5000 if the update to Joe's salary is less than 5000 and so will Bob, regardless of the Joe's new salary. If we want to have a scenario where Bob would end up having same salary as Joe, regardless of the Sam's new salary, we would need to eliminate, the first processing point (the $\uparrow$ after the $upd(emp(E\#_{sam}, sam, S_1), emp(E\#_{sam}, sam, S_{new})))$. The original encoding achieves *Postgres* – like behavior, while the suggested modification would resemble *Starburst* type of processing.

Example 4. This example illustrates the behavior of the *Starburst* active database system. The active rule is intended to cut the excessive salary increase of the employees. If as the result of the execution of a user–requested transaction the salary of any employee has been increased more than 10% of its pre–transaction value, it is reset to only 10% increase.

First, we need to be careful in encoding the net effect policy of the triggering events:

$e\_upd\_sal(E\_name, S_1, S_3)$
after $upd(emp(E\#, E\_name, S_2), emp(E\#, E\_name, S_3))$
if $e\_upd\_sal(E\_name, S_1, S_2)$

$\neg e\_upd\_sal(E\_name, S_1, S_2)$

$$\textbf{after}\ \ upd(emp(E\#, E\_name, S_2), emp(E\#, E\_name, S_3))$$
$$\textbf{if}\ \ e\_upd\_sal(E\_name, S_1, S_2)$$

Now, the active rule can be encoded as follows:

$rule\_cut\_excess(E\_name, X, Y):$
$e\_upd\_sal(E\_name, X, Y)$ **consumed** $(local)$

$$\textbf{initiates}\ upd(emp(E\#, E\_name, X), emp(E\#, E\_name, 1.1 * X)), \uparrow$$
$$\textbf{at}\ \ commit\_begin$$
$$\textbf{if}\ \ Y > 1.1 * X\ \textbf{at}\ \ commit\_begin$$

Note that the user's transaction may have caused an excessive salary increase to more than one employee. We have captured this with the variables in the definition of the active rule. Namely, in case several employees have received an excessive salary increase, the description would have triggered as many different ground instances of $rule\_cut\_excess$. By carefully specifying the *event definition* propositions we have correctly captured the net effect triggering policy and the proper pre–transaction old value of the employees salary.

The context of the last example in this section is a simplified version of the running example in [42]. Its main purpose is to illustrate how $\mathcal{L}_{active}$ can be used for maintaining integrity constraints.

Example 5. Consider the following relations:
$dept(Dept\#, Dname, Div, Loc)$
$emp(E\#, Ename, JobTitle, Sal, Dept\#)$

where there is a referential integrity constraint (a foreign key) between the relations *dept* and *emp*. Hence, anytime a particular department is deleted from the *dept* relation, all the tuples with employees from that department should be deleted from the *emp* relation.

The triggering event is specified as:

$e\_dept\_del(D\#)$ **after** $del(dept(D\#, Dname, Div, Loc),$

and the active rule is:

$r\_dept\_emp(D\#):$
$e\_dept\_del(D\#)$ **consumed** $(local)$

$$\textbf{initiates}\ del(emp(E\#, Ename, JobTitle, Sal, D\#)), \uparrow$$

Last example illustrates that we could easily encode the example given in [39] (Section 5), however, we'd like to make a subtle observation. The aforementioned example uses the aggregate statement *avg* from SQL. We can demonstrate that the calculation of an average value can be accomplished by using active rules, with undesirable inefficiency. We leave for future work how to formalize the extension of the sets $\mathcal{A}$ and $\mathcal{F}$ to incorporate such actions and fluents.

4 Semantics of $\mathcal{L}_{active}$

For the rest of this section we assume that D is the set of ground instances of the propositions in the domain description under consideration.

We will refer to any set of fluents as a *fluent state* and any set of events as an *event state*. We say that a fluent f holds in a fluent state σ if $f \in \sigma$. $\neg f$ holds in σ if $f \notin \sigma$. Similarly, an event e holds in an event state ε if $e \in \varepsilon$. $\neg e$ holds in ε if $e \notin \varepsilon$.

Let ε be an event state and σ a fluent state. Let τ be a set of (triggered) rules. Let κ be set of (considered) rules (to be fired). We refer to a tuple of the form $< \sigma, \varepsilon, \tau, \kappa >$ as an *active database state* or simply as a state.

The central concept in our semantics are the definition of transition functions called causal interpretations. A *causal interpretation* is a partial function Ψ that maps a (possibly empty) sequence of actions α and a state $< \sigma, \varepsilon, \tau, \kappa >$ into a new state. Given a domain description D, we would like to identify the causal interpretations that model the behavior of D given any initial state. We will do that through four auxiliary functions that will describe how an action, when executed in a state $< \sigma, \varepsilon, \tau, \kappa >$, affects each component of the state. We will also need an action selection function. An *action selection function* S is total function that takes a set of events ε and a set of active rules τ and returns one of the sequences of actions appearing in a rule r_i in τ such that action execution event of r_i is in ε. If such a rule does not exists it returns a special *null* action μ. Each selection function S has an associated function S' that when applied to ε and τ returns a singleton set with the rule that contains the sequence $S(\tau)$ if it is not the *null* action; otherwise is the empty set. Action selection functions will be used to determine which actions will be selected for execution when several action parts of the rules in D are ready to be executed.

We start with the definition of the function that describes the effects of an action a in the fluent state σ, when a is executed in a state $< \sigma, \varepsilon, \tau, \kappa >$. Actually, this function only depends on σ, the other parts of the state are irrelevant. First we need the following definitions. We say that a fluent literal f is an (immediate) *effect* of (executing) a in a fluent state σ if there is a fluent effect law a **causes** f **if** $p_1, \ldots, p_n$ in D whose preconditions $p_1, \ldots, p_n$ hold in σ. Let

$F_a^+(\sigma) = \{f : f \in \mathcal{F}$ and f is an effect of a in $\sigma\}$,

$F_a^-(\sigma) = \{f : f \in \mathcal{F}$ and $\neg f$ is an effect of a in $\sigma\}$ and

$Res_{\mathcal{F}}(a, \sigma) = (\sigma \cup F_a^+(\sigma)) \setminus F_a^-(\sigma)$.

$Res_{\mathcal{F}}$ is referred to as the *fluent transition function*.

The second function defines the changes on the set of events. We say that an event literal e is an (immediate) *effect* of (executing) a in a fluent state σ and event state ε if there is an event effect law e **after** a **if** $e_1, \ldots, e_m, q_1, \ldots, q_n$ in D whose preconditions $q_1, \ldots, q_n$ hold in σ and $e_1, \ldots, e_m$ hold in ε. Let

$E^+_a(\sigma, \varepsilon) = \{e : e \in \mathcal{E}$ and e is an effect of a in $\sigma, \varepsilon\}$,

$E^-_a(\sigma, \varepsilon) = \{e : e \in \mathcal{E}$ and $\neg e$ is an effect of a in $\sigma, \varepsilon\}$.

These two sets identify the events directly generated or removed by a. However, events can also be consumed by triggered rules that are *considered* and whose consumption mode is global. A rule "r : e_t **consumed** (C_s) **initiates** $[\alpha]$ **at** e_a **if** $p_1, \ldots, p_n$ **at** e_c", ground instance of rule of the form (5), is called *positively considered* in a fluent state σ and an event state ε if $e_c \in \varepsilon$ and $p_1, \ldots, p_n$ hold in σ. It is called *negatively considered* if $e_c \in \varepsilon$ and $p_1, \ldots, p_n$ do not hold in σ. r is called *considered* if it is either negatively or positively considered. Thus, the events consumed by considered rules is defined as

$E^-(\sigma, \varepsilon, \tau) = \{e : e \in \mathcal{E}, r \in \tau, C\_s = global$ and r is considered in $\sigma, \varepsilon\}$, and

$Res_{\mathcal{E}}(a, <\sigma, \varepsilon, \tau>) = (\varepsilon \cup E^+_a(\sigma, \varepsilon)) \setminus (E^-_a(\sigma, \varepsilon) \cup E^-(Res_{\mathcal{F}}(a, \sigma), (\varepsilon \cup E^+_a(\sigma, \varepsilon)) \setminus E^-_a(\sigma, \varepsilon), \tau))$.

$Res_{\mathcal{E}}$ is referred to as the *event transition function.*

Note that to change the definition of consideration to time when the condition part of a triggered rule is true we need to change "considered" to "positively considered" in E^-.

The next function describes how the set of triggered rules changes after the execution of a. New rules become triggered by the new events generated by a, and there are three possible reasons to detrigger a rule. 1) The trigger event of the rule is removed from the event state by a, 2) the rule becomes considered and the consumption mode is *local*, 3) The trigger event of the rule is removed from the event state by a rule[5] that becomes considered with consumption mode *global*.

We say that a ground instance of a rule of the form (5): r : e_t **consumed** (C_s) **initiates** $[\alpha]$ **at** e_a **if** $p_1, \ldots, p_n$ **at** e_c, is *triggered* by a set of events ε if $e_t \in \varepsilon$. Let $T^+(\varepsilon)$ be the set of rules triggered by ε in D. For a set of active rules τ, let $T^-(\tau, \varepsilon) = \tau \setminus T^+(\varepsilon)$. Let,

$L^-(\sigma, \varepsilon, \tau) = \{r : r \in \tau, C\_s = local$ and r is considered in $\sigma, \varepsilon\}$ and

$Res_T(a, <\sigma, \varepsilon, \tau>) = (\tau \cup T^+(E^+_a(\sigma, \varepsilon))) \setminus$
$(L^-(Res_{\mathcal{F}}(a, \sigma), (\varepsilon \cup E^+_a(\sigma, \varepsilon)) \setminus E^-_a(\sigma, \varepsilon), \tau) \cup T^-(\tau, (E^-_a(\sigma, \varepsilon) \cup E^-(Res_{\mathcal{F}}(a, \sigma),$
$(\varepsilon \cup E^+_a(\sigma, \varepsilon)) \setminus E^-_a(\sigma, \varepsilon), \tau))))$.

Res_T is referred to as the *triggered rule transition function.*

So far transition functions do not depend on the set of considered rules κ. Additions to κ are the positively considered rules based on the new set of fluent and event states. The deletions depend on the selection function S. We will define the function for κ only for additions, and will specify deletion only after the selection function is introduced. Thus, let

[5] Note that this rule is also being detriggered by this case.

$C^+(\sigma, \varepsilon, \tau) = \{r : r \in \tau$ and r is positively considered in $\sigma, \varepsilon\}$, and

$Res_C(a, <\sigma, \varepsilon, \tau, \kappa>) = \kappa \cup C^+(Res_{\mathcal{F}}(a, \sigma), (\varepsilon \cup E_a^+(\sigma, \varepsilon)) \setminus E_a^-(\sigma, \varepsilon), \tau)$. Res_C is referred to as the *considered rule transition function.*

Definition 1. A causal interpretation Ψ is a model for a domain description D iff for any state $s = <\sigma, \varepsilon, \tau, \kappa>$, there exists an action selection function S such that for any sequence of actions α

1. if $\alpha = []$ then $\Psi(\alpha, s) = s$.
2. if $\alpha = \uparrow \circ \beta$ then
   $\Psi(\alpha, s) = \Psi(\beta, \Psi(S(\varepsilon, \kappa), <\sigma, \varepsilon, \tau, \kappa \setminus S'(\varepsilon, \kappa)>))$.
3. if $\alpha = a \circ \beta$ and $a \neq \uparrow$ then
   $$\begin{aligned}\Psi(\alpha, s) = \Psi(\beta, <&Res_{\mathcal{F}}(a, \sigma),\\ &Res_{\mathcal{E}}(a, <\sigma, \varepsilon, \tau>),\\ &Res_{\mathcal{T}}(a, <\sigma, \varepsilon, \tau>),\\ &Res_C(a, <\sigma, \varepsilon, \tau, \kappa>)>,\end{aligned}$$
   if $F_a^+(\sigma) \cap F_a^-(\sigma) = \emptyset$ and $E_a^+(\sigma, \varepsilon) \cap E_a^-(\sigma, \varepsilon) = \emptyset$. Otherwise is undefined for the state. □

Remark. 1) Currently processing is done only at processing points. This means that if rule that has been selected for execution by S does not have a processing point at the end of the action list, no more triggered rules will be considered for firing until a new processing point is encountered. With minor modifications to the definition of models we could assume that rules by default are processed each time we get into an empty sequence of actions (so that rules will be processed at least at the end of the transaction) in addition to the explicit processing points.
2) There are also systems that have two kinds of rules: rules that are processed each time an action is executed (a la Postgres) and rules that are processed at processing points only (a la Starburst). We can also capture this by typing the rules and modifying Item 3 to process rules of the "Postgres" type after the execution of a but before executing the rest of the actions in α.

The *query language* associated with with $\mathcal{L}_{active}$ consists of hypothetical facts of the form:

$$f \textbf{ after } \alpha \textbf{ at } \sigma \tag{8}$$

where f is a fluent literal, α a sequence of actions, and σ a fluent state. For a query q, we will denote by $\neg q$ the query g **after** α **at** σ if f in the query is $\neg g$. If f is positive, it donetes $\neg f$ **after** α **at** σ.

Definition 2. We say that a query q of the form (8) is true in a model Ψ of an active database description D iff f holds in the fluent state of the state $\Psi(\alpha, <\sigma, \emptyset, \emptyset, \emptyset>)$. □

Definition 3. An active database description D *entails* a query q (written as $D \models q$) iff q is true in all models of D. The set of all facts entailed by D will be denoted by $Cn(D)$.

We will say that the answer given by D to a query q is *yes*, if $D \models q$; *no*, if $D \models \neg q$; and *unknown* otherwise. □

Space limitations do not allow us to present a translation of active database descriptions into logic programs. The translation provides a computational vehicle to implement the entailment relation. Non-determinism is implemented using the choice operator of Zaniolo and the situation calculus is used to characterize the dynamic nature of active databases. The translation will be presented in the full version of the paper. We can also have a more sophisticated query language where we can reason about past situations or hypothetical situations in which we can select what rules have been triggered or considered. Extensions to the query language will be also presented in the full version of the paper.

There are other extensions that can be done to the propositions. One is to allow in the condition of an active rule propositions the occurrence of fluents that refer to previous states of the database. We may use the syntax $p_i(\overline{X_i}, e_i)$, meaning the value of a fluent in the state in which the event e_i was induced (i.e. the state just before e_i was generated [12, 32].) Another extension is the introduction of *instead* options, to override the effects of an action with the execution of a rule. This behavior can be obtained by using defeasible specification of effect propositions similar to the ones described in [6].

4.1 Operational semantics

For a sequence of actions α do:

• While $\alpha \neq []$ do:

Let $\alpha = a \circ \beta$.
After executing a:
(Note: Execution of a automatically updates σ to $\sigma' = \sigma \cup F_a^+(\sigma) \setminus F_a^-(\sigma)$)

If $a \neq \uparrow$ then:

1. Compute $E_a^+(\sigma, \varepsilon)$ and $E_a^-(\sigma, \varepsilon)$.
   Let $\varepsilon = \varepsilon \cup E_a^+(\sigma, \varepsilon) \setminus E_a^-(\sigma, \varepsilon)$.
2. For each active rule r, if its triggering event $e_t \in E_a^+(\sigma, \varepsilon)$ then put r in τ.
3. Let $\tau^- = \emptyset$ and $\epsilon^- = \emptyset$.
4. For each rule r in τ, when its condition event e_c is in ε do:
   (a) If its condition $p_1, \ldots, p_n$ is true in σ' add r to κ.
   (b) If its C_s is local add r to τ^-.
   (c) If its C_s is global add its triggering event e_t to ε^-.
5. Let $\varepsilon = \varepsilon \setminus \varepsilon^-$.
   Let $\tau = \tau \setminus \tau^-$.

6. For every rule r in τ if its triggering event e_t is not in ε remove r from τ.

If $a = \uparrow$ then select a rule r from κ whose action event e_a is in ε.
Remove r from κ, and recursively execute its action part α'.

5 Translation to logic programming

- The initial state

  If f is true in the initial state then ΠD contains the following rule

  $f\_holds(f, s_0) \leftarrow$

  Otherwise ΠD contains the rules:

  $f\_holds(\neg f, s_0) \leftarrow$
- Inertia rule:
  - $f\_holds(F, res(A, S)) \leftarrow A \neq \uparrow, f\_holds(F, S), \mathbf{not}\ ab\_f(F, A, S)$
  - $e\_holds(E, res(A, S)) \leftarrow A \neq \uparrow, e\_holds(E, S), \mathbf{not}\ ab\_e(E, A, S)$
  - $t\_holds(R, res(A, S)) \leftarrow A \neq \uparrow, t\_holds(R, S), \mathbf{not}\ ab\_t(R, A, S)$
  - $c\_holds(R, res(A, S)) \leftarrow A \neq \uparrow, c\_holds(R, S), \mathbf{not}\ ab\_c(R, A, S)$
- Translating effect axioms:

  For each causal law of the form (1.) we have the following rules:

  $f\_holds(f, res(a, S)) \leftarrow a \neq \uparrow, f\_holds(p_1, S), \ldots, f\_holds(p_n, S)$

  $ab\_f(f', a, S) \leftarrow a \neq \uparrow, f\_holds(p_1, S), \ldots, f\_holds(p_n, S)$
- Translating event definitions

  For each event definition of the form (2.) we have the following rule:

  $$e\_caused(e, res(a, S)) \leftarrow a \neq \uparrow, e\_holds(e_1, S), \ldots, e\_holds(e_n, S),$$
  $$f\_holds(q_1, S), \ldots, f\_holds(q_m, S)$$

  In addition we have the following general rules:

  $e\_holds(E, res(A, S)) \leftarrow e\_caused(E, res(A, S)), \mathbf{not}\ ab\_e(\overline{E}, A, S)$

  $ab\_e(E', A, S) \leftarrow e\_caused(E, res(A, S))$

  $ab\_e(E', A, S) \leftarrow e\_removed(E', res(A, S))$

  $e\_true(E, res(A, S)) \leftarrow e\_holds(E, S), \mathbf{not}\ e\_caused(\overline{E}, res(A, S))$

  $e\_true(E, res(A, S)) \leftarrow e\_caused(E, res(A, S)), \mathbf{not}\ e\_caused(\overline{E}, res(A, S))$
- Translating active rules:
  - Action sequence of an active rule:

    For each active rule of the form (5.) we have the following fact:

    $action\_seq(r, \alpha) \leftarrow$
  - Defining noninertial events:

    For each active rule of the form (5.) with C_s as global we have the following rule:

    $e\_removed(e_t, res(A, S)) \leftarrow A \neq \uparrow, t\_holds(r, S), e\_true(e_c, res(A, S))$

- Defining *t_holds*

  For each active rule of the form (5.) we have the following rule:

  $t\_holds(r, res(A, S)) \leftarrow A \neq\uparrow, e\_caused(e_t, res(A, S))$
- Defining *ab_t*
  - $*$ For each active rule of the form (5.) we have the following rule:

    $ab\_t(r, A, S) \leftarrow A \neq\uparrow, e\_holds(\neg e_t, res(A, S))$
  - $*$ For each active rule of the form (5.) with C_s as local we have the following rule:

    $ab\_t(r, A, S) \leftarrow A \neq\uparrow, t\_holds(r, S), e\_true(e_c, res(A, S))$
- Defining *c_holds*

  $$\begin{aligned} c\_holds(r, res(A, S)) \leftarrow\ & A \neq\uparrow, A \neq rem(r), \\ & t\_holds(r, res(A, S)), e\_true(e_c, res(A, S)), \\ & f\_holds(p_1, res(A, S)), \ldots, f\_holds(p_n, res(A, S)) \end{aligned}$$
- Defining *ab_c*

  $ab\_c(R, res(rem(R), S)) \leftarrow$

– Defining *selected*.
  For each active rule of the form (5.)

  $selected(r, S) \leftarrow c\_holds(r, S), e\_holds(e_a, S), not\ ignored(r, S)$

– Defining *ignored*.

  $ignored(R, S) \leftarrow selected(R', S), R \neq R'$

– Defining effect axioms when the action is $\uparrow$
- $f\_holds(F, res(\uparrow, S)) \leftarrow selected(R, S), action\_seq(R, \alpha),$
  $f\_holds(F, [\alpha]res(remove(R), S))$
- $e\_holds(E, res(\uparrow, S)) \leftarrow selected(R, S), action\_seq(R, \alpha),$
  $e\_holds(E, [\alpha]res(remove(R), S))$
- $t\_holds(R', res(\uparrow, S)) \leftarrow selected(R, S), action\_seq(R, \alpha),$
  $t\_holds(R', [\alpha]res(remove(R), S))$
- $c\_holds(R', res(\uparrow, S)) \leftarrow selected(R, S), action\_seq(R, \alpha),$
  $c\_holds(R', [\alpha]res(remove(R), S))$

6 Conclusion and Future Work

We have presented a very simple language for describing active databases with an implementation-independent semantics for characterizing the evolution of the database in response to a sequence of actions. The distinction between actual and hypothetical situations enables the designer to reason about various consequences of actions' execution. This is, in a sense, very similar to the distinction between *when* and *apply* operators for δs [21] which can be used to specify a wide range of execution modes, in a bit less declarative style than $\mathcal{L}_{active}$. A very specific point of view on the semantics of active databases is given in [31]. Based on the relational machines (i.e. Turing machines augmented with relational store [1]) the authors develop a common framework for analyzing the expressive power of some of the existing systems. The tradeoff of such an in depth analysis is that only a limited number of functional features are considered. One of the main differences between our approach and the works on characterizing the active behavior by formal semantics in [41, 37] is the notion of the *event*. An interesting approach of specifying the active database functionality is given in [17]. Based on the event calculus, the authors use the notion of *history* to define event occurrences, database states and actions. The methodology is, in a sense, a "peace making" proposal between the perspectives taken in [41] and [38]. The work presented in [16] concentrates on the ability to reason about implications of event occurrences and interactions among rules, which is similar to what we do. However, there is no clear account for the description language of the active databases. The result in [42] gives a very important insight into the net effect based triggering, refining the behavior of Starburst. Namely, the work makes a distinction between *ephemeral* and *durable* changes caused by a transactions execution in the presence of active rules. Taking only the durable changes into account enables an efficient implementation of the operational semantics and adds additional benefits in automatic determination of priority among the rules and termination. However, there is no explicit notion of an event the way we (or [18]) treat it.

Observe that throughout the paper we were very cautious about the "transactional terminology". The reason is that transaction processing [23] has recently developed many variations of its own, attempting to model different advanced database applications [15]. One of the goals of our future work is to allow for concurrent and complex actions, which will inevitably lead us into using some form of advanced transaction model. This, in turn, may require modifications of the notion of active rule processing itself, and may impose a tradeoff on some of the functional features. Other issues that we plan to address soon are extending our language to incorporate deductive rules and examine the criteria for termination of the active rules.

References

1. S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint logic, relational machines and computational complexity. In *Structure in Complexity Theory*, 1992.
2. C. Baral. Reasoning about actions: non-deterministic effects, constraints and qualification. In *Proc. of IJCAI*, Montreal, 1995.
3. C. Baral and M. Gelfond. Representing concurrent actions in extended logic programs. In Bertram Fronhofer, editor, *Theoretical Approaches to Dynamic Worlds.* (to appear). Preliminary version appeared in IJCAI 93.
4. C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming (to appear)*, 1997.
5. C. Baral and J. Lobo. Formal characterization of active databases. In *International Workshop on Logic in Databases*, 1996.
6. C. Baral and J. Lobo. Formalizing defeasible causality in action theories. In *Proc. of IJCAI*, Japan, August 1997.
7. L. Brownston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison–Wesley, 1985.

8. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *Transactions on Database Systems*, 19(3):367–422, 1994.
9. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Very Large Databases*, 1991.
10. S. Ceri and J. Widom. Deriving incremental production rules for deductive data. *Information Systems*, 19(6):467–490, 1994.
11. S. Chakravarthy, B. Blaustein, A. Buchmannand M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livni, D. McCarthy, R. McKee, and A. Rosenthal. Hipac: A research project in active, time constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, 1989.
12. J. Chomicki and D. Toman. Implementing temporal integrity constraints using an active database. *IEEE Transaction on Knowledge and Data Engineering*, August 1995.
13. U. Dayal, E. Hansen, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*. Addison–Wesley, 1994.
14. K.R. Ditrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of adbms features. In *2nd International Workshop on Rules in Database Systems*, 1995.
15. A. Elmagarmid, editor. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, 1992.
16. O. Etzion. Reasoning about the behaviour of active database applications. In *International Workshop on Rules in Database Systems*, 1995.
17. A.A.A. Fernandez, H. Williams, and N.W. Paton. A logic based integration of active and deductive databases. *New Generation Computing*, 15:205–244, 1997.
18. P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *Transactions on Database Systems*, 20(4):414–471, 1995.
19. N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Very Large Databases*, 1991.
20. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.
21. S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be a first–class citizens in a database programming language. Technical Report USC–CS–94–581, revised 1995.
22. S. Ghandeharizadeh, R. Hull, D. Jacobs, J. Castillo, M.E. Molano, S.H. Lu, J. Luo, C. Tsang, and G. Zhou. On implementing a language for specifying active database execution models. In *Very Large Databases*, 1993.
23. J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Morgan Kaufmann, 1993.
24. E. Hanson and J. Widom. An overview of production rules in database systems. *Knowledge Engineering Review*, 8(2):121–143, 1993.
25. E.N. Hanson. Rule condition testing and action execution in ARIEL. In *ACM SIGMOD*, 1992.
26. G. Kartha and V. Lifshitz. Actions with indirect effects: Preliminary report. In *KR94*, pages 341–350, 1994.
27. F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678.

28. D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *ACM SIGMOD*, 1989.
29. I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic oriented approach. In *Deductive and Object Oriented Databases (DOOD)*, 1995.
30. N.W. Paton, J. Campin, A.A.A. Fernandez, and M. Howard. Formal specification of active database functionality: A survey. In *2nd International Workshop on Rules in Database Systems*, 1995.
31. P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Principles of Database Systems*, 1995. full version 1996.
32. P. Sistla and O. Wolfson. Temporal conditions and integrity constraint checking in active database systems. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM Press, San Jose, CA, 1995.
33. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedure caching and views in database systems. In *ACM SIGMOD*, 1990.
34. M. Stonebraker and G. Kemnitz. The Postgres next–generation database management system. *Communications of ACM*, 34(10):78–92, 1991.
35. J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, 1988.
36. S.D. Urban, A.P. Karadimce, and R.B. Nannapaneni. The implementation and evaluation of integrity maintenance rules in an object-oriented database. In *8th International Conference on Data Engineering*, 1992.
37. J. Widom. A denotational semantics for the Starburst production rule language, 1992. SIGMOD Record 21.
38. J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum? In *International Workshop on Rules in Database Systems*, 1993.
39. J. Widom. The Starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):583–595, 1996.
40. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.
41. C. Zaniolo. A unified semantics for active and deductive databases. In *International Workshop on Rules in Database Systems*, 1993.
42. C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Deductive and Object Oriented Databases (DOOD)*, 1995.

This article was processed using the LaTeX macro package with LLNCS style