A Unifying Semantics for Active Databases Using Non-Markovian Theories of Actions

Iluju Kiringa[†]

Ray Reiter[‡*]

[†]School of Info. Technology and Engineering
University of Ottawa
Ottawa, ON, Canada K1N 6N5
kiringa@site.toronto.edu

[‡]Department of Computer Science,
University of Toronto
Toronto, ON, Canada M5S 3G4
reiter@cs.toronto.edu

Abstract

Over the last fifteen years, database management systems (DBMSs) have been enhanced by the addition of rule-based programming to obtain active DBMSs. One of the greatest challenges in this area is to formally account for all the aspects of active behavior using a uniform formalism. In this paper, we formalize active relational databases within the framework of the situation calculus by uniformly accounting for them using theories embodying non-Markovian control in the situation calculus. We call these theories *active relational theories* and use them to capture the dynamics of active databases. Transaction processing and rule execution is modelled as a theorem proving task using active relational theories as background axioms. We show that major components of an ADBMS may be given a clear semantics using active relational theories.

1 Introduction

Over a large portion of the last thirty years, database management systems (DBMSs) have generally been passive in the sense that only users can perform definition and manipulation operations on stored data. In the last fifteen years, they have been enhanced by the addition of active behavior to obtain active DBMSs (ADBMSs). Here, the system itself performs some definition and manipulation operations automatically, based on a suitable representation of the (re)active behavior of the application domain and the operations performed during a database transaction.

The concept of rule and its execution are essential to ADBMSs. An ADBMS has two major components, a *representational* component called the rule language, and an *executional* component called the execution model. The rule language is used to specify the active behavior of an application. Typical rules used here are the so-called EVENT-CONDITION-ACTION (ECA) rules which are a syntactical construct representing the notion that an action must be performed upon detection of a specified event, provided that some specified condition holds. The execution model is intimately related to the notion of database transaction. A database transaction is a (sometimes nested or detached) sequence of database update operations such as *insert*, *delete*, and *update*, used to insert tuples into, delete them from, or update them in a database. The execution model comes in three flavors: immediate execution, deferred execution, and detached execution, meaning that rules are executed interleaved with database update operations, at the end of user transactions, and in a separate transaction, respectively.

It has been recognized in the literature that giving a formal foundation to active databases is a hard challenge [31, 10, 7, 2]. Different formalisms have been proposed for modelling parts of the concept of rule and its execution (See, e.g., [10, 7, 24]). Very often however, different parts have been formalized using different formalisms, which makes it very difficult to globally reason about active databases.

In this paper, we give a formalization of the concept of active database within the framework of the situation calculus. Building on [18] and [27], this paper aims to account formally for active databases as a further extension of the relational databases formalized as relational theories to accommodate new world knowledge, in this case representational issues associated with rules, the execution semantics, and database transactions.

Including dynamic aspects into database formalization has emerged as an active area of research. In particular, a variety of logic-based formalization attempts have been made [30, 13, 5, 27, 4]. Among these, we find model theoretic approaches (e.g., [30]), as opposed to syntactic approaches (e.g., [13, 27]).

[*] See the memoriam included at the end of this paper.

Proceedings of the 9th International Conference
on Data Base Programming Languages (DBPL),
Potsdam, Germany, September 6-8, 2003

Proposals in [27], and [4] use the language of the situation calculus [21, 28]. This language constitutes a framework for reasoning about actions that relies on an important assumption: the execution preconditions of primitive actions and their effects depend solely on the current situation. This assumption is what the control theoreticians call the Markov property. Thus non-Markovian actions are precluded in the situation calculus used in those proposals. However, in formalizing database transactions, one quickly encounters settings where using non-Markovian actions and fluents is unavoidable. For example, a transaction model may explicitly execute a $Rollback(s)$ to go back to a specific database state s in the past; a *Commit* action is executable only if the previous database states satisfy a set of given integrity constraints and there is no other committing state between the beginning of the transaction and the current state; and an event in an active database is said to have occurred in the current database state if, in some database state in the past, that event had occurred and no action changed its effect meanwhile. Thus one clearly needs to address the issue of non-Markovian actions and fluents explicitly when formalizing database transactions, and researchers using the situation calculus or similar languages to account for updates and active rules fail to do that. Our framework will therefore use non-Markovian control [11].

Though several other similar logics (e.g. dynamic logic, event calculus, and transaction logic) could have been used for our purpose, we prefer the situation calculus, due to a set of features it offers, the most beneficial of which are: the treatment of actions as first class citizens of the logic, thus allowing us to remain within the language to reason about actions; the explicit addressing of the frame problem that inevitably occurs in the context of the database updates; and perhaps most important of all, the relational database log is a first class citizen in the logic. Without a treatment of logs as first class citizens, there seems to be no obvious way of expressing properties about them in the form of logical sentences.

The main contributions of the formalization reported in this paper can succinctly be summarized as follows:

1. We construct logical theories called active relational theories to formalize active databases along the lines set by the framework in [15]; active relational theories are non-Markovian theories in which one may explicitly refer to all past states, and not only to the previous one. They provide the formal semantics of the corresponding active database model. They are an extension of the classical relational theories of [26] to the transaction and active database settings.

2. We give a logical tool that can be used to classify various execution models, and demonstrate its usefulness by classifying execution models of active rules that are executed in the context of flat database transactions.

The remainder of this paper is organized as follows. Section 2 introduces the situation calculus. Section 3 formalizes database transactions as non-Markovian theories of the situation calculus. In Section 4, both the ECA rule paradigm and the main execution models of active rules will be given a logical semantics based on an extension of theories introduced in Section 3. In Section 5, we classify the execution models. Finally, Section 6 briefly reviews related work, concludes the paper and mentions future work.

2 The Situation Calculus

The situation calculus [22], [28] is a many-sorted second order language for axiomatizing dynamic worlds. Its basic ingredients consist of *actions*, *situations* and *fluents*; its universe of discourse is partitioned into sorts for actions, situations, and objects other than actions and situations.

Actions are first order terms consisting of an action function symbol and its arguments. In modelling databases, these correspond to the elementary operations of inserting, deleting and updating relational tuples. For example, in the stock database (Example 1, adapted from [29]) that we shall use below, $price\_insert(stock\_id, price, time, trans)$ denotes the operation of inserting the tuple $(stock\_id, price, time)$ into the database relation $price$ by the transaction $trans$.

A situation is a first order term denoting a sequence of actions. These sequences are represented using a binary function symbol do: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. So $do(\alpha, s)$ is like LISP's $cons(\alpha, s)$, or Prolog's $[\alpha \mid s]$. The special constant S_0 denotes the *initial situation*, namely the empty action sequence, so S_0 is like LISP's ( ) or Prolog's [ ]. In modelling databases, situations will correspond to the database log.[1]

Relations and functions whose values vary from state to state are called *fluents*, and are denoted by predicate or function symbols with last argument a situation term. In Example 1 below, $price(stock\_id, price, time, trans, s)$ is a relational fluent, meaning that in that database state that would be reached by performing the sequence of operations in the log s, $(stock\_id, price, time)$ is a tuple in the $price$ relation, inserted there by the transaction $trans$.

Example 1 *Consider a stock database (adapted from [29]) whose schema has the relations:* $price(stock\_id, price, time, trans, s)$, $stock(stock\_id, price, closingprice, trans, s)$, *and* $customer(cust\_id, balance, stock\_id, trans, s)$, *which are relational fluents. The explanation of the attributes is as follows:* $stock\_id$ *is the identification number of a stock,* $price$ *the current price of a stock,* $time$ *the pricing time,* $closingprice$ *the closing price of the previous day,* $cust\_id$ *the identification number of a customer,* $balance$ *the balance of a customer, and* $trans$ *is a transaction identifier.* □

Notice that, contrary to the presentation in [28], a fluent now contains a further argument – which officially is its

[1] In fact, in the database setting, a situation is a sequence involving many histories corresponding to as many transactions.

second last argument – specifying which transaction contributed to its truth value in a given log. The domain of this new argument could be arbitrarily set to, e.g., integers.

In addition to the function *do*, the language also includes special predicates $Poss$, and $\sqsubset$. $Poss(a(\vec{x}), s)$ means that the action $a(\vec{x})$ is possible in the situation s; and $s \sqsubset s'$ states that the situation s is a prefixing sublog of situation s'. By stating that $s \sqsubset s'$, nothing is said about the possibility of actions that constitute s and s'. For instance,

$$Poss(price\_delete(ST1, \$100, 4PM, 1), S_0)$$

and

$$S_0 \sqsubset do(price\_delete(ST1, \$100, 4PM, 2), S_0)$$

are ground atoms of $Poss$ and $\sqsubset$, respectively.

The set $\mathfrak{W}$ of well formed formulas (wffs) of the situation calculus, together with terms, atomic formulas, and sentences are defined in the standard way of second order languages. Additional logical constants are introduced in the usual way.

In [27], it is shown how to formalize a dynamic relational database setting in the situation calculus with axioms that capture change which are: action precondition axioms stating when database updates are possible, successor state axioms stating how change occurs, unique name axioms that state the uniqueness of update names, and axioms describing the initial situation. These axioms constitute a *basic action theory*, in which control over the effect of the actions in the next situation depends solely on the current situation. This was achieved by precluding the use of the predicate $\sqsubset$ in the axioms. We extend these theories to capture active databases by incorporating non-Markovian control. We achieve this by using the predicate $\sqsubset$ in the axioms.

For simplicity, we consider only primitive update operations corresponding to insertion or deletion of tuples into relations. For each such relation $F(\vec{x}, t, s)$, where $\vec{x}$ is a tuple of objects, t is a transaction argument, and s is a situation argument, a *primitive internal action* is a parameterized primitive action of the situation calculus of the form $F\_insert(\vec{x}, t)$ or $F\_delete(\vec{x}, t)$.

We distinguish the primitive internal actions from *primitive external actions*, namely $Begin(t)$, $Commit(t)$, $End(t)$, and $Rollback(t)$, whose meaning will be clear in the sequel of this paper; these are external as they do not specifically affect the content of the database.[2] The argument t is a unique transaction identifier. Finally, the set of fluents is partitioned into two disjoint sets, namely a set of *database fluents* and a set of *system fluents*. Intuitively, the database fluents represent the relations of the database domain, while the system fluents are used to formalize the processing of the domain. Usually, any functional fluent in a relational language will always be a system fluent.

Now, in order to represent relational databases, we need some appropriate restrictions on the situation calculus.

[2] The terminology internal versus *external* action is also used in [20], though with a different meaning.

Definition 1 *A* basic relational language *is a subset of the situation calculus whose alphabet* $\mathfrak{A}$ *is restricted to (1) a finite number of constants, but at least one, (2) a finite number of action functions, (3) a finite number of functional fluents, and (4) a finite number of relational fluents.*

3 Specifying Database Transactions

This section introduces a characterization of flat transactions in terms of theories of the situation calculus. These theories give axioms of flat transaction models that constrain database logs in such a way that these logs satisfy important correctness properties of database transaction, including the so-called ACID properties[12].

Definition 2 (Flat Transaction) *A sequence of database actions is a* flat transaction *iff it is of the form* $[a_1, \ldots, a_n]$, *where the* a_1 *must be* $Begin(t)$, *and* a_n *must be either* $Commit(t)$, *or* $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, *may be any of the primitive actions, except* $Begin(t)$, $Rollback(t)$, *and* $Commit(t)$; *here, the argument* t *is a unique identifier for the atomic transaction.*

Notice that we do not introduce a term of a new sort for transactions, as is the case in [4]; we treat transactions as run-time activities — execution traces — whose design-time counterparts will be ConGOLOG programs introduced in the next chapter. We refer to transactions by their names that are of sort *object*. Notice also that, on this definition, a transaction is a semantical construct which will be denotations of situations of a special kind called legal logs in the next section.

The axiomatization of a dynamic relational database with flat transaction properties comprises the following classes of axioms:

Foundational Axioms. These are constraints imposed on the structure of database logs [25]. They characterize database logs as finite sequences of updates and can be proved to be valid sentences.

Integrity Constraints. These are constraints imposed on the data in the database at a given situation s; their set is denoted by $\mathcal{IC}_e$ for constraints that must be enforced at each update execution, and by $\mathcal{IC}_v$ for those that must be verified at the end of the flat transaction.

Update Precondition Axioms. There is one for each internal action $A(\vec{x}, t)$, with syntactic form

$$\begin{aligned} Poss(A(\vec{x}, t), s) \equiv (\exists t')\Pi_A(\vec{x}, t', s) \wedge \\ IC_e(do(A(\vec{x}, t), s)) \wedge running(t, s). \end{aligned} \tag{1}$$

Here, $\Pi_A(\vec{x}, t, s)$ is a first order formula with free variables among $\vec{x}, t$, and s. Moreover, the formula on the right hand side of (1) is uniform in s.[3] Such an axiom characterizes the precondition of the update A. Intuitively,

[3] A formula $\phi(s)$ is uniform in a situation term s if s is the only situation term that all the fluents occurring in $\phi(s)$ mention as their last argument.

the conjunct $\Pi_A(\vec{x}, t', s)$ is the part of the right hand side of (1) that really gives the precondition for executing the update $A(\vec{x}, t)$, $IC_e(do(A(\vec{x}, t), s))$ says that the integrity constraints $\mathcal{IC}_e$ are enforced in the situation resulting from the execution of $A(\vec{x}, t)$ in s, and $running(t, s)$ says that the transaction t has not terminated yet. Formally, $IC_e(s)$ and $running(t, s)$ are defined as follows:

$$IC_e(s) =_{df} \bigwedge_{IC \in \mathcal{IC}_e} IC(s).$$

$$\begin{aligned} running(t, s) =_{df} & (\exists s').do(Begin(t), s') \sqsubseteq s \wedge \\ & (\forall a, s'')[do(Begin(t), s') \sqsubset do(a, s'') \sqsubset s \supset \\ & \qquad a \neq Rollback(t) \wedge a \neq End(t)]. \end{aligned}$$

In the stock example, the following states the condition for deleting a tuple from the $customer$ relation:

$$\begin{aligned} & Poss(customer\_delete(cid, bal, sid, t), s) \equiv \\ & \quad (\exists t')customer(cid, bal, sid, t', s) \wedge \\ & \quad IC_e(do(customer\_delete(cid, bal, sid, t), s)) \wedge \\ & \quad running(t, s). \end{aligned} \tag{2}$$

Successor State Axioms. These have the syntactic form

$$\begin{aligned} & F(\vec{x}, t, do(a, s)) \equiv \\ & \quad (\exists \vec{t_1})\Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & \quad (\exists t'')a = Rollback(t'') \wedge \\ & \qquad restoreBeginPoint(F, \vec{x}, t'', s), \end{aligned} \tag{3}$$

There is one such axiom for each database relational fluent F. The formula on the right hand side of (3) is uniform in s, and $\Phi_F(\vec{x}, a, \vec{t}, s)$ is a formula with free variables among $\vec{x}, a, \vec{t}, s$; $\Phi_F(\vec{x}, a, \vec{t}, s)$ specifies how changes occur with respect to internal actions and has the canonical form

$$\gamma^+_F(\vec{x}, a, \vec{t}, s) \vee F(\vec{x}, s) \wedge \neg\gamma^-_F(\vec{x}, a, \vec{t}, s), \tag{4}$$

where $\gamma^+_F(\vec{x}, a, \vec{t}, s)$ ($\gamma^-_F(\vec{x}, a, \vec{t}, s)$) denotes a first order formula specifying the conditions that make a fluent F true (false) in the situation following the execution of a [28].

Formally, $restoreBeginPoint(F, \vec{x}, t, s)$ is defined as follows:

Abbreviation 1

$$\begin{aligned} & restoreBeginPoint(F, \vec{x}, t, s) =_{df} \\ & \quad \{(\exists a_1, a_2, s', s_1, s_2, t'). \\ & \qquad do(Begin(t), s') \sqsubset do(a_2, s_2) \sqsubset do(a_1, s_1) \sqsubseteq s \wedge \\ & \qquad\qquad writes(a_1, F, \vec{x}, t) \wedge writes(a_2, F, \vec{x}, t') \wedge \\ & \qquad [(\forall a'', s'').do(a_2, s_2) \sqsubset do(a'', s'') \sqsubset do(a_1, s_1) \supset \\ & \qquad\qquad \neg writes(a'', F, \vec{x}, t)] \wedge \\ & \qquad [(\forall a'', s'').do(a_1, s_1) \sqsubseteq do(a'', s'') \sqsubset s \supset \\ & \qquad \neg(\exists t'')writes(a'', F, \vec{x}, t'')] \wedge (\exists t'')F(\vec{x}, t'', s_1)]\} \vee \\ & \quad \{(\forall a^*, s^*, s').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \supset \\ & \qquad\qquad \neg writes(a^*, F, \vec{x}, t)] \wedge (\exists t')F(\vec{x}, t', s)\}. \end{aligned}$$

Notice that system fluents have successor state axioms that do not necessarily have the form (3). Intuitively, $restoreBeginPoint(F, \vec{x}, t, s)$ means that the system restores the value of the database fluent F with arguments $\vec{x}$ in a particular way that captures the semantics of $Rollback$:

- The first disjunct in Abbreviation 1 captures the scenario where the transactions t and t' running in parallel, and writing into and reading from F are such that t overwrites whatever t' writes before it (t) rolls back. Suppose that t and t' are such that t begins, and eventually writes into F before rolling back; t' begins after t has begun, writes into F before the last write action of t, and commits before t rolls back. Now the second disjunct in 1 says that the value of F must be set to the "before image" [3] of the first $w(t)$, that is, the value the F had just before the first $w(t)$ was executed.
- The second disjunct in Abbreviation 1 captures the case where the value F had at the beginning of the transaction that rolls back is kept.

Given the actual situation s, the successor state axiom characterizes the truth values of the fluent F in the next situation $do(a, s)$ in terms of all the past situations. Notice that Abbreviation 1 captures the intuition that $Rollback(t)$ affects all tuples within a table. As an example, the following gives a successor state axiom for the fluent $customer(cid, bal, stid, tr, s)$.

$$\begin{aligned} & customer(cid, bal, stid, t, do(a, s)) \equiv \\ & \quad ((\exists t_1)a = customer\_insert(cid, bal, stid, t_1) \vee \\ & \qquad (\exists t_2)customer(cid, bal, stid, t_2, s) \wedge \\ & \qquad \neg(\exists t_3)a = customer\_delete(cid, bal, stid, t_3)) \wedge \\ & \quad \neg(\exists t')a = Rollback(t') \vee \\ & \quad (\exists t').a = Rollback(t') \wedge \\ & \qquad restoreBeginPoint(customer, (cid, bal, stid), t', s). \end{aligned}$$

In this successor state axiom, the formula

$$\begin{aligned} & (\exists t_1)a = customer\_insert(cid, bal, stid, t_1) \vee \\ & \quad (\exists t_2)customer(cid, bal, stid, t_2, s) \wedge \\ & \qquad \neg(\exists t_3)a = customer\_delete(cid, bal, stid, t_3) \end{aligned}$$

corresponds to the canonical form (4).

Precondition Axioms for External Actions. This is a set of action precondition axioms for the transaction specific actions $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$. The external actions of flat transactions have the following precondition axioms:[4]

$$Poss(Begin(t), s) \equiv \neg(\exists s')do(Begin(t), s') \sqsubseteq s, \tag{5}$$

$$Poss(End(t), s) \equiv running(t, s), \tag{6}$$

[4]It must be noted that, in reality, a part of rolling back and committing lies with the user and another part lies with the system. So, we could in fact have something like $Rollback_{sys}(t)$ and $Commit_{sys}(t)$ on the one hand, and $Rollback_{usr}(t)$ and $Commit_{usr}(t)$ on the other hand. However, the discussion is simplified by considering only the system's role in executing these actions.

$$Poss(Commit(t), s) \equiv (\exists s').s = do(End(t), s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s], \tag{7}$$

$$Poss(Rollback(t), s) \equiv (\exists s')[s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee (\exists t', s'')[r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubseteq s]. \tag{8}$$

Notice that our axioms (5)–(8) assume that the user will only use internal actions $Begin(t)$ and $End(t)$ and the system will execute either $Commit(t)$ or $Rollback(t)$. Intuitively, the predicates $sc\_dep(t, t', s)$ and $r\_dep(t, t', s)$ means that transaction t is is strong commit dependent on transaction t', and that transaction t is is rollback dependent on transaction t', respectively.[5]

Dependency axioms. These are the following transaction model-dependent axioms:

$$r\_dep(t, t', s) \equiv transConflict(t, t', s), \tag{9}$$

$$sc\_dep(t, t', s) \equiv readsFrom(t, t', s). \tag{10}$$

The defined predicates $r\_dep(t, t', s)$, $sc\_dep(t, t', s)$ are called dependency predicates. The first axiom says that a transaction conflicting with another transaction generates a rollback dependency, and the second says that a transaction reading from another transaction generates a strong commit dependency.

Unique Names Axioms. These state that the primitive updates and the objects of the domain are pairwise unequal.

Initial Database. This is a set of first order sentences specifying the initial database state. They are completion axioms of the form

$$(\forall \vec{x}, t).F(\vec{x}, t, S_0) \equiv \vec{x} = \vec{C}^{(1)} \vee \ldots \vee \vec{x} = \vec{C}^{(r)}, \tag{11}$$

one for each (database or system) fluent F. Here, the $\vec{C}^i$ are tuples of constants. Also, $\mathcal{D}_{S_0}$ includes unique name axioms for constants of the database. Axioms of the form (11) say that our theories accommodate a complete initial database state, which is commonly the case in relational databases as unveiled in [26]. This requirement is made to keep the theory simple and to reflect the standard practise in databases. It has the theoretical advantage of simplifying the establishment of logical entailments in the initial database; moreover, it has the practical advantage of facilitating rapid prototyping of the ATMs using Prolog which embodies negation by failure, a notion close to the completion axioms used here.

Definition 3 (Basic Relational Theory) *Suppose* $\mathfrak{R} = (\mathfrak{A}, \mathfrak{W})$ *is a basic relational language. Then a theory* $\mathcal{D} \subseteq \mathfrak{W}$ *that comprises the axioms of the form described above is a* basic relational theory.

[5] A transaction t is rollback depend on transaction t' iff, whenever t' rolls back in a log, then t must also roll back in that log; t is strong commit depend on t' iff, whenever t' commits in s, then t must also commit in s.

4 Formalizing Active Databases

4.1 GOLOG

GOLOG [18] is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus according to Section 2. It has the standard – and some not so standard – control structures found in most Algol-like languages: *Sequence* ($[\alpha\,;\,\beta]$; Do action α, followed by action β); *Test actions* ($p?$; Test the truth value of expression p in the current situation); *Nondeterministic action choice* ($\alpha \mid \beta$; Do α or β); *Nondeterministic choice of arguments* ($(\pi\ x)\alpha$; nondeterministically pick a value for x, and for that value of x, do action α); *Conditionals* and *while* loops; and *Procedures*, including recursion.

In [8], GOLOG has been enhanced with parallelism to obtain ConGOLOG.

With the ultimate goal of capturing active databases as theories extending basic relational theories, it is appropriate to adopt an operational semantics of GOLOG programs based on a single-step execution of these programs; such a semantics is introduced in [8]. First, two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that program δ may perform one step in situation s, ending up in situation s', where program δ' remains to be executed. $Final(\delta, s)$ means that program δ may terminate in situation s. A single step here is either a primitive or a testing action. Then the two predicates are characterized by appropriate axioms.

Program execution is captured by using the abbreviation $Do(\delta, s, s')$ [28]. Single-stepping a given program δ means executing the first primitive action of δ, leaving a remaining fragment δ' of δ to be executed. $Do(\delta, s, s')$ intuitively means that, beginning in a given situation s, program δ is single-stepped until the ultimate remainder of program δ may terminate in situation s'.

Formally, we have [8]:

$$Do(\delta, s, s') =_{df} (\exists \delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'),$$

where $Trans^*$ denotes the transitive closure of $Trans$.

In the sequel, we shall develop a syntax for capturing the concept of ECA rule in our framework. After that, we shall model some of the various execution semantics of ECA rules using the interaction of the execution semantics of transaction programs written in (Con)GOLOG — via the Do predicate — with the relational theories developed so far for database transactions.

4.2 Rule Language

An *ECA rule* is a construct of the following form:

$$< t : R : \tau : \zeta(\vec{x}) \longrightarrow \alpha(\vec{y}) > . \tag{12}$$

In this construct, t specifies the transaction that fires the rule, τ specifies events that trigger the rule, and R is a constant giving the rule's identification number (or name). A rule is triggered if the event specified in its event part occurred since the beginning of the open transaction in which that event part is evaluated. Events are one of the predicates $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$, or a combination thereof using logical connectives. The ζ part specifies the rule's condition; it mentions predicates $F\_inserted(r,\vec{x},t,s)$ and $F\_deleted(r,\vec{x},t,s)$ called *transition fluents*, which denote the transition tables [29] corresponding to insertions into and deletions from the relation F. In (12), arguments t, R, and s are suppressed from all the fluents; the first two ones are restored when (12) is translated to a GOLOG program, and s is restored at run time. Finally, α gives a GOLOG program which will be executed upon the triggering of the rule once the specified condition holds. Actions also may mention transition fluents. Notice that $\vec{x}$ are free variables mentioned by ζ and contain all the free variables $\vec{y}$ mentioned by α.

Example 2 *Consider the following active behavior for Example 1. For each customer, his stock is updated whenever new prices are notified. When current prices are being updated, the closing price is also updated if the current notification is the last of the day; moreover, suitable trade actions are initiated if some conditions become true of the stock prices, under the constraint that balances cannot drop below a certain amount of money. Two rules for our example are shown in Figure 1.* □

To characterize the notion of transition tables and events, we introduce the fluent $considered(r,t,s)$ which intuitively means that the rule r can be considered for execution in situation s with respect to the transaction t. The following gives an abbreviation for $considered(r,t,s)$:

$$considered(r,t,s) =_{df} (\exists t').running(t',s). \tag{13}$$

Intuitively, this means that, as long as t is running, any rule r may be considered for execution. In actual systems this concept is more sophisticated than this scheme.[6]

For each database fluent $F(\vec{x},t,s)$, we introduce the *transition fluents* $F\_inserted(r,\vec{x},t,s)$ and $F\_deleted(r,\vec{x},t,s)$. The following successor state axiom characterizes $F\_inserted(r,\vec{x},t,s)$:

$$\begin{aligned} &F\_inserted(r,\vec{x},t,do(a,s)) \equiv \\ &\quad considered(r,t,s) \wedge a = F\_insert(\vec{x},t)] \vee \\ &\quad F\_inserted(r,\vec{x},t,s) \wedge \neg a = F\_delete(\vec{x},t). \end{aligned} \tag{14}$$

Definition (14) means that a tuple $\vec{x}$ is considered inserted in situation $do(a,s)$ iff the internal action $F\_insert(\vec{x},t)$ was executed in the situation s while the rule r was considered, or it was already inserted and a is not the internal action $F\_delete(\vec{x},t)$. This captures the notion of *net effects* [29] of a sequence of actions. Such net effects are accumulating only changes that really affect the database; in this case, if a record is deleted after being inserted, this amounts to nothing having happened. Further net effect policies can be captured in this axiom. The transition fluent $F\_deleted(r,\vec{x},t,s)$ is characterized in a similar way.

Finally, for each database fluent $F(\vec{x},t,s)$, we introduce the *event fluents* $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$. The event fluent $F\_inserted(r,t,s)$ corresponding to an insertion into the relation F has the following successor state axiom:

$$\begin{aligned} &F\_inserted(r,t,do(a,s)) \equiv \\ &\quad a = F\_insert(\vec{x},t') \wedge considered(r,t,s) \vee \\ &\quad F\_inserted(r,t,s). \end{aligned} \tag{15}$$

The event fluent $F\_deleted(r,t,s)$ corresponding to a deletion from the relation F has a similar successor state axiom.

4.3 Active Relational Theories

An *active relational language* is a relational language extended in the following way: for each $n+2$-ary fluent $F(\vec{x},t,s)$, we introduce two $n+3$-ary transition fluents $F\_inserted(r,\vec{x},t,s)$ and $F\_deleted(r,\vec{x},t,s)$, and two 3-ary event fluents $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$.

Definition 4 (Active Relational Theory for flat transactions) *A theory $\mathcal{D} \subseteq \mathfrak{W}$ is an* active relational theory *iff it is of the form*

$$\mathcal{D} = \mathcal{D}_{brt} \cup \mathcal{D}_{tf} \cup \mathcal{D}_{ef},$$

where

1. *$\mathcal{D}_{brt}$ is a basic relational theory.*
2. *$\mathcal{D}_{tf}$ is the set of axioms for transition fluents.*
3. *$\mathcal{D}_{ef}$ is the set of axioms for event fluents which capture simple events. Complex events are defined by combining the event fluents using $\neg$ and $\wedge$.*

4.4 Execution Models

In this section, we specify the execution models of active databases by assuming that the underlying transaction model is that of flat transactions.

4.4.1 Classification

The three components of the ECA rule — i.e. **E**vent, **C**ondition, and **A**ction — are the main dimensions of the representational component of active behavior. Normally, either an indication is given in the rule language as to how the ECA rules are to be processed, or a given processing model is assumed by default for all the rules of the ADBMS. To ease our presentation, we assume the execution models by default.

An execution model is tightly related to the coupling modes specifying the timing constraints of the evaluation

[6]For example, in Starburst [29], r will be considered in the future course of actions only from the time point where it last stopped being considered.

Figure 1: Rules for updating stocks and buying shares

$$<trans\ :\ Update\_stocks\ :\ price\_inserted\ :$$
$$(\exists c, time, bal, price')[price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge stock(s\_id, price', clos\_pr)]$$
$$\longrightarrow$$
$$stock\_insert(s\_id, price, clos\_pr) >$$

$$<trans\ :\ Buy\_100shares\ :\ price\_inserted\ :$$
$$(\exists\ new\_price, time, bal, pr, clos\_pr)[price\_inserted(s\_id, new\_price, time) \wedge$$
$$customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge new\_price < 50 \wedge clos\_pr > 70]$$
$$\longrightarrow$$
$$buy(c, s\_id, 100) >$$

---

of the rule's condition and the execution of the rule's action relative to the occurrence of the event that triggers the rule. We consider the following coupling modes: [7]

1. EC coupling modes:

   Immediate: Evaluate C immediately after the ECA rule is triggered.
   Delayed: Evaluate C at some delayed time point, usually after having performed many other database operations since the time point at which the rule has been triggered.

2. CA coupling modes:

   Immediate: Execute A immediately after C has been evaluated.
   Delayed: Execute A at some delayed time point, usually after having performed many other database operations since the time point at which C has been evaluated.

The execution model is also tightly related to the concept of transaction. In fact, the question of determining when to process the different components of an ECA rule is also answered by determining the transactions within which – if any – the C and A components of the ECA rule are evaluated and executed, respectively. In other words, the transaction semantics offer the means for controlling the coupling modes by allowing one the flexibility of processing the rule components in different, well-chosen transactions. In the sequel, the transaction triggering a rule will be called *triggering transaction* and any other transaction launched by the triggering transaction will be called *triggered transaction*. We assume that all database operations are executed within the boundaries of transactions. From this point of view, we obtain the following refinement for the delayed coupling mode:

1. **Delayed** EC coupling mode: Evaluate C at the end of the triggering transaction T, after having performed all the other database operations of T, but before T's terminal action.

2. **Delayed** CA coupling mode: Execute A at the end of the triggering transaction T, after having performed all the other database operations of T and after having evaluated C, but before T's terminal action.

In presence of flat transactions, we also obtain the following refinement of the immediate coupling mode:

1. **Immediate** EC coupling mode: Evaluate C within the triggering transaction immediately after the ECA rule is triggered.

2. **Immediate** CA coupling mode: Execute A within the triggering transaction immediately after evaluating C.

Notice that the semantics of flat transactions rules out the possibility of nested transactions. For example, we can not process C in a flat transaction and then process A in a further flat transaction, since we quickly encounter the necessity of nesting transactions whenever the execution of a rule triggers further rules. Also, we can not have a delayed CA coupling mode such as: Execute A at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, after T's terminal action, and after the evaluation of C. The reason is that, in the absence of nesting of transactions, we will end up with a large set of flat transactions which are independent from each other. This would make it difficult to relate these independent flat transactions as belonging to the processing of a few single rules.

The refinements above yield for each of the EC and CA coupling modes two possibilities: (1) immediate, and (2) delayed. There are exactly 4 combinations of these modes. We will denote these combinations by pairs (i, j) where i and j denote an EC and a CA coupling modes, respectively. For example, $(1, 2)$ is a coupling mode meaning a combination of the immediate EC and delayed CA coupling modes.

[7] Our presentation is slightly more general than the original one in [14], in which the relationships between coupling modes and execution models, and those between transactions and execution models were not conceptually separated.

Moreover, we will call the pairs (i, j) interchangeably coupling modes or execution models. The context will be clear enough to determine what we are writing about. However, we have to consider these combinations with respect to the constraint that we always execute A strictly after C is evaluated.[8] The following combinations satisfy this constraint: $(1, 1)$, $(1, 2)$, and $(2, 2)$; the combination $(2, 1)$, on the contrary, does not satisfy the constraint.

4.4.2 Immediate Execution Model

Here, we specify the execution model $(1, 1)$. This can be formulated as: **Evaluate C immediately after the ECA rule is triggered and execute A immediately after evaluating C within the triggering transaction.**

Suppose we have a set $\mathcal{R}$ of n ECA rules of the form (12). Then the following GOLOG procedure captures the immediate execution model $(1, 1)$:

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \; ; \; \zeta_1(\vec{x}_1)[R_1, t]? \; ; \; \alpha_1(\vec{y}_1)[R_1, t]] | \\
\qquad\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \; ; \; \zeta_n(\vec{x}_n)[R_n, t]? \; ; \; \alpha_n(\vec{y}_n)[R_n, t]] | \\
\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \\
\qquad\qquad \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \; ? \\
\textbf{endProc} \; .
\end{array} \tag{16}
$$

The notation $\tau[r, t]$ means the result of restoring the arguments r and t to all event fluents mentioned by τ, and $\zeta(\vec{x})[r, t]$ means the result of restoring the arguments r and t to all transition fluents mentioned by ζ. For example, if τ is the complex event

$$price\_inserted \wedge customer\_inserted,$$

then $\tau[r, t]$ is

$$price\_inserted(r, t) \wedge customer\_inserted(r, t).$$

Notice that the procedure (16) above formalizes *how* rules are processed using the immediate model examined here: the procedure $Rules(t)$ nondeterministically selects a rule R_i (hence the use of $|$), tests if an event $\tau_i[R_i, t]$ occurred (hence the use of $?$), in which case it immediately tests whether the condition $\zeta_i(\vec{x}_i)[R_i, t]$ holds (hence the use of ;), at which point the action part $\alpha_i(\vec{y}_i)$ is executed. The last test condition of (16) permits to exit from the rule procedure when none of the rules is triggered.

4.4.3 Delayed Execution Model

Now, we specify the execution model $(2, 2)$ that has both EC and CA coupling being delayed modes. This asks to **evaluate C and execute A at the end of a transaction between the transaction's last action and either its commitment or its failure**. However, notice that the constraint of executing A after C has been evaluated must be enforced.

[8] This constraint is in fact stricter than a similar constraint found in [14], where it is stated that "A cannot be executed before C is evaluated". The formulation of [14], however, does not rule out simultaneous action executions and condition evaluations, a situation that obviously can lead to disastrous behaviors.

Let the interval between the end of a transaction (i.e., the situation $do(End(t), s)$, for some s) and its termination (i.e., the situation $do(Commit(t), s)$ or $do(Rollback(t), s)$, for some s) be called *assertion interval*. We use the fluent $assertionInterval(t, s)$ to capture the notion of assertion interval. The following successor state axiom characterizes this fluent:

$$
\begin{array}{l}
assertionInterval(t, do(a, s)) \equiv a = End(t) \vee \\
\qquad assertionInterval(t, s) \wedge \neg termAct(a, t).
\end{array} \tag{17}
$$

Now, the following GOLOG procedure captures the delayed execution model $(2, 2)$:

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \; ; \\
\quad (\zeta_1(\vec{x}_1)[R_1, t] \wedge assertionInterval(t))? \; ; \; \alpha_1(\vec{y}_1)] | \\
\qquad\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \; ; \\
\quad (\zeta_n(\vec{x}_n)[r_n, t] \wedge assertionInterval(t))? \; ; \; \alpha_n(\vec{y}_n)] | \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge \\
\qquad\qquad assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array} \tag{18}
$$

Here, both the C and A components of triggered rules are executed at assertion intervals.

4.4.4 Mixed Execution Model

Here, we specify the execution model $(1, 2)$ that mix both immediate EC and delayed CA coupling modes. This execution model asks to **evaluate C immediately after the ECA rule is triggered and to execute A after evaluating C in the assertion interval.** This model has the semantics

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \; ; \\
\quad \zeta_1(\vec{x}_1)[R_1, t]? \; ; \; assertionInterval(t)? \; ; \; \alpha_1(\vec{y}_1)] | \\
\qquad\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \; ; \\
\quad \zeta_n(\vec{x}_n)[r_n, t]? \; ; \; assertionInterval(t))? \; ; \; \alpha_n(\vec{y}_n)] | \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge \\
\qquad\qquad assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array} \tag{19}
$$

Here, only the A components of triggered rules are executed at assertion intervals.

4.4.5 Abstract Execution of Rule Programs

We "run" a GOLOG program T embodying an active behavior by establishing that

$$\mathcal{D} \models (\exists s)\ Do(T, S_0, s), \tag{20}$$

where S_0 is the initial, empty log, and $\mathcal{D}$ is the active relational theory for flat transactions. This exactly means that we look for some log that is generated by the program T, and pick any instance of s resulting from the proof obtained by establishing this entailment.

5 Classification Theorems

There is a natural question which arises with respect to the different execution models whose semantics have been given above: what (logical) relationship may exist among them? To answer this question, we must develop a (logical) notion of equivalence between two given execution models. Suppose that we are given two programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ corresponding to the execution models (i, j) and (k, l), respectively.

Definition 5 (Database versus system queries) *Suppose Q is a situation calculus query. Then Q is a* database query *iff the only fluents it mentions are database fluents. A* system query *is one that mentions at least one system fluent.*

Establishing an equivalence between the programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ with respect to an active relational theory $\mathcal{D}$ amounts to establishing that, for all database queries $Q(s)$ and transactions t, whenever the answer to $Q(s)$ is "yes" in a situation resulting from the execution of $Rules^{(i,j)}(t)$ in S_0, executing $Rules^{(k,l)}(t)$ in S_0 results in a situation yielding "yes" to $Q(s)$.

Definition 6 (Implication of Execution Models) *Suppose $\mathcal{D}$ is an active relational theory, and let $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ be ConGOLOG programs corresponding to the execution models (i, j) and (k, l), respectively. Moreover, suppose that for all database queries Q, we have*

$$(\forall s, s', s'', t).Do(Rules^{(m,n)}(t), s, s') \wedge \\ Do(Rules^{(m,n)}(t), s, s'') \supset Q[s'] \equiv Q[s''],$$

where (m, n) is (i, j) or (k, l). Then a rule program $Rules^{(i,j)}(t)$ implies another rule program $Rules^{(k,l)}(t)$ ($Rules^{(i,j)}(t) \implies Rules^{(k,l)}(t)$) iff, for every database query Q,

$$(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s'']]\}. \tag{21}$$

Definition 7 (Equivalence of execution models) *Assume the conditions and notations of Definition 6. Then $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ are equivalent ($Rules^{(i,j)}(t) \cong Rules^{(k,l)}(t)$) iff, for every database query Q,*

$$(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \equiv \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s'']]\}.$$

We restrict our attention to database queries since we are interested in the final state of the content of the database, regardless of the final values of the system fluents.

Theorem 1 *Assume the conditions of Definition 6. Then $Rules^{(2,2)}(t) \implies Rules^{(1,1)}(t)$.*

Proof (Outline). By Definition 6, we must prove that, whenever Q is a database query, we have

$$(\forall t, s).[(\exists s').Do(Rules^{(2,2)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(1,1)}(t), s, s'') \wedge Q[s'']]. \tag{22}$$

Therefore, by the definitions of $Rules^{(1,1)}(t)$, $Rules^{(2,2)}(t)$, $Do/3$, and the semantics of ConGolog given in [8], we must prove:

$$\begin{array}{l}
(\forall t, s). \\
\{(\exists s').Trans^*(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? ;\ (\zeta_1(\vec{x}_1)[R_1, t] \wedge \\
\qquad assertionInterval(t))? ;\ \alpha_1(\vec{y}_1)]| \\
\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? ;\ (\zeta_n(\vec{x}_n)[r_n, t] \wedge \\
\qquad assertionInterval(t))? ;\ \alpha_n(\vec{y}_n)]| \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\quad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge \\
\qquad assertionInterval(t)\}\ ?\},\ s,\ nil,\ s') \wedge Q[s']\} \supset \\
\{(\exists s'').Trans^*(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? ;\ \zeta_1(\vec{x}_1)[R_1, t]? ; \\
\qquad \alpha_1(\vec{y}_1)[R_1, t]]| \\
\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? ;\ \zeta_n(\vec{x}_n)[R_n, t]? ; \\
\qquad \alpha_n(\vec{y}_n)[R_n, t]]| \\
\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\quad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \\
\qquad \zeta_n(\vec{x}_n)[R_n, t])]\ ?\},\ s,\ nil,\ s'') \wedge Q[s'']\}.
\end{array}$$

By the semantics of $Trans$, we may unwind the $Trans^*$ predicate in the antecedent of the big formula above to obtain, in each step, a formula which is a big disjunction of the form

$$(\exists s').[(\phi_1^1 \wedge \phi_2^1 \wedge assertionInterval(t) \wedge \phi_3^1) \vee \cdots \\ \vee (\phi_1^n \wedge \phi_2^n \wedge assertionInterval(t) \wedge \phi_3^n) \vee \\ \Phi] \wedge Q[s'], \tag{23}$$

where ϕ_1^i represents the formula $\tau_i[R_i, t]$, ϕ_2^i represents $\zeta_i(\vec{x}_i)[R_i, t]$, and ϕ_3^i represents the situation calculus formula generated from $\alpha_i(\vec{y}_i)$, with $i = 1, \cdots, n$; Φ represents the formula in the last test action of $Rules^{(2,2)}(t)$. Similarly, we may unwind the $Trans^*$ predicate in the consequent of the big formula on the previous page, second column, to obtain, in each step, a formula which is a big disjunction of the form

$$\begin{array}{l} (\exists s'').[(\phi_1^1 \wedge \phi_2^1 \wedge \phi_3^1) \vee \\ \cdots \qquad \vee (\phi_1^n \wedge \phi_2^n \wedge \phi_3^n) \vee \Phi'] \wedge Q[s''], \end{array} \tag{24}$$

where ϕ_1^i, ϕ_2^i, and ϕ_3^i are to interpret as above, and Φ' represents the formula in the last test action of $Rules^{(1,1)}(t)$. Φ' differs from Φ only through the fact that Φ is a conjuction with one more conjunct which is $assertionInterval(t)$. Also, since no nested transaction is involved, and since both rule programs involved are confluent, we may set $s' = s''$. Therefore, clearly (23) implies (24). ■

Theorem 2 *Assume the conditions of Definition 6. Then* $Rules^{(1,2)}(t) \cong Rules^{(2,2)}(t)$.

Proof. This proof is similar to that of Theorem 1, so we omit it. ■

Corollary 1 *Assume the conditions of Definition 6. Then* $Rules^{(1,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.

Proof. The proof is immediate from Theorems 1 and 2. ■

Contrary to this section, where we assume flat transactions, we could also specify the execution models of active databases by assuming that the underlying transaction model is that of nested transactions [23]. In the latter case, various flavors of immediate and delayed execution models are considered. They all are shown to imply the basic case $Rules^{(1,1)}$. Moreover, all the execution models examined are not equivalent to $Rules^{(1,1)}$ [16].

6 Discussion

Among logic-oriented researchers [31, 19, 9, 1, 2], Baral and Lobo develop a situation calculus-based language to describe actions and their effects, events, and evaluation modes of active rules [2]. In [4], Bertossi *et al.* propose a situation calculus-based formalization that differ considerably from our approach. Their representation of rules forces the action part of their rules to be a primitive database operation. Unlike Bertossi et al., we base our approach on GOLOG, which allows the action component of our rules to be arbitrary GOLOG programs. Moreover, transactions are first-class citizens in their approach.

Thus far, ACTA [6] seems to our knowledge the only framework addressing the problem of specifying transaction models in full first order logic. We complement the approach of ACTA with a formalization of rules and give implementable specifications. The method for such an implementation is given elsewhere [16].

We have used one single logic – the situation calculus — to accounts for virtually all features of rule languages and execution models of ADBMSs. The output of this account is a conceptual model for ADBMSs in the form of active relational theories to be used in conjunction with a theory of complex actions. Thus, considering the software development cycle as depicted in Figure 2, an active relational theory corresponding to an ADBMS constitutes, together with a theory of complex actions a conceptual model for that ADBMS. Since our theories are implementable specifications, implementing the conceptual model provides one with a rapid prototype of the specified ADBMS.

The emerging SQL3 standard [17] contains further dimensions of active behavior that are important and are not mentioned in this paper:

- AFTER/BEFORE/INSTEAD rule activation: SQL3 provides the possibility that a rule be fired before, after, or instead of a certain event.
- Rule activation granularity: rules can be activated for each row or for each statement.
- Interruptability: rule actions may be interruptable in order for other triggered rules to be activated or not.

These issues are dealt with in [16].

Ideas expressed here may be extended in various ways. First, real ADBMSs may be given the same kind of semantics to make them comparable with respect to a uniform framework; [16] addresses this issue. Second, formalizing rules as GOLOG programs can be fruitful in terms of proving formal properties of active rules since such properties can be proved as properties of GOLOG programs. Here, the problems arising classically in the context of active database like confluence and termination [29] are dealt with, and the relationships between the various execution models in terms of their equivalence/nonequivalence are also studied here. This is a reasoning task that will appear in the full version of this paper. Finally, possible rule processing semantics different from existing ones may be studied within our framework.

In Memoriam

Ray Reiter passed away on September 16, 2002 when this work and some others were almost completed. Ray's work has been very influential in foundations of databases and artificial intelligence. To pick just a few of his most important contributions, his concept of Closed World Assumption in database theory, his Default Logic, and his formalization of system diagnosis are well known. His last major contribution was in the area of specifications using the situation calculus [28]. Many in the field considered Ray to be an exceptionally intelligent, and at the same time a very humble human being. The deep intellectual impression that he left and his great achievements will last for years to come. It is hard to imagine that Ray is no longer with us; we miss him very much. Salut Ray!

Figure 2: Relational theories as conceptual models of active database management systems

. . . ⇒ requirements ⇒ analysis ⇒ conceptual model ⇒ design ⇒ prototype ⇒ implementation ⇒ . . .

References

[1] C. Baral and J. Lobo. Formal characterizations of active databases. In *International Workshop on Logic in Databases, LIDS'96*, 1996.

[2] C. Baral, J. Lobo, and G. Trajcevski. Formal characterizations of active databases: Part ii. In *Proceedings of Deductive and Object-Oriented Databases, DOOD'97*, 1997.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, MA, 1987.

[4] L. Bertossi, J. Pinto, and R. Valdivia. Specifying database transactions and active rules in the situation calculus. In H. Levesque and F. Pirri, editors, *Logical Foundations of Cognitive Agents. Contributions in Honor of Ray Reiter*, pages 41–56, New-York, 1999. Springer Verlag.

[5] A. Bonner and M. Kifer. Transaction logic programming. Technical report, University of Toronto, 1992.

[6] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[7] T. Coupaye and C. Collet. Denotational semantics for an active rule execution model. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 36–50. Springer Verlag, 1995.

[8] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, 1997.

[9] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A logic-based integration of active and deductive databases. *New Generation Computing*, 15(2):205–244, 1997.

[10] P. Fraternali and L. Tanca. A structured approach to the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20:414–471, 1995.

[11] A. Gabaldon. Non-markovian control in the situation calculus. In G. Lakemeyer, editor, *Proceedings of the Second International Cognitive Robotics Workshop*, pages 28–33, Berlin, 2000.

[12] J. Gray and Reuter A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[13] A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[14] M. Hsu, R. Ladin, and R. McCarthy. An execution model for active database management systems. In *Proceedings of the third International Conference on Data and Knowledge Bases*, pages 171–179. Morgan Kaufmann, 1988.

[15] I. Kiringa. Towards a theory of advanced transaction models in the situation calculus (extended abstract). In *Proceedings of the VLDB 8th International Workshop on Knowledge Representation Meets Databases (KRDB'01)*, 2001.

[16] I. Kiringa. *Logical Foundations of Active Databases*. PhD thesis, Computer Science, University of Toronto, Toronto, 2003.

[17] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in sql-3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer Verlag, 1999.

[18] H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[19] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proceedings of the Seventh International Conference on Management of Data*, Pune, 1995. Tata and McGraw-Hill.

[20] N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.

[21] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[22] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[23] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing. Information Systems Series*. The MIT Press, Cambridge, MA, 1985.

[24] P. Picouet and V. Vianu. Expressiveness and complexity active databases. In *ICDT'97*, 1997.

[25] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *J. ACM*, 46(3):325–364, 1999.

[26] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 163–189, New-York, 1984. Springer Verlag.

[27] R. Reiter. On specifying database updates. *J. of Logic Programming*, 25:25–91, 1995.

[28] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, 2001.

[29] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[30] M. Winslett. *Updating Logical Databases*. Cambridge University Press, Cambridge, MA, 1990.

[31] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In T.W. Ling and A.O. Mendelzon, editors, *Fourth International Conference on Deductive and Object-Oriented Databases*, pages 55–72, Berlin, 1995. Springer Verlag.