Journal of Intelligent Information Systems manuscript No.
(will be inserted by the editor)

Specifying Active Databases as Non-Markovian Theories of Actions[⋆]

Iluju Kiringa

School of Information Technology and Engineering
Faculty of Engineering, University of Ottawa, Ottawa, Canada
kiringa@site.uottawa.ca

Received: / Revised version:

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 the major components of an ADBMS, namely the rule sets and the execution models, may be given a clear semantics using active relational theories. More precisely: we represent the rule set as a program written in a suitable version of the situation calculus based language ConGolog; then we extend an existing situation calculus based framework for modelling advanced transaction models to one for modelling the execution models of active behaviors.

1 Introduction

1.1 The Problem: A Formal Account of Active Databases

Active databases unify traditional database technology with rule-based programming to express reactive capabilities. Until the late 1980's, traditional database management systems (DBMSs) were passive in the sense that only users or application programs can activate definition and manipulation operations on stored data. An important and useful enhancement of the early 1990's has been the addition of

[⋆] An early version of this paper appeared in [26,24] active behavior to them 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 a rule language, and an *executional* component called an 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 best explained in conjunction with 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 the execution of rules is interleaved with the execution of database update operations, is done at the end of transactions, and is done in a separate transaction, respectively. An active database is essentially made of an underlying (relational) database, plus the two aforementioned components, which we will call them the *active behavior* [37] of the corresponding active database.

There is a large body of literature on active databases. In particular, most researchers in the area have worked on prototypes; several active database systems have been developed [15,54,38,37].

It has been recognized in the literature that the question of giving a formal foundation to active databases is particularly difficult [57,38,15,5,11,5]. Different formalisms have been proposed for modelling parts of the concept of rule and its execution (See, e.g., [38,15,11]).

In this paper, we give a formalization of the concept of active database within the framework of the situation calculus. Building on [27] and [46], 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 [55,19,9,46,8]. Among these, we find model theoretic approaches (e.g. [55]), as opposed to syntactic approaches taken in [19,46].

Proposals in [46], and [8] use the language of the situation calculus [33,47]. 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 [16].

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 (i.e., the problem of accounting for anything that does not change in the domain [35]) 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.

1.2 Contributions

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

1. We extend a general theory of database transactions developed in [25] to account for active databases. In doing so, we construct logical theories called *active relational theories* (ARTs) to formalize active databases along the lines set by the framework in [25]; active relational theories are non-Markovian theories in which one may explicitly refer to all past database states, and not only to the previous one. In addition to the building blocks of the basic relational theories corresponding to various transaction models, the new theories we introduce contain building blocks that are specific to active behaviors. The new building blocks include an event logic, a fragment of the situation calculus used to logically capture and specify event algebras used to express complex events. The ARTs provide the formal semantics of the corresponding active database model; they are an extension of the classical relational theories of [45] 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.
3. We capture ECA-rules as programs written in ConGolog, a situation calculus based language for reasoning about actions [12]. We show how to use the semantics of these programs in simulating the relational database transactions with the active relational theories corresponding to various transaction models as background axioms.

1.3 Outline

The paper is organized as follows. Section 2 introduces the situation calculus, and the basic relational theories used for formalizing database transactions as non-Markovian theories of this calculus. Section 3 extends the basic relational theories to active relational theories, which are theories of actions used for modelling the representational component of active behaviors. In Section 4, we give the semantics of execution models of active behaviors by compiling ECA-rules into ConGolog programs. Section 5 is devoted to related work. Finally, we conclude in Section 6 where we also discuss some possible future work.

2 Background: Situation Calculus, Relational Theories, and Active Databases

2.1 The Situation Calculus and non-Markovian Theories

The situation calculus [35,47] 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 [54]) that we shall use below, the function $price\_insert(stock\_id, price, time, trans)$ 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*.

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 database 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, whose schema has the following 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. $\square$

In addition to the function do, the language also includes special predicates $Poss$, and $\sqsubset$. $Poss(a(\mathbf{x}), s)$ means that the action $a(\mathbf{x})$ is possible in the situation s; and $s \sqsubset s'$ states that the situation s' is reachable from s by performing some sequence of actions — s is said to be a subhistory of 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. Let us call the given language $\mathcal{L}_{sitcalc}$.

The set $\mathfrak{W}$ of well formed formulas (wffs) of $\mathcal{L}_{sitcalc}$, 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 [46], 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 $F(\mathbf{x}, t, s)$, where $\mathbf{x}$ 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 $F\_insert(\mathbf{x}, t)$ or $F\_delete(\mathbf{x}, t)$.

We distinguish the primitive internal actions from *primitive external actions* which are $Begin(t)$, $Commit(t)$, $End(t)$, 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.[1] The argument t is a unique transaction identifier. Finally, the set of fluents of a relational language 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 $\mathcal{L}_{sitcalc}$.

[1] The terminology internal versus *external* action is also used in [30], though with a different meaning.

Definition 1 *A* basic relational language *is a subset of $\mathcal{L}_{sitcalc}$ 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.*

2.2 Database Transaction Models as non-Markovian Theories of Actions

This section summarizes a characterization of flat transactions in terms of theories of the situation calculus given in [23]. 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.

Definition 2 (Flat Transaction) *A sequence of database actions is a* flat transaction *iff it is one of the following:*

1. *Atomic transaction: $[a_1, \ldots, a_n]$, 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.*
2. *Transaction: $at_1 \bullet \ldots \bullet at_m$, where the at_i, $1 \leq i \leq m$, are atomic transactions.*[2]

Notice that we do not introduce a term of a new sort for transactions, as is the case in [8]; we treat transactions as run-time activities — execution traces — whose design-time counterparts will be ConGolog programs introduced later in this paper. 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 [42]. 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 $A(\mathbf{x}, t)$, with syntactic form

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

[2] Given two atomic transactions $A = [A_1, \cdots, A_n]$ and $B = [B_1, \cdots, B_m]$, $A \bullet B$ is an abbreviation for $[A_1, \cdots, A_n, B_1, \cdots, B_m]$.

Here, $\Pi_A(\mathbf{x}, t, s)$ is a first order formula with free variables among $\mathbf{x}, t$, and s. Moreover, the formula on the right hand side of (1) is uniform in s.[3] These axioms characterize the preconditions of the update A; $IC_e(s)$ and $running(t, s)$ are defined as follows:

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

In the stock example, the following states the condition under which a tuple may be deleted from the $customer$ relation:

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

Successor State Axioms. These have the syntactic form

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

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 $\Phi_F(\mathbf{x}, a, \mathbf{t}, s)$ is a formula with free variables among $\mathbf{x}, a, \mathbf{t}, s$; $\Phi_F(\mathbf{x}, a, \mathbf{t}, s)$ specifies how changes occur with respect to internal actions and has the following canonical form [47]:

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

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

The formal definition of $restoreBeginPoint(F, \mathbf{x}, t, s)$ is as follows:

[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.

Abbreviation 1

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

Notice that system fluents have successor state axioms that have to be specified on a case by case basis and do not necessarily have the form (3). Intuitively, $restoreBeginPoint(F, \mathbf{x}, t, s)$ means that the system restores the value of the database fluent F with arguments $\mathbf{x}$ 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" [7] 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.

The following is a successor state axiom for $customer(cid, bal, stid, tr, s)$.

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

In this successor state axiom, the formula

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

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]

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

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

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

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

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 $Rollback(t)$.

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.[5]

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

[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 simplifi ed by considering only the system's role in executing these actions.

[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

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

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

one for each (database or system) fluent F. Here, the $\mathbf{C}^i$ 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 [45]. This requirement is made to keep the theory simple and to reflect the standard practice in databases. It has the theoretical advantage of simplifying the establishment of logical entailments in the initial database.

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.

The $Rollback(t)$ and $Commit(t)$ actions are coercive actions that must always be executed whenever they are possible; we call them *system actions*:

Abbreviation 2

$$systemAct(a, t) =_{df} a = Commit(t) \vee a = Rollback(t).$$

We define legal database logs by taking these system actions into account as follows:

Abbreviation 3

$$\begin{aligned} legal(s) =_{df} & (\forall a, s^*)[do(a, s^*) \sqsubseteq s \supset Poss(a, s^*)] \wedge \\ & \quad (\forall a', a'', s', t)[systemAct(a', t) \wedge responsible(t, a', s') \wedge \\ & \quad\quad responsible(t, a'', s') \wedge Poss(a', s') \wedge do(a'', s') \sqsubset s \supset a' = a'']. \end{aligned} \tag{12}$$

Here, $responsible(t, a, s)$ intuitively means that each transaction t is responsible for the execution of the action a in the log s. Abbreviation (12) says that a log s is legal iff all the actions mentioned in s are possible, and any system action that is responsible for a possible action must have been executed in the log s.

2.3 Active Databases

An active database is a (relational) database augmented with an active behavior [37]. An ADBMS captures the (re)active behavior of application domains. A system embodies reactive behavior if it offers the possibility of automatic actions in response to relevant happenings called *events*. A reactive behavior involves an association of events with actions that should be performed automatically by the system once these events occur; it also involves a way of detecting the occurrences of events; it finally involves a specification of how the system should perform the actions associated with the events that may have occurred.

To be an ADBMS, a DBMS should support the following [37]:

– a reactive model, also considered as a *knowledge model*, for defining events and associating them with actions; and
– an *execution model* for monitoring events and reacting to detected events.

2.3.1 Knowledge Model The knowledge model is expressed in ECA-rules. An ECA-rule has three parts, each one giving the events, conditions, and actions of the rule. Other terms used synonymously for ECA-rules in ADBMSs are: triggers, monitors, alerters, or production rules. Each of the parts of a rule is specified using a specific language: an event language for events, a condition language for conditions, and an action language for actions.

An event is a “happening of interest” [54] that occurs instantaneously at specific time points; it is to be distinguished from an event occurrence which indicates that the event indeed happened. An event may be primitive or composite, that is, a combination of primitive events or other composite events by means of appropriate operators such as logical junctors, sequences, or temporal qualifications. A rule is *triggered* if its event part matches an event occurrence.

Generally, conditions of ECA-rules are expressed as database predicates such as conditions written in a SQL *where* clause, restricted predicates, database queries, and application-defined conditions like procedures.

Actions of ECA-rules involve a *task* to be performed. They include: *update operations*; transactions commands like *rollback* and *commit*; actions to *inform* users of specific database situations of interest; application procedures that may involve *external* calls; and alternative actions to *do instead* of the actual action associated with the event part of the rule.

2.3.2 Execution Model The execution model specifies the run-time behavior of a set of ECA-rules. Although the details vary from system to system, the general pattern of the execution model is the following: The defined set of rules is monitored by the system for relevant events; then, for any given rule that is triggered by some event and before its action can be executed, its condition is checked, and if true, its action is finally executed. A triggered rule whose condition part is evaluated to $true$ is said to be *activated*.

In [32], the notion of coupling modes is introduced to describe the synchronization of rule triggering, condition evaluation, and action execution; it also describes the relationship between rules and transactions. Generally, the concept of “coupling mode” is now used in the former sense.

There are two kinds of coupling modes: the Event-Condition and the Condition-Action coupling modes. They describe the temporal relationship between triggering events and condition evaluation, and between condition evaluation and action execution, respectively. Two possible modes are: *immediate coupling* and *delayed coupling*. In the former setting, the condition is immediately evaluated upon termination of the triggering events; in the later one, it is delayed until some defined time point. There are two sub-cases of the later setting: *deferred* and *detached*. In deferred mode, conditions are evaluated within the same transaction, whereas in detached mode they are evaluated in a separate transaction.

The treatment that the event triggering a rule may undergo is called *event consumption mode*. Here, two issues that are relevant are the scope and the time of event consumption. The first issue amounts to the question of how far the processed events retain their triggering capabilities, and the second one amounts to when events are consumed. Three scopes of event consumption are possible: *no* consumption (meaning that processed events remain capable of triggering further rules), *local* consumption (meaning that they no longer can trigger the processed rule), and *global* consumption (meaning that they no longer can trigger any other rule). Two kinds of time consumption are possible: either before condition evaluation (rule consideration time), or after condition evaluation (rule execution time).

The *net effect policy* indicates whether and how the net effects of events should be taken into account rather than their individual occurrence. Such net effects are accumulating only changes that really affect the database; sample policies are: (i) if a record is first updated and then deleted, only the deletion is retained; (ii) if a record is first inserted and then updated, an insertion of the updated record is retained; (iii) if a record is updated many times, the composition of all updates is retained as a single update; (iv) finally, if a record is deleted after being inserted, this amounts to nothing having happened.

3 Specifying Knowledge Models

In Section 2.2, we have specified relational database transactions models as basic relational theories, which are non-Markovian theories formulated in a finite fragment of the situation calculus. The present section is devoted to extending the basic relational theories to model the representational component of active behaviors.[6] The new theories we will introduce in this section are called active relational theories. As active databases are intimately related to transactions, a substantial building block of these new theories is made of basic relational theories. As we shall see, an active relational theory precisely encompasses a basic relational theory capturing a specific transaction model and axioms for typical active database fluents that are induced by the original database fluents of the domain.

Events are traditionally described using an event algebra. Virtually every proposed ADBMS brings about a different event algebra. This makes it very difficult to analyze these proposals in a uniform way by spelling out what they may have in common, or how they may differ. Typically, logic might act as a framework for dealing with these issues. This section treats events as (somewhat constrained) formulas of the situation calculus. We provide a framework for devising the semantics of complex events in the situation calculus. Such semantics, formulated as a class of axioms of active relational theories, are used for reasoning about the occurrence and consumption modes of events.

[6] Readers from Knowledge Representation might get confused by the qualification "Knowledge" in the title of this section. It seems that a title like "Specifying Events" or "Specifying Active Relational Theories" would be a more appropriate title of the section. However, due to [37], "Knowledge model" is now prevalent in active databases to denote all aspects of active behavior related to events.

3.1 ECA-Rules

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

$$< t : R : \tau : \zeta(\mathbf{x}) \rightarrow \alpha(\mathbf{y}) > . \tag{13}$$

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 occurrs. The relationship between the event occurrence and the triggering of a rule is dictated by *consumption modes*. In its simplest form, the semantics of event consumption is that 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 $F\_deleted(r, t, s)$, called *event fluents*, or a combination thereof using logical connectives. The ζ part specifies the rule's condition; it mentions predicates $F\_inserted(r, \mathbf{x}, t, s)$ and $F\_deleted(r, \mathbf{x}, t, s)$ called *transition fluents*, which denote the transition tables ([54]) corresponding to insertions into and deletions from the relation F. In (13), arguments t, R, and s are suppressed from all the fluents; the two first ones are restored when (13) is translated to a ConGolog program, and s is restored at run time. Finally, α gives a ConGolog program which will be executed upon the triggering of the rule once the specified condition holds. Actions also may mention transition fluents. Notice that $\mathbf{x}$ are free variables mentioned by ζ and contain all the free variables $\mathbf{y}$ mentioned by α. Details of ConGolog programs will be introduced in the next section.

Example 2 Consider the following active behavior for the stock trading database of 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. ■

3.2 Transition Fluents and Net Effect Policy

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 $considered(r, t, s)$:

$$considered(r, t, s) =_{df} (\exists t').running(t', s). \tag{14}$$

Intuitively, this means that, as long as transaction t is running, any rule r may be considered for execution. In actual systems this concept is more sophisticated than this scheme.[7]

[7] For example, in Starburst [54], r will be considered in the future course of actions only from the time point where it last stopped being considered.

$$\begin{array}{l} <trans\ :\ Update\_stocks\ :\ price\_inserted\ : \\ \quad (\exists c, time, bal, price')[price\_inserted(s\_id, price, time) \wedge \\ \qquad customer(c, bal, s\_id) \wedge stock(s\_id, price', clos\_pr)] \\ \quad \rightarrow \\ \quad stock\_insert(s\_id, price, clos\_pr) > \end{array}$$

$$\begin{array}{l} <trans\ :\ Buy\_100shares\ :\ price\_inserted\ : \\ \quad (\exists\ new\_price, time, bal, pr, clos\_pr)[price\_inserted(s\_id, new\_price, time) \wedge \\ \qquad customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge new\_price < 50 \wedge clos\_pr > 70] \\ \quad \rightarrow \\ \quad buy(c, s\_id, 100) > \end{array}$$

Fig. 1 Rules for updating stocks and buying shares

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

$$\begin{array}{ll} F\_inserted(r, \mathbf{x}, t, do(a, s)) \equiv & \\ \qquad considered(r, t, s) \wedge (\exists t')a = F\_insert(\mathbf{x}, t') \vee & (15) \\ \qquad F\_inserted(r, \mathbf{x}, t, s) \wedge \neg(\exists t'')a = F\_delete(\mathbf{x}, t''). & \\ F\_deleted(r, \mathbf{x}, t, do(a, s)) \equiv & \\ \qquad considered(r, t, s) \wedge (\exists t')a = F\_delete(\mathbf{x}, t') \vee & (16) \\ \qquad F\_deleted(r, \mathbf{x}, t, s) \wedge \neg(\exists t'')a = F\_insert(\mathbf{x}, t''). & \end{array}$$

Axiom (15) means that a tuple $\mathbf{x}$ is considered inserted in situation $do(a, s)$ iff the internal action $F\_insert(\mathbf{x}, t')$ was executed in the situation s while the rule r was considered, or it was already inserted and a is not the internal action $F\_delete(\mathbf{x}, t')$; here, t' is transaction that can be different than t. This captures the notion of *net effects* ([54]) 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.

3.3 Event Logics

3.3.1 Primitive and Complex Event Fluents Events that trigger ECA-rules are generally associated with the data manipulation language of the underlying database. In the situation calculus, for each database fluent $F(\mathbf{x}, t, s)$, we introduce the *primitive event fluents* $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$.

The primitive event fluent $F\_inserted(r,t,s)$ corresponding to an insertion into the relation F has the following successor state axiom:

$$\begin{aligned} &F\_inserted(r,t,do(a,s)) \equiv \\ &\qquad (\exists \mathbf{x},t')a = F\_insert(\mathbf{x},t') \wedge considered(r,t,s) \vee \\ &\qquad F\_inserted(r,t,s). \end{aligned} \tag{17}$$

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

$$\begin{aligned} &F\_deleted(r,t,do(a,s)) \equiv \\ &\qquad (\exists \mathbf{x},t')a = F\_delete(\mathbf{x},t') \wedge considered(r,t,s) \vee \\ &\qquad F\_deleted(r,t,s). \end{aligned} \tag{18}$$

Definition 4 (Primitive Event Occurrence) *A primitive event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r,t,s]$. Here $\mathcal{D}$ is a relational theory incorporating the successsor state axioms for the primitive event fluents.*

So, on this definition, an event occurrence (or, equivalently, event detection) is a situation calculus query. Following [3], we call this an *event query*.

In many ADBMSs, complex events are built from simpler, and ultimately, the primitive ones using some *event algebra* [3,58]. Using logical means, we now specify the semantics of complex events that accounts for the active dimension of consumption mode. This development will ultimately lead to a logic for events, instead of an algebra.

That complex events are built from simpler ones is just one of intuitive assumptions one can make about events. In [58], Zimmer and Unland make five basic assumptions about events, which we adopt in the context of the situation calculus as follows:

– Events are interpreted over a set of situations (logs).
– Primitive events are detected at situations, in the order at which they occurred.
– Complex events are built from primitive ones (components) using logical connectives, and many complex events can independently be built from the same set of simpler ones.
– The situation at which a complex action is considered to have occurred is the situation at which the very last of its components occurs; here, "last" means the ordering of situations mentioned above.
– Many events may occur at the same situation, that is, simultaneously.

In order to build complex events, the usual logical connectives and symbols $\wedge$, $\vee$, $\neg$, $\forall$, as well as the ordering predicate $\sqsubset$. These logical symbols and predicates will be used to introduce complex events in the form of abbreviations. The following fluents are used to express some basic constructs for building complex events: $seq\_ev(r,t,e_1,e_2,s)$, $simult\_ev(r,t,e_1,e_2,s)$, $conj\_ev(r,t,e_1,e_2,s)$, $disj\_ev(r,t,e_1,e_2,s)$, and $neg\_ev(r,t,e,s)$. Table 1 gives the informal semantics of these fluents.

Fluent	Informal semantics
$seq\_ev(r,t,e_1,e_2,s)$	event e_1 occurs before event e_2 in s
$simult\_ev(r,t,e_1,e_2,s)$	events e_1 and e_2 occur simultaneously in s
$conj\_ev(r,t,e_1,e_2,s)$	events e_1 and e_2 occur together in any order in s
$disj\_ev(r,t,e_1,e_2,s)$	either event e_1 or event e_2 occurs in s
$neg\_ev(r,t,e,s)$	event e does not occur in s

Table 1 Informal semantics of basic complex events

In what follows concerning execution semantics, it is appropriate to define what counts as a term or a formula whose rule and transaction arguments have been either suppressed or restored. To this end, we introduce the concepts of *rule id and transaction id suppressed* terms and formulas, and *rule id and transaction id restored* terms and formulas, respectively.

Definition 5 (Rid and Tid-Suppressed Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the rid and tid suppressed-terms (rts-terms) and formulas (rts-formulas) of $\mathfrak{R}$ are inductively given by a procedure similar to Definition 13 whose details we omit here. So we omit these.*

Definition 6 (Rid and Tid-Restored Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the rid and tid restored-terms (rtr-terms) and formulas (rtr-formulas) of $\mathfrak{R}$ are inductively given by a procedure similar to Definition 17. Again, details of such a procedure will be clear later in that definition. Whenever t and ϕ are rts-term and rts-formula, respectively, and r and t are rule and transaction names, respectively, we use the notation $t[r,t]$ and $\phi[r,t]$ to denote the corresponding rtr-term and rtr-formula, respectively.*

With reference to the syntax of an ECA-rule (see (13)), the notation $\tau(\mathbf{x})[r,t]$ means the result of restoring the arguments r and t to all event fluents mentioned by τ, $\zeta(\mathbf{x})[r,t]$ means the result of restoring the arguments r and t to all transition fluents mentioned by ζ, and $\alpha(\mathbf{x})[r,t]$ means the result of restoring the arguments r and t to all transition fluents and passing t to actions mentioned by α. For example, if τ is the complex event

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

then $\tau[r,t]$ is

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

In the absence of consumption modes, the formal situation calculus based-semantics of complex events in terms of simpler ones is as follows:

$$neg\_ev(r,t,e,s) =_{df} (\exists r')\neg e[r',t,s], \tag{19}$$

$$seq\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_2[r',t,s] \wedge (\exists r'',s').s' \sqsubset s \wedge e_1[r'',t,s'], \tag{20}$$

$$simult\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_1[r',t,s] \wedge (\exists r'')e_2[r'',t,s], \tag{21}$$

$$\begin{aligned} conj\_ev(r,t,e_1,e_2,s) =_{df} & (\exists r_1)seq\_ev(r_1,t,e_1,e_2,s)\vee \\ & (\exists r_2)seq\_ev(r_2,t,e_2,e_1,s) \vee (\exists r_3)simult\_ev(r_3,t,e_1,e_2,s), \end{aligned} \tag{22}$$

$$disj\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_1[r',t,s] \vee (\exists r'')e_2[r'',t,s]. \tag{23}$$

Definition 7 (Complex Event Occurrence) *A complex event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r,t,s]$. Here $\mathcal{D}$ is a relational theory incorporating the abbreviations (19–(23) above for the complex event fluents.*

In spirit of [58], the following good language design principles are emphasized with respect to complex events of any logic for events:

- **Minimality**: the logic must provide a very small minimal core of constructs that are such that different constructs express different semantics.
- **Symmetry**: The semantics of the constructs is context free.
- **Orthogonality**: The core language must allow every meaningful complex event to be expressible.

From the basic constructs (19–23) above, the set $\{seq\_ev(r,t,e_1,e_2,s), e_1, \cdots, e_n\}$ is the minimal core from which all the others complex events are built, where the $e_i, i = 1, \cdots, n$, are primitive event fluents. More precisely, all the other complex events can be defined using this set. Any other construct not belonging to that core must satisfy the good language design principles of symmetry and orthogonality listed above.

3.3.2 Event Fluents and Consumption Modes Once we have specified a way of building a complex event e from simpler ones, we still have to specify which occurrences of the component of e must be selected in order for e to occur (*event occurrence selection*), and what to do with those occurrences once they have been used in the occurrence of e (*occurrence consumption*). *Consumption modes* are used to determine the event occurrence selection and consumption of the events.

Presumably, it suffices to assign consumption modes to the minimal core set $\{seq\_ev(r,t,e_1,e_2,s), e_1, \cdots, e_n\}$ of the logic for events.

As for primitive event fluents, occurrence selection is trivial: from axioms (17) and (18) we see clearly that the first occurrence of a primitive event fluent may trigger any considered ECA-rule. From axioms (17) and (18), we also see that a primitive event fluent remains unconsumed for any later considered rule. So this way we achieve a no-consumption scope. To achieve a local consumption scope, we change (17) and (18) respectively to

$$\begin{aligned}
&F\_inserted(r,t,do(a,s)) \equiv \\
&\qquad (\exists \mathbf{x},t').a = F\_insert(\mathbf{x},t') \wedge considered(r,t,s) \vee \\
&\qquad F\_inserted(r,t,s) \wedge \neg(\exists \mathbf{y},t'')(a = F\_insert(\mathbf{y},t') \wedge t' = t''),
\end{aligned} \tag{24}$$

and

$$\begin{aligned}
&F\_deleted(r,t,do(a,s)) \equiv \\
&\qquad (\exists \mathbf{x},t')a = F\_delete(\mathbf{x},t') \wedge considered(r,t,s) \vee \\
&\qquad F\_deleted(r,t,s) \wedge \neg(\exists \mathbf{y},t'')(a = F\_delete(\mathbf{y},t') \wedge t' = t'').
\end{aligned} \tag{25}$$

Finally, we achieve a global consumption scope by changing (17) and (18) respectively to

$$\begin{aligned}
&F\_inserted(r,t,do(a,s)) \equiv \\
&\qquad (\exists \mathbf{x},t')a = F\_insert(\mathbf{x},t') \wedge considered(r,t,s) \vee \\
&\qquad F\_inserted(r,t,s) \wedge \neg(\exists r')(F\_inserted(r',t,s) \wedge r \neq r'),
\end{aligned} \tag{26}$$

and

$$\begin{aligned}
&F\_deleted(r,t,do(a,s)) \equiv \\
&\qquad (\exists \mathbf{x},t')a = F\_delete(\mathbf{x},t') \wedge considered(r,t,s) \vee \\
&\qquad F\_deleted(r,t,s) \wedge \neg(\exists r')(F\_deleted(r',t,s) \wedge r \neq r').
\end{aligned} \tag{27}$$

In general. one imposes a specific consumption mode upon the sequence fluent $seq\_ev(r,t,e_1,e_2,s)$ by defining a conjunct $\Psi_{CM}(t,\mathbf{s})$ such that

$$seq\_ev(r,t,e_1,e_2,s') =_{df} (\exists \mathbf{s})\Psi_{seq}(t,\mathbf{s},s') \wedge \Psi_{CM}(t,\mathbf{s}), \tag{28}$$

where $\Psi_{seq}(t,\mathbf{s},s')$ is a situation calculus formula specifying the semantics of $seq\_ev(r,t,e_1,e_2,s)$ (i.e, the right-hand side of (20)); $\Psi_{CM}(t,\mathbf{s})$ is a situation calculus formula that specifies the consumption mode used.

If $\mathcal{L}$ is a distinguished fragment of the situation calculus such that $\Psi_{CM}(t,\mathbf{s}) \in \mathcal{L}$, then this induces the *consumption mode class* $CM_{\mathcal{L}}$. In general, $\mathcal{L}$ can be any fragment of the situation calculus. However, as we shall see in the sequel of this section, formulas $\Psi_{CM}(t,\mathbf{s})$ used in practice belong to logics $\mathcal{L}$ that enjoy particularly desirable properties (e.g., decidability) with respect to specific problems such as the equivalence of two given complex events ([3]).

To deal with consumption modes for sequences, we introduce further terminology adapted from [58]. Suppose $e = seq\_ev(r,t,e_1,e_2,s)$; then e_1 is called the *initiator* and e_2 the *terminator* of e. A component e' of a sequence e is said to be *consumed* iff it no longer can contribute to the detection of e.

By virtue of the Zimmer-Unland assumptions about events, the sequence denoted by the fluent $seq\_ev(r,t,e_1,e_2,s)$ occurs when its terminator e_2 occurs, provided that its initiator occurred according to a given consumption mode.

Some possible consumption modes for event sequences are (many of these can be found in [58] and [3]):

- **First**: Selects the oldest occurrence of the initiator, after which this occurrence is consumed.
- **Consumed Last**: Selects the most recent occurrence of the initiator, after which this occurrence is consumed.
- **Non-Consumed Last**: Selects the most recent occurrence of the initiator, which remains unconsumed as long as there is no occurrence of the initiator.
- **Cumulative**: Selects all occurrences of the initiator up to the situation where the terminator occurs, after which all these occurrences of the initiator are consumed.
- **FIFO**: Selects the earliest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.
- **LIFO**: Selects the latest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.

Example 3 Suppose we have 18 situations and the following occurrences of events E_1 and E_2:

$$\begin{aligned} E_1 &: S_0, S_1, S_2, S_3, S_4, S_8, S_9, S_{11}, S_{15}, S_{16} \\ E_2 &: S_5, S_6, S_7, S_{10}, S_{12}, S_{13}, S_{14}, S_{17} \end{aligned}$$

The six consumption modes considered in this section can now be illustrated as in Figure 2. An arrow $E_1 \rightarrow E_2$ means that E_1 is selected as initiator when E_2 occurs in order for $seq\_ev(r, t, E_1, E_2, s)$ to occur.

Now we spell out details of these consumption modes.

First. We express this by taking $\Psi_{CM}(t, \mathbf{s})$ in (28) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*]). \tag{29}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned} \mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*])]. \end{aligned} \tag{30}$$

Consumed Last. We express this by taking $\Psi_{CM}(t, \mathbf{s})$ in (28) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{31}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned} \mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]]. \end{aligned} \tag{32}$$

Non-Consumed Last. We express this by taking $\Psi_{CM}(t, \mathbf{s})$ in (28) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]. \tag{33}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned} \mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]]. \end{aligned} \tag{34}$$

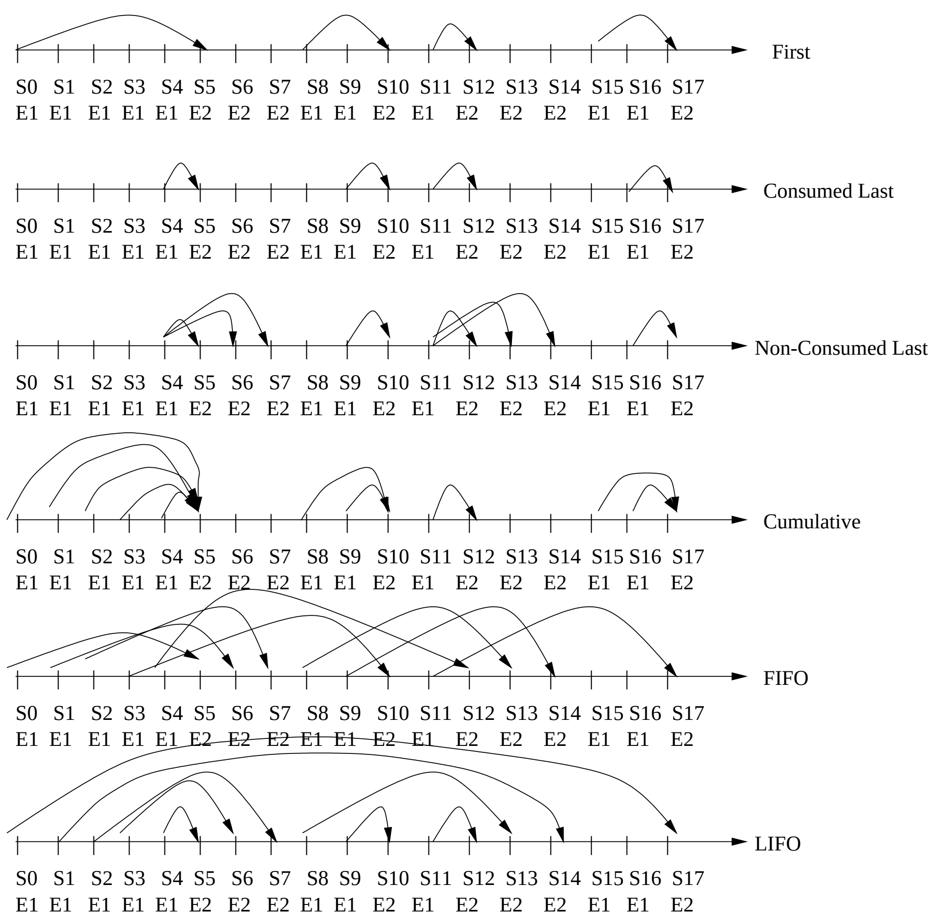
First
Consumed Last
Non-Consumed Last
Cumulative
FIFO
LIFO
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17
E1 E1 E1 E1 E1 E2 E2 E2 E1 E1 E2 E1 E2 E2 E2 E1 E1 E2

Fig. 2 Informal semantics of basic complex events

Cumulative. Here, we take $\Psi_{CM}(t, \mathbf{s})$ in (28) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{35}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]].\end{aligned} \tag{36}$$

FIFO. Here, $\Psi_{CM}(t, \mathbf{s})$ in (28) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \wedge seq\_ev(r, t, e_1, e_2, s^*))] \wedge \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1, t, s^*] \supset \\ &\qquad (\exists s^{**})s^{**} \sqsubset s \wedge seq\_ev(r, t, e_1, e_2, s^{**})]].\end{aligned} \tag{37}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r',t,s] \land (\exists s',r'')[s' \sqsubset s \land e_1[r'',t,s'] \land \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r,t,s^*] \land seq\_ev(r,t,e_1,e_2,s^*))] \land \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1,t,s^*] \supset \\ &\qquad\qquad (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r,t,e_1,e_2,s^{**})]]].\end{aligned} \tag{38}$$

LIFO. Here, $\Psi_{CM}(t,\mathbf{s})$ in (28) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1,t,s^*] \land seq\_ev(r,t,e_1,e_2,s^*))] \land \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1,t,s^*] \supset \\ &\qquad\qquad (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r,t,e_1,e_2,s^{**})]].\end{aligned} \tag{39}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r',t,s] \land (\exists s',r'')[s' \sqsubset s \land e_1[r'',t,s'] \land \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r,t,s^*] \land seq\_ev(r,t,e_1,e_2,s^*))] \land \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1,t,s^*] \supset \\ &\qquad\qquad (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r,t,e_1,e_2,s^{**})]]].\end{aligned} \tag{40}$$

For the purpose of characterizing (some of) the consumption modes, and, later, specifying properties of ECA-rule sets, the set of operators of first order past temporal logic can be introduced using a set of appropriate abbreviations as follows: [2]:

Definition 8 (First Order Past Temporal Logic)

$$\begin{aligned}&previously(\phi,s) =_{df} (\exists s')(\exists a).s = do(a,s') \land \phi(s'), \\ &past(\phi,s) =_{df} (\exists s').S_0 \sqsubseteq s' \sqsubset s \land \phi(s'), \\ &always(\phi,s) =_{df} (\forall s').S_0 \sqsubseteq s' \sqsubset s \supset \phi(s'), \\ &since(\phi,\psi,s) =_{df} (\exists s')[S_0 \sqsubseteq s' \sqsubset s \land \psi(s') \land (\forall s'').s' \sqsubset s'' \sqsubseteq s \supset \phi(s')].\end{aligned}$$

First order past temporal formulas expressed in the situation calculus are formulas that may include the logical connectives $\neg$, $\land$, $\lor$ and $\supset$, quantification over individuals of sort $objects$, and the predicates abbreviated above. In the abbreviations above, ϕ and ψ are first order past temporal formulas.

Now we may characterize (some of) the consumption modes above by stating the following:

Proposition 1 *Each of the First, Consumed-Last, Non-Consumed-Last, and Cumulative consumption modes are expressible in the the past temporal fragment of the situation calculus.*

In the context of the situation calculus, the whole development above leads to the concept of an *event logic* which we now formally express as a definition.

Definition 9 (Event Logic) *An event logic is a triple $(E, C, \mathcal{L})$, where E is a set of event fluents, C is a set of event connectives, together with the predicate $\sqsubset$, and $\mathcal{L}$ is a fragment of the situation calculus specifying the consumption mode associated with event sequences.*

Definition 10 (Implication and Equivalence Problems for an Event Logic) *Suppose $e[r, t, s]$ and $e'[r, t, s]$ are two events of a given event logic $\mathcal{E}$. Then the implication and equivalence problems for $\mathcal{E}$ are the problems of establishing whether, for given R and T, $\mathcal{D} \models (\forall s).e[R, T, s] \supset e'[R, T, s]$, and $\mathcal{D} \models (\forall s).e[R, T, s] \equiv e'[R, T, s]$, respectively. Here $\mathcal{D}$ specifies the semantics of events according to the event logic $\mathcal{E}$.*

Assume the sequence fluents $seq\_ev^F(r, t, e_1, e_2, s)$, $seq\_ev^{CL}(r, t, e_1, e_2, s)$, $seq\_ev^{NL}(r, t, e_1, e_2, s)$, and $seq\_ev^{CUMUL}(r, t, e_1, e_2, s)$ denote event sequence fluents with the consumption modes First, Consumed-Last, Non-Consumed-Last, and Cumulative, whose semantics have been given above. Then we have the following result:

Theorem 1 *Suppose $\mathcal{E} = (E, C, \mathcal{L})$ is the event logic given by:*

– $E \ = \ \{F\_inserted(r, t, s), F\_inserted(r, t, s), seq\_ev^{CM}(r, t, e_1, e_2, s),$
$simult\_ev(r, t, e_1, e_2, s), conj\_ev(r, t, e_1, e_2, s), disj\_ev(r, t, e_1, e_2, s),$
$neg\_ev(r, t, e, s)\}$,
with $CM \in \{F, CL, NL, CUMUL\}$;
– $C \ = \ \{\neg, \wedge, \sqsubset\}$;
– *$\mathcal{L}$ is the past temporal fragment of the situation calculus.*

Then both the implication and the equivalence problems for $\mathcal{E}$ are PSPACE-hard.

3.4 Active Relational Theories

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

Definition 11 (Active Relational Theory for a transaction model M) *Suppose $\mathcal{L} = (\mathfrak{R}, \mathfrak{W})$ is an active relational language. Then a theory $\mathcal{D} \subseteq \mathfrak{W}$ is an* active relational theory *for a transaction model M iff $\mathcal{D}$ 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 for the transaction model M;*
2. *$\mathcal{D}_{tf}$ is the set of axioms for transition fluents;*
3. *$\mathcal{D}_{ef}$ is the set of axioms and definitions for simple and complex event fluents which are expressed in a given event logic.*

Definition 12 *(***Active Relational Database***) An active relational database is a pair $(\mathfrak{R}, \mathcal{D})$, where $\mathfrak{R}$ is an active relational language and $\mathcal{D}$ is an active relational theory.*

Assume the same notations $seq\_ev^F(r,t,e_1,e_2,s)$, $seq\_ev^{CL}(r,t,e_1,e_2,s)$, $seq\_ev^{NL}(r,t,e_1,e_2,s)$, and $seq\_ev^{CUMUL}(r,t,e_1,e_2,s)$ with meanings as in Theorem 1, and suppose $seq\_ev^{FIFO}(r,t,e_1,e_2,s)$ and $seq\_ev^{LIFO}(r,t,e_1,e_2,s)$ denote event sequence fluents with the consumption modes FIFO and LIFO, respectively. Then we have the following result:

Theorem 2 *Suppose $\mathcal{D}$ is an active relational theory with global consumption scope for the primitive event fluents. Then the following equivalences can be established:*

1. *First, Consumed-Last and cumulative consumption modes are equivalent; i.e.,*

   $\mathcal{D} \models seq\_ev^F(r,t,e_1,e_2,s)$ *iff* $\mathcal{D} \models seq\_ev^{CL}(r,t,e_1,e_2,s)$ *iff* $\mathcal{D} \models seq\_ev^{CUMUL}(r,t,e_1,e_2,s)$.
2. *Non-Consumed-Last, LIFO, and FIFO consumption modes are equivalent; i.e.,*

   $\mathcal{D} \models seq\_ev^{NL}(r,t,e_1,e_2,s)$ *iff* $\mathcal{D} \models seq\_ev^{FIFO}(r,t,e_1,e_2,s)$ *iff* $\mathcal{D} \models seq\_ev^{LIFO}(r,t,e_1,e_2,s)$.

It is important to make clear what the equivalences above means. Intuitively, the logical equivalence of two consumption modes M_1 and M_2 amounts to the fact that any given sequence will occur at exactly the same situations under both M_1 and M_2. This ultimately leads to the same active behavior under both M_1 and M_2. Notice that the theorem assume global consumption scope. It still is open whether these equivalences still hold in the case of local consumption scope.

4 Specifying Execution Models

The previous section was devoted to extending basic relational theories to model the reactive models of active behaviors. The new theories introduced there were called active relational theories.

Up to this point, we have uniquely dealt with transactions, informally viewed as execution traces. It is now time to turn our attention to what kind of programs we are supposed to execute in order to get the execution traces we have characterized in Section 2.1, and to how we execute these programs. So we distinguish between transactions which are sequences of database actions and transaction programs which must be executed in order to get those sequences of database actions. In the present section, we specify transaction programs as well formed programs written in ConGolog, a situation calculus based programming language. Such *well formed ConGolog programs* are executed using a special ternary predicate $Do(P,s,s')$ which will serve as an abstract interpreter; $Do(P,s,s')$, introduced in [27], intuitively means: s is a situation reached by executing program P in the situation s'. The predicate Do is defined such that the situations reached by executing well formed ConGolog programs are all legal in the sense of Section 2.1.

We specify a given execution model of active behavior by compiling a set of given ECA-rules into a ConGolog program called *rule program* whose structure is constrained according to that given execution model. Now, the semantics of the predicate $Do(P, s, s')$ is given in a way such that the rule program is implicitly executed whenever a transaction program is executed. It is important to notice that an execution model of the active behavior of transactions is concerned with execution traces, not with programs, as we still are concerned with the situations — thus with sequences of database actions — reached by executing transaction programs.

4.1 Non-Markovian ConGolog

GOLOG, introduced in [27] and enhanced with parallelism in [12] and [13] to yield ConGolog, is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus. It has the following Algol-like control structures:

- nil, the empty program;
- *sequence* ($[\alpha\ ;\ \beta]$; do action α, followed by action β);
- *test actions* ($\phi?$; test the truth value of expression ϕ in the current situation);
- *nondeterministic action choice* ($\alpha \mid \beta$; do α or β);
- *nondeterministic choice of arguments* ($(\pi\ \mathbf{x})\alpha$; nondeterministically pick a value for $\mathbf{x}$, and for that value of $\mathbf{x}$, do action α);
- *conditionals* and *while* loops; and
- *procedures*, including recursion.

The following are ConGolog constructs for expressing parallelism:

- *Concurrency* ($[\alpha\ \parallel\ \beta]$; do α and β in parallel);
- *Concurrent iteration* ($\alpha^{\parallel}$; do α zero or more times in parallel).

The purpose of this section is to show how ConGolog programs are used to capture transaction programs and how the semantics of these programs is used to simulate the transaction models.

4.1.1 Well-formed ConGolog Programs ConGolog syntax is built using constructs that suppress any reference to situations in which test are evaluated. These will be restored at run time by the ConGolog interpreter. The following is a restriction to relational languages of a similar definition given in [42].

Definition 13 (Situation-Suppressed Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the* situation-suppressed *terms (ss-terms) of $\mathfrak{R}$ are given by:*

1. *Any variable or constant of sort $\mathcal{A}$, $\mathcal{O}$, or $\mathcal{S}$ of $\mathfrak{R}$ is an ss-term.*
2. *Whenever F is a functional ss-fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of the appropriate sort, then $F(t_1, \cdots, t_n)$ is an ss-term.*
3. *If a is an action function symbol of $\mathfrak{R}$, and $t_1, \cdots, t_n$ are variables or constants of $\mathfrak{R}$, then $a(t_1, \cdots, t_n)$ is a ss-term.*
4. *For any situation term σ and any action term a, $do(a, \sigma)$ is an ss-term.*

The situation-suppressed formulas (ss-formulas) of $\mathfrak{R}$ are inductively given as follows:

1. *Whenever t, t' are ss-terms of the same sort, then $t = t'$ is an ss-formula. Notice that an ss-formula here, contrary to [42], may mention an equality between terms of sort* situations*.*
2. *Whenever t is an ss-term of sort $\mathcal{A}$, then $Poss(t)$ is an ss-formula.*
3. *Whenever F is an $n+1$-ary relational fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of sort $\mathcal{O}$, then $F(t_1, \cdots, t_n)$ is a ss-formula.*
4. *Whenever P is an m-ary situation independent predicate of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of sort $\mathcal{O}$, then $P(t_1, \cdots, t_n)$ is a ss-formula.*
5. *Whenever t and t' are situation terms of $\mathfrak{R}$, then $t \sqsubset t'$ is an ss-formula.*
6. *Are ϕ and ψ ss-formulas of $\mathfrak{R}$, so are also $\neg\phi$, $\phi \wedge \psi$, and $(\exists x)\phi$ for any variable x.*

Calling situation terms like S_0, $do(A, S_0)$, etc "situation"-suppressed might sound counterintuitive. However, this definition just means that ss-formulas are first order and may still mention situation terms, but never as last argument of fluents; therefore ss-formulas quantify only over those situations that are mentioned in equalities between terms of sort $situations$ and in $\sqsubset$-atoms. For example, the following is an ss-formula:

$$S_0 \sqsubset do(A, (do(B, S_0))) \wedge (\forall x, y, z, w, t)[accounts(x, y, z, w, t) \supset z \geq 0],$$

since the fluent $accounts(x, y, z, w, t, s)$ has its situation argument removed, whereas the following is not:

$$S_0 \sqsubset do(A, (do(B, S_0))) \wedge (\forall x, y, z, w, t, s)[accounts(x, y, z, w, t, s) \supset z \geq 0].$$

Definition 14 (Well formed ConGolog Program for Flat Transactions) *A ConGolog program for flat transactions has the following syntax:*[8]

$$\begin{aligned}\langle prog\rangle \ ::= \ & \langle internal\ action\rangle \mid \langle test\ action\rangle? \mid (\langle prog\rangle ; \langle prog\rangle) \mid (\langle prog\rangle | \langle prog\rangle) \mid \\ & (\langle prog\rangle \parallel \langle prog\rangle) \mid \langle prog\rangle^{\parallel} \mid (\pi x)\langle prog\rangle \mid \langle prog\rangle^{*} \mid \langle procedure\ call\rangle \mid \\ & (\textbf{\textit{proc}}\ P_1(\mathbf{x}_1)\langle prog\rangle\ \textbf{\textit{endProc}}\ ; \cdots ;\ \textbf{\textit{proc}}\ P_n(\mathbf{x}_n)\langle prog\rangle\ \textbf{\textit{endProc}}\ ;\ \langle prog\rangle)\end{aligned}$$

Notice that

1. *$\langle internal\ action\rangle$ is a situation-suppressed internal action term.*
2. *$\langle test\ action\rangle$ is an ss-formula.*
3. *The variable x in $(\pi x)\langle prog\rangle$ must be of sort* actions *or* objects*, never of sort $situations$.*
4. *$\langle procedure\ call\rangle$ is a predicate — a procedure name — of the form $P(t_1, \cdots, t_n)$ where the t_i are ss-terms whose sorts match those of the n arguments in the declaration of P.*

A well formed ConGolog program for flat transactions is syntactically defined as follows:

$$\begin{aligned}\langle wfprog\rangle ::= \ & (\textbf{\textit{proc}}\ P_1(\mathbf{x}_1)\langle prog\rangle\ \textbf{\textit{endProc}}\ ; \cdots ;\ \textbf{\textit{proc}}\ P_n(\mathbf{x}_n)\langle prog\rangle\ \textbf{\textit{endProc}}\ ; \\ & \quad Begin(t); \langle prog\rangle; End(t)) \mid \\ & \langle wfprog\rangle \parallel \langle wfprog\rangle\end{aligned}$$

[8] As in [27], loops and conditionals can be defined in terms of the constructs given here.

4.1.2 Semantics of Well Formed ConGolog Programs With the ultimate goal of handling database transactions, it is appropriate to adopt an operational semantics of well formed ConGolog programs based on a single-step execution of these programs; such a semantics is introduced in ([12]). 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. These axioms contain, for example, the following cases (See [12] and [27] for full details):

$$\begin{aligned}
&Trans(\delta_1;\delta_2, s, \delta, s') \equiv Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s') \vee \\
&\qquad (\exists\gamma).\delta = (\gamma;\delta_2) \wedge Trans(\delta_1, s, \gamma, s'), \\
&Trans(\delta_1|\delta_2, s, \delta, s') \equiv Trans(\delta_1, s, \delta, s') \vee Trans(\delta_2, s, \delta, s')
\end{aligned}$$

to express the semantics of sequences and nondeterministic choice of actions, respectively.

Our axioms for $Trans$ differs from that of [12] with respect to the handling of primitive and test actions:

Definition 15 (Semantics of $Trans$)

$$\begin{aligned}
&Trans(a, s, a', s') \equiv Poss(a, s) \wedge a' = nil \wedge \\
&\quad \{(\exists a'', s'', t)[s'' = do(a, s) \wedge systemAct(a'', t) \wedge \\
&\qquad Poss(a'', s'') \wedge s' = do(a'', s'')] \vee \\
&\quad s' = do(a, s) \wedge [(\forall a'', t) systemAct(a'', t) \supset \neg Poss(a'', s')]\},
\end{aligned} \tag{41}$$

$$Trans(\phi?, s, a', s') \equiv Holds(\phi, s, s') \wedge a' = nil. \tag{42}$$

In the characterization above, we take particularities of system actions into account when processing primitive actions. These actions must occur whenever possible, so the interpreter must test for their possibility upon each performance of a primitive action. The formula (41) captures this requirement; it intuitively means that the primitive action a may legally execute one step in the log s, ending in log s' where a' remains to be executed iff a is possible, the remaining action a' is the empty transaction, and either any possible system action a'' is executed immediately after the primitive action a has been executed and the log s' contains the action a followed by the system action a'', or no system action is possible and the log s' contains only the action a. The formula (42) says that the test action ϕ? may legally be performed in one or more steps in the log s, ending in log s' where a' remains to be executed iff ϕ holds in s, yielding a log s' in a way to be explained below, and a' is an empty program.

Given situation calculus axioms of a domain theory, an execution of a program δ in situation s is the task of finding a situation s' such that there is a final configuration (δ', s'), for some remaining program δ', after performing a couple of transitions from δ, s to δ', s'. Program execution is captured by using the abbreviation $Do(\delta, s, s')$ ([47]). In the single-step semantics, $Do(\delta, s, s')$ intuitively means that program δ is single-stepped until the remainder of program δ may terminate in situation s'; and s' is one of the logs reached by single-stepping the program δ, beginning in a given situation s. Formally, we have ([12]):

$$Do(\delta, s, s') =_{df} (\exists \delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'), \qquad (43)$$

where $Trans^*$ denotes the transitive closure of $Trans$. Finally, a program execution starting in situation S_0 is formally the task of finding a situation s' such that $\mathcal{D} \models Do(\delta, S_0, s')$, where $\mathcal{D}$ is the domain theory.

Definition 16 *The notation $\phi[s]$ denotes the situation calculus formula obtained from a given formula ϕ by restoring the situation argument s in all the fluents (as their last argument) occurring in ϕ.*

The predicate $Holds(\phi, s, s')$ captures the revised Lloyd-Topor transformations of [47]; these are transformations in the style of Lloyd-Topor([28]), but without its auxiliary predicates. The predicate $Holds(\phi, s, s')$ takes a formula ϕ and establish whether it holds in the log s or not. If ϕ is a fluent literal, then the next log s' will be $do(\phi, s)$; if it is a nonfluent literal, then $s' = s$; otherwise revised Lloyd-Topor transformations are performed on ϕ until we reach literals.

To formally define the $Holds(\phi, s, s')$ predicate, we need to define concepts of situation-restored term and situation-restored formula whose semantics are the opposite of those of ss-terms and ss-formulas, respectively.

Definition 17 (Situation-Restored Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the* situation-restored *terms (sr-terms) of $\mathfrak{R}$ are inductively given by:*[9]

1. *Any variable or constant of sort $\mathcal{A}$, $\mathcal{O}$, or $\mathcal{S}$ of $\mathfrak{R}$ is an sr-term.*
2. *Whenever F is a functional ss-fluent of $\mathfrak{R}$, σ is a situation term, and $t_1, \cdots, t_n$ are sr-terms of the appropriate sort, then $F(t_1, \cdots, t_n, \sigma)$ is an sr-term.*
3. *If a is an action function symbol of $\mathfrak{R}$, and $t_1, \cdots, t_n$ are variables or constants of $\mathfrak{R}$, then $a(t_1, \cdots, t_n)$ is a sr-term.*
4. *For any situation term σ and any action term a, $do(a, \sigma)$ is an sr-term.*

The situation-restored *formulas (sr-formulas) of $\mathfrak{R}$ are inductively given as follows:*

1. *Whenever t, t' are sr-terms of the same sort, then $t = t'$ is an sr-formula.*
2. *Whenever t is an sr-term of sort $\mathcal{A}$, then $Poss(t)$ is an sr-formula.*
3. *Whenever F is an $n+1$-ary relational ss-fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are sr-terms of sort $\mathcal{O}$, and σ is a situation term, then $F(t_1, \cdots, t_n, \sigma)$ is a sr-formula.*
4. *Whenever P is an m-ary situation independent predicate of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are sr-terms of sort $\mathcal{O}$, then $P(t_1, \cdots, t_n)$ is a sr-formula.*
5. *Whenever σ and σ' are situation terms of $\mathfrak{R}$, then $\sigma \sqsubset \sigma'$ is an sr-formula.*
6. *Are ϕ and ψ sr-formulas of $\mathfrak{R}$, so are also $\neg\phi$, $\phi \wedge \psi$, and $(\exists x)\phi$ for any variable x.*

[9] Any fluents whose situation is suppressed will be called ss-fluents.

Whenever t and ϕ are a ss-term and a ss-formula, respectively, and σ is a situation term, we use the notation $t[\sigma]$ and $\phi[\sigma]$ to denote the corresponding sr-term and sr-formula, respectively.

Definition 18 (Semantics of $Holds$)

$$
\begin{array}{l}
Holds(\phi, s, s') =_{df} \phi[s] \land s' = do(\phi, s), \textit{ when } \phi \textit{ is a fluent literal},\\
Holds(\phi, s, s') =_{df} \phi[s] \land s' = s, \textit{ when } \phi \textit{ is a nonfluent literal},\\
Holds((\phi_1 \land \phi_2), s, s') =_{df} (\exists s'').Holds(\phi_1, s, s'') \land Holds(\phi_2, s'', s'),\\
Holds((\phi_1 \lor \phi_2)?, s, s') =_{df} Holds(\phi_1, s, s') \lor Holds(\phi_2, s, s'),\\
Holds((\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\phi_1 \lor \phi_2, s, s'),\\
Holds((\phi_1 \equiv \phi_2), s, s') =_{df} Holds((\phi_1 \supset \phi_2) \land (\phi_2 \supset \phi_1), s, s'),\\
Holds((\forall x)\phi, s, s') =_{df} Holds(\neg(\exists x)\neg\phi, s, s'),\\
Holds((\exists x)\phi, s, s') =_{df} Holds(\phi[c/x], s, s'),\\
Holds(\neg\neg\phi, s, s') =_{df} Holds(\phi, s, s'),\\
Holds(\neg(\phi_1 \land \phi_2), s, s') =_{df} Holds(\neg\phi_1, s, s') \lor Holds(\neg\phi_2, s, s'),\\
Holds(\neg(\phi_1 \lor \phi_2), s, s') =_{df} (\exists s'').Holds(\neg\phi_1, s, s'') \land Holds(\neg\phi_2, s'', s'),\\
Holds(\neg(\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\neg(\phi_1 \lor \phi_2), s, s'),\\
Holds(\neg(\phi_1 \equiv \phi_2), s, s') =_{df} Holds(\neg[(\phi_1 \supset \phi_2) \land (\phi_2 \supset \phi_1)], s, s'),\\
Holds(\neg(\forall x)\phi, s, s') =_{df} Holds((\exists x)\neg\phi, s, s'),\\
Holds(\neg(\exists x)\phi, s, s') =_{df} \neg Holds(\phi[c/x], s, s').
\end{array}
$$

The notation $\phi[c/x]$ stands for formula ϕ with a constant c substituted for x. Definition 18 expresses a particular semantics for test actions that is appropriate for handling database transactions. It is important to notice how our test actions are different than those of [12] and why they are needed. Our test actions differ from those of ConGolog ([12]) in two ways. First of all, unlike in ConGolog, ours are genuine actions and not merely tests that may be forgotten as soon as they are executed. We record test actions in the log; i.e. performing a test changes the situation. Second, depending on the syntactic form of the formula in the test, we may end up executing more than just a “single step”. More precisely, more than one single actions are added to the log whenever more than one tests of fluent literals are involved in the formula being tested. This semantics is dictated by the very nature of database transaction models. Here, many test actions correspond to database reading actions. A transaction has no means of remembering which transaction it had read from other than to record reading actions in the log. This cannot be done with the semantics for test action found in [12]. In other words, in the absence of test actions in the log, the semantics of [12] has no straightforward way to express such things as transaction T_1 reads data from transaction T_2.

4.1.3 Simulation of Well Formed ConGolog Programs We use the ConGolog language as a transaction language for specifying and simulating transaction models at the logical level. To simulate a specific transaction model, we first pick the appropriate basic relational theory $\mathcal{D}$ corresponding to that transaction model. Then, we write a well formed ConGolog program T expressing the desired transactional behavior. Simulating the program T amounts to the task of establishing the entailment

$$\mathcal{D} \models (\exists s')\ Do(T, S_0, s'). \tag{44}$$

To establish the entailment (44), we need to accommodate non-Markovian tests. These are tests involving the predicate $\sqsubset$; they allow to test whether a log is a sublog of another log. Henceforth, the regression operator [47] must incorporate a case handling the predicate $\sqsubset$. Such a regression operator is defined in [16]. Details on this are out of the scope of this paper.

Example 4 Consider the Debit/Credit example of Section 2.1. In addition to the axioms given in Section 2.1, we have the following successor state axiom characterizing the system fluent $served(aid, s)$ which is used for synchronization purposes:

$$served(aid, do(a, s)) \equiv report(aid) \vee served(aid, s).$$

The action $report(aid)$, whose precondition axiom is

$$Poss(report(aid), s) \equiv true,$$

is used to make the fluent $served(aid, s)$ true by indicating that a request emitted by the owner of the account aid has been granted. The situation independent predicate $requested(aid, req)$ registers such requests, where req is a positive or negative real number corresponding to a deposit or a withdrawal of that amount of money.

Now we give the following ConGolog procedures which are well-formed and capture the essence of the debit/credit example:

```
**proc** $a\_update(t, aid, amt)$
  $(\pi\ bid, abal, abal', tid)[accounts(aid, bid, abal, tid, t)?\ ;$
    $[abal' = abal + amt]?\ ;$
    $a\_del(aid, bid, abal, tid, t)\ ;$
    $a\_ins(aid, bid, abal', tid, t)]$
**endProc**
**proc** $execDebitCredit(t, bid, tid, aid, amt)$
    $a\_update(aid, amt)\ ;$
    $(\pi\ abal)\ [accounts(aid, bid, abal, tid, t)?\ ;$
              $t\_update(t, tid, amt)\ ; b\_update(t, bid, amt)]$
**endProc**
**proc** $processReq(t, tid, aid, amt)$
  $(\pi\ bid, abal)[accounts(aid, bid, abal, tid, t)?\ ; execDebitCredit(t, bid, tid, aid, amt)]$
**endProc**
```

```
**proc** $processTrans(t)$
  $Begin(t);$
    $[(\pi\ bid, aid, abal, tid, req).$
      $\{accounts(aid, bid, abal, tid, t) \wedge requested(aid, req) \wedge \neg served(aid)\}?\ ;$
      $report(aid)\ ; Spawn(t, aid)\ ;\ processReq(t, tid, aid, req)\ ;\ End(aid)]^{||}\ ;$
    $\neg((\exists\ aid, req).requested(aid, req) \wedge served(aid))?\ ;$
  $End(t)$
**endProc**
```

Similarly to the first procedure, we can give procedures $t\_update(tid, amt)$ and $b\_update(bid, amt)$ for updating teller and branch balances, respectively. ■

The ACID properties are enforced by the interpreter that either commits work done so far or rolls it back whenever the database general ICs are violated. Thus, well formed programs are a specification of transactions with the full scale of a programming language at the logical level. Notice that a formula ϕ in a test ϕ? is in fact an ss-formula whose situation argument is restored at run-time by the interpreter. Notice also the use of the concurrent iteration in the last procedure; this spawns a new child transaction for each account that emitted a request but has not yet been served. For simplicity in this example, we have assumed that each account has at most one request; this allows us to use the account identifiers aid to denote spawn subtransactions.

Now we can simulate the program, say $processTrans(T)$ of Example 4, by performing the theorem proving task of establishing the entailment

$$\mathcal{D} \models (\exists s')\ Do(processTrans(T), S_0, s'),$$

where S_0 is the initial, empty log, and $\mathcal{D}$ is the basic relational theory for nested transactions that comprises the axioms above; this exactly means that we look for some log that is generated by the program T. We are interested in any instance of s resulting from the proof obtained by establishing this entailment. Such an instance is obtained as a side-effect of this proof.

In Definition 15, we take particularities of system actions into account. These actions must occur whenever they are possible, so the interpreter must test for their possibility upon each performance of a primitive action. Definition 15 captures this requirement.

Definition 19 (Universal Possibility Assumption (UPA) for Test Actions) *This is the sentence*

$$(\forall F, \mathbf{x}, t, s)Poss(F(\mathbf{x}, t)[s], s). \tag{45}$$

The UPA allows unrestricted carrying out of test actions which in the database setting are database queries. Using the UPA and Definition 15, we can show that Do generates only legal situations:

Theorem 3 *Suppose $\mathcal{D}$ is a relational theory, and let T be a well formed ConGolog program for flat transactions. Then,*

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset legal(s). \tag{46}$$

4.2 Specifying the Execution Models with Flat Transactions

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

4.2.1 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 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:[10]

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.

[10] They were first introduced in the HiPAC system [21] and have since been widely used in most ADBMS proposals [37]. Our presentation is slightly more general than the original one, in which the relationships between coupling modes and execution models, and those between transactions and execution models were not conceptually separated.

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. 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.[11] 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.

4.2.2 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.**

[11] This constraint is in fact stricter than a similar constraint found in [21], where it is stated that "A cannot be executed before C is evaluated". The formulation of [21], however, does not rule out simultaneous action executions and condition evaluations, a situation that obviously can lead to disastrous behaviors.

Suppose we have a set $\mathcal{R}$ of n ECA-rules of the form (13). Then the following GOLOG procedure captures the immediate execution model $(1,1)$:

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1,t]? \; ; \; \zeta_1(\mathbf{x}_1)[R_1,t]? \; ; \; \alpha_1(\mathbf{y}_1)[R_1,t]]| \\
\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n,t]? \; ; \; \zeta_n(\mathbf{x}_n)[R_n,t]? \; ; \; \alpha_n(\mathbf{y}_n)[R_n,t]]| \\
\quad \neg[(\exists \mathbf{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\mathbf{x})[R_1,t]) \vee \ldots \vee \\
\qquad\qquad (\exists \mathbf{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\mathbf{x}_n)[R_n,t])] \; ? \\
\textbf{endProc} \; .
\end{array}
\tag{47}
$$

Notice that the procedure (47) 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 $\zeta_i(\mathbf{x}_i)[R_i,t]$ holds (hence the use of ;), at which point the action part $\alpha_i(\mathbf{y}_i)$ is executed. The last test condition of (47) permits to exit from the rule procedure when none of the rules is triggered.

4.2.3 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.

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

$$
\begin{array}{l}
assertionInterval(t, do(a,s)) \equiv a = End(t) \vee \\
\qquad\qquad assertionInterval(t,s) \wedge \neg termAct(a,t).
\end{array}
\tag{48}
$$

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

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1,t]? \; ; \; (\zeta_1(\mathbf{x}_1)[R_1,t] \wedge assertionInterval(t))? \; ; \\
\qquad\qquad\qquad\qquad\qquad \alpha_1(\mathbf{y}_1)]| \\
\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n,t]? \; ; \; (\zeta_n(\mathbf{x}_n)[r_n,t] \wedge assertionInterval(t))? \; ; \\
\qquad\qquad\qquad\qquad\qquad \alpha_n(\mathbf{y}_n)]| \\
\quad \neg\{[(\exists \mathbf{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\mathbf{x})[R_1,t]) \vee \ldots \\
\qquad \vee (\exists \mathbf{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\mathbf{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array}
\tag{49}
$$

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

4.2.4 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

```
proc $Rules(t)$
    $(\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1, t]? \, ; \, \zeta_1(\mathbf{x}_1)[R_1, t]? \, ; \, assertionInterval(t)? \, ;$
                                                        $\alpha_1(\mathbf{y}_1)] |$
                        $\vdots$
    $(\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n, t]? \, ; \, \zeta_n(\mathbf{x}_n)[r_n, t]? \, ; \, assertionInterval(t))? \, ;$        (50)
                                                        $\alpha_n(\mathbf{y}_n)] |$
    $\neg\{[(\exists \mathbf{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\mathbf{x})[R_1, t]) \vee \ldots$
        $\vee (\exists \mathbf{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\mathbf{x}_n)[R_n, t])] \wedge assertionInterval(t)\} \, ?$
endProc.
```

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

Example 5 Consider the stock trading database of Example 2, and also the active behavior described there. That is, customer stocks are updated whenever new prices are notified. When a current price of a stock is being updated, its closing price is also updated if the current price notification is the last of the day. Suitable trade actions are initiated whenever some conditions become true of a stock price, under the specific constraint that the customer balance cannot drop below a certain amount. Under the delayed execution model, the two rules shown in Figure 1 can be compiled into the rule program shown in Figure 3. ■

4.3 Semantics of Rule Programs

4.3.1 Abstract Execution of Rule Programs Given the program $Rules(t)$ specified as in (47)–(50), we can now complete the logical characterization of the execution models by showing how the predicate $Trans(\delta, s, \delta', s')$ of [12] must be modified to handle primitive actions:

$$
\begin{aligned}
& Trans(a, s, a', s') \equiv \\
& (\exists a^*, s'', s^*, t).transOf(a, t, s) \wedge Poss(a, s) \wedge a' = nil \wedge \\
& \{[s'' = do(a, s) \wedge systemAct(a^*, t) \wedge Poss(a^*, s'') \wedge s' = do(a^*, s'')] \vee \\
& \quad [s^* = do(a, s) \wedge [(\forall a'', t') systemAct(a'', t') \supset \\
& \qquad \neg Poss(a', s^*) \wedge Do(Rules(t), s^*, s')]]\}.
\end{aligned}
\tag{51}
$$

With the last conjunct, we interleave the execution of each action with the execution of $Rules(t)$. The interpreter picks one of the triggered rules, according to

```
proc $Rules(trans)$
    $(\pi\ c, time, bal, price', s\_id, price, clos\_pr)$
      $[price\_inserted[Update\_stocks, trans]\ ?\ ;$
      $[\{price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge$
           $stock(s\_id, price', clos\_pr)\}[Update\_stocks, trans] \wedge$
           $assertionInterval(trans)]\ ?\ ;$
      $stock\_insert(s\_id, price, clos\_pr)[Update\_stocks, trans]]\ |$
    $(\pi\ new\_price, time, bal, pr, clos\_pr, c, s\_id, 100))$
      $[price\_inserted[Buy\_100shares, trans]\ ?\ ;$
      $[\{price\_inserted(s\_id, new\_price, time) \wedge customer(c, bal, s\_id) \wedge$
           $stock(s\_id, pr, clos\_pr) \wedge new\_price < 50 \wedge$
           $clos\_pr > 70\}[Update\_stocks, trans] \wedge$
           $assertionInterval(trans)]\ ?\ ;$
      $buy(c, s\_id, 100)[Update\_stocks, trans]]\ |$
      $\neg[(\exists c, time, bal, price')(price\_inserted[Update\_stocks, trans] \wedge$
        $\{price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge$
          $stock(s\_id, price', clos\_pr)\}[Update\_stocks, trans])] \vee$
        $(\exists new\_price, time, bal, pr, clos\_pr)(price\_inserted[Buy\_100shares, trans] \wedge$
          $\{customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge$
           $new\_price < 50 \wedge clos\_pr > 70\}[Update\_stocks, trans]) \wedge$
           $assertionInterval(trans)\}\ ?$
endProc.
```

Fig. 3 Rules for updating stocks and buying shares: delayed execution model

(47)–(50), executes it, and comes back at the beginning of $Rules(t)$; it does so until the last test condition of (47)–(50) becomes true; the semantics (47)–(50) will make sure that rule execution follows the appropriate execution model.[12] Finally, $transOf(t, a, s)$ is characterized as follows:

$$transOf(t, a, s) \equiv \bigvee_{A \in \mathcal{A}} (\exists \mathbf{x}) a = A(\mathbf{x}, t). \qquad (52)$$

Here, $\mathcal{A}$ denotes the set of the database actions of the domain.

We execute a GOLOG program T embodying an active behavior by performing the theorem proving task of establishing entailments of the form (44), where $\mathcal{D}$ is now the active relational theory for an appropriate transaction model.

[12] Notice that this semantics means that transitions may in fact be big leaps involving many actions. This may prevent some desirable concurrency. We leave this problem out of the scope of this document.

Using the notion of well formed ConGolog programs introduced in Definition 14, together with the notion of legal database log defined in (12), we can show the following:

Theorem 4 *Suppose $\mathcal{D}$ is an active relational theory for flat transactions, and let T be a well formed ConGolog program for flat transactions. Then,*

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset legal(s), \tag{53}$$

and, more generally,

$$\mathcal{D} \models (\forall s, s').legal(s) \supset [Do(T, s, s') \supset legal(s')]. \tag{54}$$

4.3.2 Classification Theorems for Execution Models There is a natural question which arises with respect to the different execution models whose semantics have been given above: is it possible to reduce the set of execution models described above to a handful of classes by virtue of some equivalence mechanism? 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 20 (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.*

Intuitively, establishing an equivalence between the programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ with respect to a background 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 21 (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*[13]

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

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

$$\begin{aligned}(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s'']]\}.\end{aligned} \tag{55}$$

[13] We will come back to this sentence later in Section 4.5. The sentence expresses the so-called confluence property of active rules.

Definition 22 (Equivalence of execution models) *Assume the conditions and notations of Definition 21. Then the rule programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ are equivalent ($Rules^{(i,j)}(t) \cong Rules^{(k,l)}(t)$) iff, for every database query Q,*

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

It is important to see why we restrict our attention to database queries. We do so since we are interested in the final state of the content of the database, regardless of the final values of the system fluents. Consider the execution models $(1, 1)$ and $(1, 2)$ and suppose that we use the Do predicate to execute a well formed program T assuming the rules in Figure 1. Assume that the classic flat transaction model is used for the transaction program T. Therefore the execution model $(1, 1)$ will involve no system fluent since the flat transaction model does not involve any. On the contrary, the execution model $(1, 2)$ will involve a system fluent if the triggered transaction associated with this execution model involves any system fluent. In general, different execution models are most likely to involve different system fluents so that, from the point of view of these, virtually no two execution models would be equivalent. Fortunately, the content of the database is usually what matters to the user, not the internal state of the system which may be considered as a black box. This justifies restricting our attention to database queries in establishing the relationships among execution models.

Theorem 5 *Assume the conditions of Definition 21. Then the following holds: $Rules^{(2,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.*

Theorem 6 *Assume the conditions of Definition 21. Then the following holds: $Rules^{(1,2)}(t) \cong Rules^{(2,2)}(t)$.*

Corollary 1 *Assume the conditions of Definition 21. Then the following holds: $Rules^{(1,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.*

4.4 Priorities

Rules are assigned priorities by the programmer who provides an explicit (partial) order among rules. Given a set $\mathcal{R} \neq \emptyset$ of rules, this amounts to partitioning the set $\mathcal{R}$ into subsets $\mathcal{R}_i$, $1 \leq i \leq k$, such that rules in $\mathcal{R}_i$ all have equal priority, and rules in $\mathcal{R}_i$ have priority higher than the rules in $\mathcal{R}_j$, for all j such that $i < j$. As an example, we partition the set of rules of Example 2 into two subsets; the first subset contains the rule $Update\_stocks$ and the second one contains $Buy\_100shares$.

Suppose that the set $\mathcal{R}$ of rules is partitioned into subsets $\mathcal{R}_i = \{r_{i_1}, \ldots, r_{i_{l_i}}\}$, $1 \leq i \leq k$, in the fashion explained above. Then the procedure below represents the set $\mathcal{R}$ of rules with priorities:

```
proc $Rules(t)$
    $r_{1_1} \mid r_{1_2} \mid \ldots \mid r_{1_{l_1}} \mid$
    $\{\neg[(\tau_{r_{1_1}} \wedge (\exists \mathbf{x})\zeta_{r_{1_1}}(\mathbf{x})) \vee \ldots \vee (\tau_{r_{1_{l_1}}} \wedge (\exists \mathbf{x})\zeta_{r_{1_{l_1}}}(\mathbf{x}))]?\ ;$
      $[r_{2_1} \mid r_{2_2} \mid \ldots \mid r_{2_{l_2}} \mid$
      $(\neg[(\tau_{r_{2_1}} \wedge (\exists \mathbf{x})\zeta_{r_{2_1}}(\mathbf{x})) \vee \ldots \vee (\tau_{r_{2_{l_2}}} \wedge (\exists \mathbf{x})\zeta_{r_{2_{l_2}}}(\mathbf{x}))]?\ ;\ Rules_{rest})]\}$
endProc,
```

where $\tau_{r_{1_j}}$ and $\zeta_{r_{1_j}}(\mathbf{x})$ denote the event and the condition parts of rule r_{1_j}, which is the j-th rule of the subset $\mathcal{R}_1$ of $\mathcal{R}$, respectively; $Rules_{rest}$ is a GOLOG program representing the remaining rules and their priorities in $\mathcal{R}_3 \cup \ldots \cup \mathcal{R}_k$; and $Rules_{rest}$ iterates the construction in the body of the procedure $Rules(t)$.

Rules within a subset $\mathcal{R}_i$ are selected nondeterministically until one is found, at which point their action parts are executed according to the semantics expressed in (47)–(50). The test action at the end of $\mathcal{R}_i$ means that if no triggered rule of $\mathcal{R}_i$ has a true condition, the rule processing stops for that subset and continues with rules of lower priorities. Notice that if the processing of rules in $\mathcal{R}_i$ leads to the triggering of one of the rules r_{j_k}, such that $j > i$, control goes back to the rules of higher priority.

Example 6 Using the immediate execution model, the procedure for the prioritized rules in Example 2 is given in Figure 4: Upon the signaling of a $price\_inserted$ event, rule $Update\_stocks$ updates the stock price in the database for some customer if this stock is being monitored. Rule Buy_100_shares is also triggered by the same $price\_inserted$ event. If the price of some monitored stock has been updated and the new price lies between some threshold, then a suitable trading action is performed. However, $Update\_stocks$ has priority over Buy_100_shares and will be processed first. In this example, instead of using the notation $\phi[\mathbf{y}]$, we have restored the arguments $\mathbf{y}$ involved, in this case the arguments $Update\_stocks$, Buy_100_shares, and t (for "transaction"). ■

4.5 Properties of Rule Programs

Usually, even a relatively small number of ECA-rules can display a complex and unpredictable run-time behavior, such as non-termination, and discrepancies in the final states of the database depending on how rules are selected for execution. Therefore, in designing ECA-rules, it is important to be able to predict run-time behavior of rule at design-time. This is done by analyzing the set of ECA-rules. Rule analysis is a design-time inspection of rules for compliance with a set of desired properties. The most important properties rule designers must care about are ([54]):

– *Termination*: This ensures that a rule program reaches a database state in which no further rules are triggered.

```
proc $Rules(t)$
$(\pi\ s\_id, pr', clos\_pr)[price\_inserted(Update\_stocks, t)?$ ;
$(\exists\ c, time, bal, pr')[price\_inserted(Update\_stocks, s\_id, pr, time, t) \wedge$
  $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr', clos\_pr, t)]?$ ;
  $stock\_insert(s\_id, pr, clos\_pr, t)]$   |
  $\{\neg[price\_inserted(Update\_stocks, t) \wedge$
   $(\exists\ s\_id, pr, c, time, bal, pr', clos\_pr)[price\_inserted(s\_id, pr, time, t) \wedge$
   $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr', clos\_pr, t)]]?$ ;
   $(\pi\ c, s\_id, 100)[price\_inserted(Buy\_100\_shares, t)?$ ;
    $(\exists\ new\_pr, time, bal, pr, clos\_pr)[price\_inserted(Buy\_100\_shares, s\_id, new\_pr, time, t) \wedge$
    $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr, clos\_pr, t) \wedge new\_pr < 50 \wedge clos\_pr > 70]?$ ;
    $buy(c, s\_id, 100, t)]$   |
    $\{\neg[price\_inserted(Buy\_100\_shares, t) \wedge (\exists\ c, s\_id, new\_pr, time, bal, pr, clos\_pr)$
     $[price\_inserted(Buy\_100\_shares, s\_id, new\_pr, time, t) \wedge customer(c, bal, s\_id, t) \wedge$
     $stock(s\_id, pr, clos\_pr, t) \wedge new\_pr < 50 \wedge clos\_pr > 70]]?$   $\}\}$
endProc .
```

Fig. 4 Prioritized rules for updating stocks and buying shares

– *Confluence*: This ensures that whenever a rule program reaches two final database states, then the two states are the same, independently of the order of the execution of non-prioritized rules.
– *Observable determinism*: This ensures that a rule program always performs the same visible actions, independently of the execution order of non-prioritized rules.

This section shows how the first two of these properties can be expressed in the situation calculus.[14] The last one is left out.

4.5.1 General Properties Here, we briefly illustrate the use of our framework for specifying properties of well formed ConGolog programs. We appeal to a well known hierarchy of properties expressible in temporal logic of Manna and Pnueli ([31]) who distinguish two classes of properties: *Safety*, and *Progress*. In the situation calculus, a safety property is syntactically characterized by a formula of the form $(\forall \mathbf{s})\phi$, where ϕ is any first order past temporal formula expressed in the situation calculus. A progress property is syntactically characterized by a formula of the form $(Q_1 s_1) \cdots (Q_n s_n)\phi$, where ϕ is any first order past temporal formula expressed in the situation calculus, and the Q_i must contain at least one occurrence of $\exists$.

[14] The general treatment of these properties deserves a full paper of its own. Here, we just show how to formulate the properties in the situation calculus without reasoning about them.

A classical example of safety property is the *partial correctness* of a given program T: if T terminates, then it does so in a situation satisfying a desirable property, say ϕ; i.e.

$$(\forall s).Do(T, S_0, s) \supset \phi(s).$$

Notice that transaction systems are designed to terminate. So they constitute a domain where this kind of property can be considered.

Checking partial correctness amounts to establishing the entailment

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset \phi(s). \tag{56}$$

As an illustration of partial correctness checking, we have the entailment (67) from Theorem 3. Another illustration is

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset always(\phi, s),$$

where T is a given well formed ConGolog program, and ϕ is any first order past temporal formula expressed in the situation calculus . Suppose we have the following abbreviation for $well\text{–}formed\text{–}sit(s)$:

$$\begin{aligned}&well\text{–}formed\text{–}sit(s) =_{df}\\ &\qquad (\forall s')[do(Commit(t), s') \sqsubset s \supset \neg(\exists s'')do(Rollback(t), s'') \sqsubset s].\end{aligned}$$

Then the following is a further example of checking correctness:

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset well\text{–}formed\text{–}sit(s),$$

with T being a given well formed ConGolog program.

Given a relational theory $\mathcal{D}$, a property $\phi(s)$, and a well formed ConGolog program T, it is important to look for computationaly feasible ways of establishing the entailment (56). The answer to this question would take us too far aside; nevertheless, we can mention one possible mechanism for doing this: in order to check the property $\phi(s)$, we must answer a historical query, i.e. check the validity of $\phi(s)$ over a finite log starting in S_0 and involving only ground operations. We do this by first generating a ground log S such that $Do(T, S_0, S)$ using the relational theory $\mathcal{D}$. Then we use the regression mechanism for non-Markovian situation calculus theories defined in [16] to reduce our task to establishing an entailment involving only the initial database. Regressing the given formula $\phi(s)$ means using $\mathcal{D}$ to transform it into a logically equivalent formula $\phi'(s')$ which mentions a shorter sublog s' of s. Repeatedly performing this mechanism leads ultimately to a formula whose only log is S_0. So, thanks to the regression, checking a property in the log resulting from the execution of a transaction amounts to checking that property – as a theorem proving task – in the initial database.

4.5.2 Termination and Correctness Recall that termination ensures that a rule program reaches a database state in which no further rules are triggered. In the situation calculus, this can be expressed by writing a formula that captures the fact that a rule program $Rules^(i,j)(t)$ reaches a final situation with the Do predicate. Formally, suppose $\mathcal{D}$ is an active relational theory, and let $Rules^{(i,j)}(t)$ be the ConGolog program corresponding to the execution model (i,j). Then a rule program $Rules^{(i,j)}(t)$ terminates iff

$$(\forall t,s)(\exists s')Do(Rules^{(i,j)}(t),s,s'). \tag{57}$$

So, termination of active rules is a progress properties since one of the quantifiers of the formula (57) is an existential quantifier.

Strictly speaking, termination just means that rule processing ceases at some point. That is what the formula (57) expresses. If we strictly follow the way termination is defined above, that is, as a property that “ensures that a rule program reaches a database state in which no further rules are triggered”, we can view termination as a correctness property. In fact, it is easy to see that the following holds:

Proposition 2 *The following is a valid situation calculus formula:*

$$(\forall t,s,s').Do(Rules^{(i,j)}(t),s,s') \supset \neg\Phi^{(i,j)},$$

where $\Phi^{(i,j)}$ is the formula in the last test condition of $Rules^{(i,j)}$; i.e., for $Rules^{(1,1)}$, $\Phi^{(i,j)}$ is the formula

$$(\exists \mathbf{x}_1)(\tau_1[R_1,t]\wedge\zeta_1(\mathbf{x})[R_1,t]) \vee\ldots\vee (\exists \mathbf{x}_n)(\tau_n[R_n,t]\wedge\zeta_n(\mathbf{x}_n)[R_n,t]).$$

4.5.3 Confluence Confluence ensures that a rule program never reaches two divergent database states, independently of the order of the execution of non-prioritized rules. Notice that, informally, a database state is the set of all database fluents, together with their truth values. Confluence must ensure that all possible executions of a rule program reach the same database state. We may express this property in the situation calculus as follows:

$$\begin{aligned}(\forall Q)(\forall s,s',s'',t).Do(Rules^{(i,j)}(t),s,s')\wedge\\ Do(Rules^{(i,j)}(t),s,s'') \supset \{Q[s']\equiv Q[s'']\},\end{aligned} \tag{58}$$

where Q is a database query, and i and j are fixed. The formula (58) above shows that confluence is a safety property since all the quantifiers involved in it are universal quantifiers.

5 Related Work

In general, the formal specification of active behavior can be classified in two groups. The first group uses the denotational semantics [49] as formalism to describe the execution models of actual ADBMSs [52, 11, 44]. In the second group, logic is used, especially in form of first-order logic, event calculus, or situation calculus [56, 57, 29, 4, 5, 14]. We also will cover other approaches that exist, but do not fit into the present classification, due to their limited scope.

5.1 Denotational Semantics-based Formalization

In [52], Widom formally specifies the execution model of the Starburst ADBMS, using the denotational semantics. A denotational semantics is generally defined as a mapping of the syntactic constructs of a formal language into an abstract meaning formalized in a suitable mathematical model [49]. This mapping is represented as a *meaning function* taking programs of the language as an input value and producing the function computed by those programs. In the context of an ADBMS, a denotational semantics is a meaning function. This takes a set of active rules as an input and produces a function which maps the set of database states and the set of allowed database modification operations into a new set of database states.

The semantics described in [52] assumes both deterministic and non-deterministic selection of which rule to consider first when more than one rule is triggered. It is divided into three parts. The first describes the different domains of various help or supporting functions used to define the meaning function $\mathcal{M}$; the second defines these supporting functions themselves; and the third defines the meaning function $\mathcal{M}$. Here, $\mathcal{M}$ has a set of rules $Rules \in 2^{\mathcal{R}}$ and a rule ordering $o \in \mathcal{O}$ as input, where $\mathcal{R}$ is the set of active rules, $2^{\mathcal{R}}$ is the powerset of $\mathcal{R}$, and $\mathcal{O}$ is the set of rule orderings. The meaning of o and $Rules$, denoted by $\mathcal{M}[Rules, o]$, is defined as a function $\phi : \ \triangle \ \times \ \mathcal{S} \ \longrightarrow \mathcal{S} \cup \{\perp\}$, where $\triangle$ is the domain of sets of database changes, $\mathcal{S}$ is the domain of database states, and $\perp$ denotes the non-termination of rule execution. Putting it all together, Widom formally defines $\mathcal{M}$ as a function $2^{\mathcal{R}} \times \mathcal{O} \longrightarrow \triangle \times \mathcal{S} \rightarrow \mathcal{S} \cup \{\perp\}$.

To our knowledge, this proposal is the first one to have succeeded in giving a formal foundation to active rules. For this reason, it is widely referred to in the literature. Unfortunately, though Widom argues that her work is providing a denotational semantics for the Starburst rule language, it is in fact providing only a specification of the execution model of the Starburst ADBMS, abstracting from its rule language.

Coupaye and Collet present another attempt to use denotational semantics for formally specifying ADBMSs ([11]). They give a formal specification of the execution model of the NAOS system developed at the university of Grenoble. In NAOS, a rule condition can be a query expressed in O_2SQL, which is an object-oriented version of SQL, and a rule action can be a O_2C program. Both O_2SQL and O_2C are languages in the style of Heraclitus (see Section 5.3): they may be used to express the formal semantics of ADBMSs.

Coupaye and Collet are essentially applying the set of semantic functions developed by Widom on the NAOS system, taking the object-oriented character of NAOS conditions and actions into account. For example, they extend the definition of the valuation function for the net effect of rule actions by incorporating the effect of object-oriented related operations such as object creation, or method invocation. They also add some new domains for supporting functions that valuate new features found in NAOS such as the dynamic activation or deactivation of rules.

Like in the denotational semantics approach, we give a declarative semantics that allows to reason about the behavior of active rules. Unlike there, ours is a formalization that can be generalized without difficulties, thus allowing a comparison between the various execution models that exist.

5.2 Logic-based Formalization

The idea of using first-order logic as a mean to formalize ADBMSs has been advocated first by Widom and Zaniolo ([53], [56]). Some other researchers have tried to give a logical account of the semantics of ADBMSs [53, 56, 57, 20, 15, 29, 4, 5, 14]. Most of these researchers have been motivated by the existence of well developed semantics for deductive rules [50, 28, 18, 51].

Databases have been formalized in logic now for about 20 years (for a description of the main contributions to the use of logic in databases, the actual state of the art in commercial implementations, and future trends, see [36]). Reiter [45] provides the first of these formalizations.

Deductive databases (Datalog) are an extension of relational databases formalized *à la* Reiter. Datalog programs are function-free logic programs. Concepts playing an important role in relational databases like integrity constraints are also formalized in logic (for an account, see [36]).

Considering semantics of deductive databases, some researchers attempt to provide similar semantics to ADBMSs [56, 57, 20, 29, 14]). They feel a need for combining active and deductive DBMSs into a single and reconciled paradigm. This need is being addressed in different ways. Widom argues that deductive rules could be naturally extended to run on active DBMSs [53]. Zaniolo, Harrison and Dietrich, and Ludäscher *et al.* state that some syntactically and semantically constrained active rules would run on deductive DBMSs [56, 57, 20, 29]. Finally, Fernandes *et al.* advocate that active and deductive DBMSs have no intrinsic similarities with respect to their operational semantics, but they can be integrated into one hybrid system [14].

The most elaborated semantics in this approach that unifies semantics for active and deductive databases are given in [56, 57, 29]. The main result of this effort is a demonstration of a possible unification of active and deductive rules within a common logical framework of a suitably extended version of Datalog. The unified semantics keep the classical deductive database engine while simulating active rules by re-writing them as deductive rules with updates. The obvious advantage of the approach is the possibility of reusing old components of a deductive DBMS without major changes. In addition, this approach provides a semantic account for an active rule as an unfragmented unity. However, we stress one drawback in the approach: it is not clear whether all aspects of the active behavior like coupling modes or operational semantics of active rules can be simulated within a pure semantics for deductive database updates.

Some other researchers are proposing logic-based formalizations that do not directly address the issue of combining the active and deductive DBMSs into a single and reconciled paradigm. For example, Picouet and Vianu [39, 40] address expressiveness and complexity questions such as: the relevance of active features, their impact on the expressive power and the complexity of the systems, and the simplification of execution models and their equivalence. They describe a generic framework for formalizing active databases. They use it to articulate and factor out common features of sevral prototypes of ADBMSs. Within their framework, they investigate the impact of the various dimensions of the active behavior on the expressiveness and complexity of active databases.

The main contribution of the work of Picouet and Vianu seems to be the insight provided into which active features are essential and which are not. A feature is essential if it has an impact on the expressive power and the complexity of the ADBMS. However, they omit the event part of ECA-rules in their approach. The impact of this omission on the expressiveness and the complexity of an ADBMS is an open question. Though the operational semantics abstracted by Picouet and Vianu seems general enough to account for many existing ADBMSs, the style of their formalization remains operational. In this paper, we have showed how to express execution models in a purely declarative way.

Baral and Lobo develop a situation calculus-based language called $\mathcal{L}_{active}$ to describe actions and their effects, events, and evaluation modes of active rules [4, 5]. What Baral and Lobo propose can be considered as preliminary work on how the situation calculus may be used to formalize active rules: in these papers, many aspects such as modelling complex events, complex and concurrent actions, and transactions remain as future work. Beside this limitation, there is another one: having presented their language, Baral and Lobo give a translation method for transforming active database rules in a corresponding logic program expressed in the situation calculus notation. In our opinion, this presents a conceptual problem: their language seems to be a set of metaconstructs that have to be translated to the situation calculus. Why not directly express active rules in the situation calculus? The purpose of introducing a new language seems unclear to us.

In [41], Pinto introduces various notions of action occurrence. He views an occurrence statement as a constraint on the *legal* path in the tree of possible futures. His theory incorporates many ingredients that are partly user-provided facts. Pinto argues that his framework can be used as an effective tool for formalizing active databases, assuming Reiter's formalization of DB transactions. The idea is to let an action occur in the state following the performance of a triggering action, according to the general (simplified) pattern $\phi \supset occurs(\alpha_2(\bar{x}_2), do(\alpha_1(\bar{x}_1), s))$, where ϕ is the condition of the rule, $do(\alpha, s)$ means the situation following the execution of action α in situation s, and α_1 triggers α_2. The same idea is found in [34]. Pinto and McCarthy's proposals are yet too fragmentary to allow a comparison with our approach.

Bertossi *et al.* [8] propose a situation calculus-based formalization that differ considerably from our approach. They first extend Reiter's specification of database updates to database transactions, incorporating a formalization of static integrity constraints and the notion of action occurrence introduced in [41]. They then specify active rules as sentences of the situation calculus. This 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.

Finally, Fraternali and Tanca address the problem of giving a formal semantics based on formal concepts that captures most of the features of known ADBMSs in the similar way fixpoint semantics captures the deductive nature of deductive DBMSs [15].

5.3 Other Approaches

In [22], Hull and Jacobs present a rich language using an operational semantics. Based on the database language Heraclitus, the discussion in this work shows how to theoretically analyze alternative rule processing semantics. As an example of such an analysis, they examine approaches of accessing delta relations. Heraclitus is an imperative language that is statically typed, supports (possibly persistent) relation types and variables, supports a type called delta which stores insertions to, or deletions from, a relation, and has an embedded relational calculus subpart to manipulate relations and delta values.

Constructs of Heraclitus can be used to express rule processing semantics. Rules are expressed in Heraclitus as a function from deltas to deltas. A particular semantics of an existing system can be expressed as a procedure called *rule application template*, which uses rules defined as functions. A special kind of indexing called *indexed families* is used to manipulate and refer to functions; it is a mapping from integers to the appropriate code fragments. The authors exclude considering explicit arrays of rules, since this would introduce rules as first-class citizens.

Using the framework of Heraclitus, Hull and Jacobs specify the semantics of Starburst.

In [6], an approach based on relational algebra is described, aiming at the analysis and optimization of CA rules. It shows how to propagate changes introduced by the execution of the action part of a rule, for example r_1, to the condition of another rule, say r_2, in order to know whether r_1 may trigger r_2.

There are some other approaches which are of limited scope since they cover only one aspect of active behavior. As an example, graph-based formalism is used, especially in the form of event graphs [10] or Petri nets [17] to model and/or detect composite events. In [1], a graph-based approach is also used to model provable properties of sets of active rules, such as confluence, termination, and determinism.

6 Conclusion

6.1 Summary

This paper has proposed foundations of active databases using the situation calculus. Our approach allows to formally specify and reason about both the ECA-rule language and the execution models for rules. The theories introduced in the paper allow a precise definition of the properties of the main dimensions of active behavior, such as relational databases, database querying and updates, (advanced) database transaction models, events, conditions, actions, execution of ECA-rules, etc.

Developing mathematical foundations for dynamical systems has attracted many research efforts since Amir Pnueli [43] first showed the importance of using temporal logic to specify semantics and dynamical properties of concurrent programs. Ray Reiter's book [47] on situation calculus theories as "logical foundations for specifying and implementing dynamical systems" is mainly about foundations for autonomous, cognitive robots that perceive and act in changing environments and reason about their actions and the knowledge they accumulate about these actions. This paper aimed to give similar foundations, this time not to cognitive agents, but to the dynamical world of active databases. It focused on logical theories for capturing advanced transaction models, complex events, execution models of ECA-rules, and a methodology for obtaining Prolog implementations of these theories.

To start with our endeavour, we have recapitulated an existing framework for capturing database transaction models in the sistuation calculus. Then, we have extended the framework for modelling transactions to reactive and execution models of active behaviors. With respect to these, the main contributions of this paper can succinctly be summarized as follows:

- We constructed logical theories called *active relational theories* to formalize active databases along the lines set by the framework of basic relational theories introduced in [25]. Like the later, active relational theories are non-Markovian theories. They provide the formal semantics of the corresponding active database model, along with an underlying database transaction model. They are an extension of the classical relational theories of [45] to the transaction and active database settings.
- We specified event algebras in the situation calculus, and gave precise semantics to the following dimensions of active behavior: event consumption modes, rule priorities, and net effects. We also specified various execution models in the situation calculus, together with their coupling modes, that is, immediate, deferred, and detached execution models.
- The main result here is a set of classification theorems for the various semantics identified for important dimensions of active behavior such as consumption modes, and execution models. These theorems say roughly which semantics are equivalent and which are not.

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. Thus, an active relational theory corresponding to an ADBMS constitutes a conceptual model for that ADBMS. Since active relational theories are implementable specifications, implementing the conceptual model provides one with a rapid prototype of the specified ADBMS.

6.2 Future Work

Ideas expressed and developed in this paper may be extended in various ways. we mention a few of them.

- **Properties of Rule programs**. Formalizing rules as ConGolog programs can be fruitful in terms of proving formal properties of active rules since proving such properties can be reduced to proving properties of programs. Here, the problems arising classically in the context of active database like confluence and termination [54] are dealt with. In Section 6.2, we gave a preliminary indication on how these properties can be formulated. We appealed to the well known distinction between *Safety* and *Progress* properties due to Manna and Pnueli [31] and argued that the classical properties such termination, confluence, and determinism that are ascribed to ECA-rule programs may be fruitfully viewed in the light of the general classification of Manna and Pnueli. However, much work remains to be done on these issues.
- **Further Classification Theorems**. The issue of classifying execution models and consumption modes needs further study. For example, we classified consumption modes under the assumption that the scope of consumption is global. We would like to know what happens when the local scope is considered. Also, by assuming more advanced transaction models such as those modelled in [25], the issue of classifying the execution models becomes more intriguing.
- **Systematic Implementation of some of the Semantics**. Being foundational, this work has been theoretical by its nature. Though we have shown how to simulate the theories of this paper, it remains to systematically implement a full fledged system by following the guidelines laid down above. This issue is the subject of a future paper.
- **Development Methodology**. Finally, we could explore ways of making the framework of this paper part of a logic-based development methodology for active rule systems. Such a methodology would exhibit the important advantage of uniformity in many of its phases by using the single language of the situation calculus.

References

1. A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behaviour of active database rules. *ACM Transaction on Database Systems*, 20:3–41, 1995.
2. M. Arenas and L. Bertossi. Hypothetical temporal queries in databases. In A. Borgida, V. Chaudhuri, and V. Staudt, editors, *Proceedings of the ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98)*, pages 4.1–4.8, 1998. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.
3. J. Bailey and S. Mikulás. Expressivemenss issues and decision problems for active database event queries. In *ICDT'2001*, pages 69–82, 2001.
4. C. Baral and J. Lobo. Formal characterizations of active databases. In *International Workshop on Logic in Databases, LIDS'96*, 1996.

5. 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.
6. E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 163–179. Springer Verlag, 1993.
7. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, MA, 1987.
8. 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.
9. A. Bonner and M. Kifer. Transaction logic programming. Technical report, University of Toronto, 1992.
10. S. Chakravarthy and D. Mishra. An event specifi cation language (snoop) for active databases and its detection. Technical Report UF-CIS-TR-91-23, University of Florida, 1991.
11. 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.
12. 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 Artifi cial Intelligence*, pages 1221–1226, 1997.
13. G. De Giacomo, Y. Lespérance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus: foundations. *Artifi cial Intelligence*, 121(1-2):109–169, 2000.
14. 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.
15. P. Fraternali and L. Tanca. A structured approach to the defi nition of the semantics of active databases. *ACM Transactions on Database Systems*, 20:414–471, 1995.
16. A. Gabaldon. Non-markovian control in the situation calculus. In *Proceedings of AAAI*, Edmonton, Canada, 2002.
17. S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proceeding on Research Issues in Data Engineering, RIDE'94*, pages 2–9, 1994.
18. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R Kowalski and K.A. Bowen, editors, *Proceedings the 5th International Conference on Logic Programming*, pages 1070–1080, Cambridge, MA, 1988. MIT Press.
19. A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.
20. J.V. Harrisson and S.W. Dietrich. Integrating active and deductive rules. In *Rules in Database Systems, RIDS'93*, pages 288–305, Edinburgh, 1993. Springer-Verlag.
21. 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.
22. R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on VLDB*, Barcelona, 1991.
23. I Kiringa. Simulation of advanced transaction models using golog. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*, 2001.

24. I. Kiringa. Specifying event logics for active databases. In *KRDB*, 2002.
25. I. Kiringa and A. Gabaldon. Synthesizing advanced transaction models using the situation calculus. 2006. Submitted to the *Journal of Intelligent Information Systems*.
26. I. Kiringa and R. Reiter. Specifying semantics of active databases in the situation calculus (extended abstract). In *DBLP*, 2002.
27. 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.
28. J.W. Lloyd. *Foundations of Logic Programming, Second, Extended Edition*. Springer-Verlag, Berlin, 1988.
29. 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.
30. N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.
31. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, New-York, 1991.
32. D. McCarthy and U. Dayal. The architecture of an acctive data base management system. In *ACM-SIGMOD Conference on Management of Data*, Portland, 1989.
33. J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.
34. J. McCarthy and T. Costello. Combining narratives. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'98*, pages 48–59, San Francisco, CA, 1998. Morgan Kaufmann.
35. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artifi - cial intelligence. *Machine Intelligence*, 4:463–502, 1969.
36. J. Minker. Logic and databases: A 20 year retrospective. In *International Workshop on Logic in Databases, LIDS'96*, pages 3–57, 1996.
37. N.W. Paton. *Active Rules in Database Systems*. Springer Verlag, New York, 1999.
38. N.W. Paton, J. Campin, A.A.A. Fernandes, and M.H. Williams. Formal specifi cations of active database functionality: A survey. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 21–35. Springer Verlag, 1995.
39. P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *ACM Symposium on Principles of Database Systems*, pages 126–138, San José, 1995.
40. P. Picouet and V. Vianu. Expressiveness and complexity active databases. In *ICDT'97*, 1997.
41. J.A. Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. *Journal of Logic and Computation*, 8:777–808, 1998.
42. F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–364, 1999.
43. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
44. S. Reddi, A. Poulovassilis, and C. Small. Pfl an active functional dbpl. In N.W. Paton, editor, *Active Rules in Databases Systems*, pages 297–308, New-York, 1999. Springer Verlag.
45. 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.

46. R. Reiter. On specifying database updates. *J. of Logic Programming*, 25:25–91, 1995.
47. R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, 2001.
48. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
49. R.D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, 1976.
50. M.H. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.
51. A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general programs. In *Proceedings the 7th Annual ACM Symposium on Principles of Database Systems*, pages 221–230. ACM Press, 1988.
52. J. Widom. A denotational semantics for the starburst production rule language. *SIGMOD RECORD*, 21(3):4–9, September 1992.
53. J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum? In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 306–315. Springer Verlag, 1993.
54. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.
55. M. Winslett. *Updating Logical Databases*. Cambridge University Press, Cambridge, MA, 1990.
56. C. Zaniolo. A unifi ed semantics for active and deductive databases. In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 271–287. Springer Verlag, 1993.
57. 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.
58. D. Zimmer and R. Unland. On the semantics of complex events in active database managements systems. Unpublished, 2001.

A Proofs

In proving Theorem 4 below, the following lemma will be useful.

Lemma 1 *Let $\mathcal{D}$ be a basic relational theory. Then*

$$\begin{aligned}\mathcal{D}_f \cup \{(12)\} \models (\forall s, a)\{&legal(S_0)\wedge\\ &[legal(do(a,s)) \equiv legal(s) \wedge Poss(a,s)\wedge\\ &(\forall a', t).systemAct(a',t) \wedge responsible(t,a',s) \wedge Poss(a',s) \supset a = a']\}.\end{aligned}$$

Proof: See [25]. □

Proposition 1

It is suffi cient to give a formula of the past temporal fragment of the situation calculus for each of the consumption modes involved.

1. **First**. The formula in (30), i.e.,

$$\begin{aligned}&(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s']\wedge\\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1,t,s^*] \vee \neg(\exists r_2)e_2[r_2,t,s^*])]\end{aligned}$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t],(\exists r_1)e_1[r_1,t] \vee \neg(\exists r_2)e_2[r_2,t]).$$

2. **Consumed Last**. The formula in (32), i.e.,

$$\begin{aligned}&(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s']\wedge\\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*] \wedge \neg(\exists r_2)e_2[r_2,t,s^*]]\end{aligned}$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t],\neg(\exists r_1)e_1[r_1,t] \wedge \neg(\exists r_2)e_2[r_2,t]).$$

3. **Non-Consumed Last**. The formula in (34), i.e.,

$$\begin{aligned}&(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s']\wedge\\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*]]\end{aligned}$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t],\neg(\exists r_1)e_1[r_1,t]).$$

4. **Cumulative**. The formula in (36), i.e.,

$$\begin{aligned}&(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s']\wedge\\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2,t,s^*]]\end{aligned}$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t],\neg(\exists r_2)e_2[r_2,t]). \quad ■$$

Theorem 1

Suppose an event logic $\mathcal{E} = (E, C, \mathcal{L})$ given by:

- $E = \{F\_inserted(r,t,s), F\_inserted(r,t,s), seq\_ev^{CM}(r,t,e_1,e_2,s),$
  $simult\_ev(r,t,e_1,e_2,s), conj\_ev(r,t,e_1,e_2,s), disj\_ev(r,t,e_1,e_2,s),$
  $neg\_ev(r,t,e,s)\}$,
  with $CM \in \{F, CL, NL, CUMUL\}$;
- $C = \{\neg, \wedge, \sqsubset\}$;
- $\mathcal{L}$ is the past temporal fragment of the situation calculus.

Suppose further that $\mathcal{D}$ is a set of situation calculus formula that specify the semantics of events according to the event logic $\mathcal{E}$, and that $e[r,t,s]$ and $e'[r,t,s]$ are two events of the event logic $\mathcal{E}$ such that we want to establish, for given R and T, that $\mathcal{D} \models (\forall s).e[R,T,s] \supset e'[R,T,s]$. Assume that S_n is the actual situation. Since we deal with the past temporal fragment of the situation calculus, the implication problem is reducible to the problem of checking whether $\mathcal{D}$ logically implies

$$\neg e[R,T,S_0] \vee e'[R,T,S_0] \wedge \neg e[R,T,S_1] \vee e'[R,T,S_1] \wedge \cdots \wedge \neg e[R,T,S_n] \vee e'[R,T,S_n], \tag{59}$$

with $S_0 \sqsubseteq S_1 \sqsubseteq \cdots \sqsubseteq S_n$, where $S_1, S_2, \cdots S_{n-1}$ are all the successive intermediate situations between S_0 and S_n. Notice that (59) mentions only atoms (all of which are from the set E) and its only connectors are from the set C.

Using Proposition 1, we can transform (59) into a formula of the past temporal logic fragment of the situation calculus (and vice-versa). Since this past temporal formula will only mention atoms, we use a straightforward encoding to transform each of its atoms into a proposition. By Theorem 4.1 of [48], stating that propositional linear temporal logic is PSPACE-complete, we conclude that the implication problem is PSPACE-hard. The proof for the equivalence problem follows easily from the implication case. ■

Theorem 2

1. From (29), (31), (35), it is sufficient to establish the following three entailments:[15]

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \\ ((\exists r_1)e_1[r_1,t,s^*] \vee \neg(\exists r_2)e_2[r_2,t,s^*])\} \supset \\ (\exists s'')(\forall s^{**})\{s'' \sqsubset s^{**} \sqsubset s \supset \\ \neg(\exists r_1)e_1[r_1,t,s^{**}] \wedge \neg(\exists r_2)e_2[r_2,t,s^{**}]\},\end{aligned} \tag{60}$$

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \\ \neg(\exists r_1)e_1[r_1,t,s^*] \wedge \neg(\exists r_2)e_2[r_2,t,s^*]]\} \supset \\ (\exists s'')(\forall s{*}{*})\{s' \sqsubset s^{**} \sqsubset s \supset \\ \neg(\exists r_2)e_2[r_2,t,s^{**}]]\},\end{aligned} \tag{61}$$

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \\ \neg(\exists r_2)e_2[r_2,t,s^*]]\} \supset \\ (\exists s')(\forall s^*)\{s' \sqsubset s^{**} \sqsubset s \supset \\ ((\exists r_1)e_1[r_1,t,s^{**}] \vee \neg(\exists r_2)e_2[r_2,t,s^{**}])]\}.\end{aligned} \tag{62}$$

Let us establish the entailment (60). We must prove that the antecedent of the involved formula taken as premise entails the existence of a situation s'' such that

$$(\forall s^{**})\{s'' \sqsubset s^{**} \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^{**}] \wedge \neg(\exists r_2)e_2[r_2,t,s^{**}]\}.$$

[15] Without loss of generality, we assume that e_1 and e_2 below are simple event fluents.

We set $s = do(A, s'')$, for some A. Then, obviously, for all s^{**} such that $s'' \sqsubset s^{**} \sqsubset s$, we obtain vacuously $\neg(\exists r_1)e_1[r_1, t, s^{**}] \wedge \neg(\exists r_2)e_2[r_2, t, s^{**}]$. Note that the fact that $\neg(\exists r_1)e_1[r_1, t, s^{**}]$ holds will never contradict the assumptions, since there is no situation between s'' and s in which $(\exists r_1)e_1[r_1, t, s^{**}]$ could hold.

To establish the entailment (61), it suffice to notice that, by applying first order proof rules, we obtain the following goal to prove:

$$\begin{aligned}
&e_2[sk_1(t,s),t,s],\ sk_2(t,s) \sqsubset s,\ e_1[sk_3(t,s),t,sk_2(t,s)],\\
&sk_2(t,s) \sqsubset s^* \sqsubset s \supset (\neg(\exists r_1)e_1[r_1,t,s^*] \wedge \neg(\exists r_2)e_2[r_2,t,s^*]),\\
&s' \sqsubset S^{**} \sqsubset S, (\neg(\exists r_1)e_1[r_1,t,S^{**}] \wedge \neg(\exists r_2)e_2[r_2,t,S^{**}])\\
&\Longrightarrow\\
&\neg(\exists r_2)e_2[r_2,T,S^{**}].
\end{aligned} \tag{63}$$

Finally, to establish the entailments (62), we have to prove:

$$\begin{aligned}
&e_2[sk_1(t,s),t,s],\ sk_2(t,s) \sqsubset s,\ e_1[sk_3(t,s),t,sk_2(t,s)],\\
&sk_2(t,s) \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2,t,s^*],\\
&s' \sqsubset S^{**} \sqsubset S, \neg(\exists r_2)e_2[r_2,t,S^{**}]\\
&\Longrightarrow\\
&(\exists r_1)e_1[r_1,T,S^{**}] \vee \neg(\exists r_2)e_2[r_2,T,S^{**}].
\end{aligned} \tag{64}$$

Both entailments (63) and (64) are obvious.

2. Here, it is sufficient to establish the following three entailments:

$$\begin{aligned}
\mathcal{D} \models &(\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*]\} \supset\\
&\quad (\exists s'')\{(\forall s^{**})[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1,t,s^{**}] \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s^{**}))] \wedge\\
&\quad (\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset [(\exists r_1)e_1[r_1,t,s^{**}] \supset\\
&\qquad\qquad (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s_1)]]\},\\
\mathcal{D} \models &(\exists s')\{(\forall s^{**})[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1,t,s^{**}] \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s^{**}))] \wedge\\
&(\forall s^{**})[s^{**} \sqsubset s' \sqsubset s \supset\\
&\qquad [(\exists r_1)e_1[r_1,t,s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s_1)]]\} \supset\\
&\quad (\exists s'')(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1,t,s^*] \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s^*))] \wedge\\
&\quad (\forall s^*)[s'' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1,t,s^*] \supset\\
&\qquad\qquad (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s_))]],\\
\mathcal{D} \models &(\exists s')(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1,t,s^*] \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s^*))] \wedge\\
&(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset\\
&\qquad [(\exists r_1)e_1[r_1,t,s^*] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r,t,e_1,e_2,s_1)]] \supset\\
&\qquad (\exists s'')(\forall s^*)\{s'' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*]\}.
\end{aligned}$$

The proofs are straightforward though tedious. Let us prove the first of these entailments. After some skolemizations and quantifier eliminations, using $\mathcal{D}$, we have, for fixed r, s, t, e_1, and e_2 (call them R, S, T, E_1 and E_2, respectively), the following to establish:

$$\begin{array}{l} S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*] \\ \Longrightarrow \\ \{(\forall s^{**})[s^{**} \sqsubset s \supset \neg((\exists r_1)E_1[r_1, T, s^{**}] \wedge seq\_ev(R, T, E_1, E_2, s^{**}))] \wedge \\ (\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset \\ \qquad [(\exists r_1)E_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge seq\_ev(R, T, E_1, E_2, s_1)]]\}, \end{array}$$

We first show that

$$\begin{array}{l} S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*] \\ \Longrightarrow \\ (\forall s^{**})[s^{**} \sqsubset s \supset \neg((\exists r_1)E_1[r_1, T, s^{**}] \wedge seq\_ev(R, T, E_1, E_2, s^{**}))]. \end{array}$$

This task is equivalent (after some logical transformations) to:

$$\begin{array}{l} S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*], S^{**} \sqsubset s, \\ \Longrightarrow \\ seq\_ev(R, T, E_1, E_2, S^{**}) \supset \neg(\exists r_1)E_1[r_1, T, S^{**}]. \end{array}$$

By moving $seq\_ev(R, T, E_1, E_2, S^{**})$ to the antecedents, and Modus Ponens in the antecedents and unification, we get $(\exists r_1)E_1[r_1, T, S^{**}]$.
Now we show that

$$\begin{array}{l} S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*] \\ \Longrightarrow \\ (\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset \\ \qquad [(\exists r_1)E_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge seq\_ev(R, T, E_1, E_2, s_1)]]. \end{array}$$

First skolemize the universal $(\forall s^{**})$, and then move $s^{**} \sqsubset s'' \sqsubset s$ and $(\exists r_1)E_1[r_1, t, s^{**}]$ (with appropriate skolem functions instead of s^{**}) before $\Longrightarrow$. After that, it is easy to see that the definition of the LIFO consumption mode given in (39) applies. ■

Theorem 3

The proof is by induction over the structure of the well fromed program T. Assume for fixed s that $Do(T, S_0, s)$.

Case: T is a primitive database update.
By Definition (43), we have

$$(\exists \delta').Trans^*(T, S_0, \delta', s) \wedge Final(\delta', s).$$

By Definition (41), this implies that

$$\begin{array}{l} (\exists \delta').Poss(T, S_0) \wedge \delta' = nil \wedge \\ \quad \{(\exists a'', s'', t)[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge \\ \qquad Poss(a'', s'') \wedge s = do(a'', s'')] \vee \\ \quad s = do(T, S_0) \wedge [(\forall a'', t)systemAct(a'', t) \supset \neg Poss(a'', S_0)]\} \wedge \\ \quad Final(\delta', S_0), \end{array} \qquad (65)$$

which, by the semantics of $Final$ (See [27]) and minor transformations, is equivalent to

$$\begin{array}{l} Poss(T,S_0)\wedge \\ \{(\exists a'',s'',t)[s''=do(T,S_0)\wedge systemAct(a'',t)\wedge \\ \qquad Poss(a'',s'')\wedge s=do(a'',s'')]\vee \\ s=do(T,S_0)\wedge [(\forall a'',t)systemAct(a'',t)\supset \neg Poss(a'',S_0)]\}. \end{array} \tag{66}$$

Now we must show that the following two formulas hold:

$$\begin{array}{l} Poss(T,S_0)\wedge(\exists a'',s'',t)[s''=do(T,S_0)\wedge systemAct(a'',t)\wedge \\ \qquad Poss(a'',s'')\wedge s=do(a'',s'')]\supset legal(s), \end{array} \tag{67}$$

$$\begin{array}{l} Poss(T,S_0)\wedge s=do(T,S_0)\wedge \\ \qquad [(\forall a'',t)systemAct(a'',t)\supset\neg Poss(a'',S_0)]\supset legal(s). \end{array} \tag{68}$$

Proof of (67): By Lemma 1, we have $legal(S_0)$. Notice that (67) is equivalent to

$$\begin{array}{l} Poss(T,S_0)\wedge(\exists a'',t)[systemAct(a'',t)\wedge \\ \qquad Poss(a'',do(T,S_0))\wedge s=do(a'',do(T,S_0))]\supset legal(s). \end{array} \tag{69}$$

By assuming the antecedent of (69), we conclude by Lemma 1 and the fact that $legal(S_0)$ that $legal(s)$.
Proof of (68): Notice that the formula (68) is equivalent to

$$Poss(T,S_0)\wedge(\exists a'',t)[systemAct(a'',t)\supset\neg Poss(a'',S_0)]\supset legal(s). \tag{70}$$

The proof of this is immediate, using Lemma 1.

Case: T is a test action of the form Φ?.
By Definition (43), we have

$$(\exists a').Holds(\Phi,S_0,s)\wedge Final(nil,s).$$

Therefore, by the semantics of $Final$, we have $Holds(\Phi,S_0,s)$. By unwinding $Holds(\Phi,S_0,s)$ and using Definition 18, whenever we reach a fluent literal, we record that literal into the log. Therefore, since $legal(S_0)$, by the UPA for test actions, we get by Lemma 1 $legal(s)$, with $s=do([\phi_1,\cdots,\phi_n],S_0)$. The ϕ_i are fluents introduced into the log by test actions generated through the unwinding of $Holds(\Phi,S_0,s)$.

Cases: T neither a primitive database update, nor a test action.
All these cases reduce to the base cases treated above by unwinding the program T using the ConGolog definitions in [27]. ∎

Theorem 4

As in Theorem 3, the proof is by induction on the structure of T. All cases to consider are like there, except for primitive database updates. Hence, we deal with this case. Assume for fixed s that $Do(T,S_0,s)$. Then, as in Theorem 3, by Definition (43), we have

$$(\exists\delta').Trans^*(T,S_0,\delta',s)\wedge Final(\delta',s).$$

By Definition (51), this implies that

$$\begin{aligned}&(\exists \delta', s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge \delta' = nil \wedge\\ &\quad \{[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee\\ &\quad\; s^* = do(T, S_0) \wedge [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge\\ &\quad\; Do(Rules(t), s^*, s)\} \wedge\\ &\quad Final(\delta', S_0),\end{aligned} \tag{71}$$

which, by the semantics of $Final$ (See [27]) and minor transformations, is equivalent to

$$\begin{aligned}&(\exists s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge\\ &\quad \{[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee\\ &\quad\; s^* = do(T, S_0) \wedge [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge\\ &\quad\; Do(Rules(t), s^*, s)\}.\end{aligned} \tag{72}$$

Now we must show that the following two formulas hold:

$$\begin{aligned}&(\exists s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge\\ &\quad [s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge\\ &\qquad\quad s = do(a'', s'')] \supset legal(s),\end{aligned} \tag{73}$$

$$\begin{aligned}&(\exists s'', a'', s'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge s^* = do(T, S_0) \wedge\\ &\qquad\qquad [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge\\ &\qquad\qquad\; Do(Rules(t), s^*, s) \supset legal(s).\end{aligned} \tag{74}$$

Proof of (73): By Lemma 1, we have $legal(S_0)$. Notice that (67) is equivalent to

$$\begin{aligned}&(\exists a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge\\ &\qquad [systemAct(a'', t) \wedge Poss(a'', do(T, S_0)) \wedge\\ &\qquad\quad s = do(a'', do(T, S_0))] \supset legal(s).\end{aligned} \tag{75}$$

From the antecedent of (75), we get, by Lemma 1 and the fact that $legal(S_0)$, that

$$(\exists s^*, t).TransOf(T, t, S_0) \wedge S_0 \sqsubset s \wedge legal(s). \tag{76}$$

Therefore, we conclude that $legal(s)$.

Proof of (74): Through an argument similar to the proof of (74), we find that there is a situation s^* that is legal. Now, to execute any one of the rule programs (47) – (50) in some legal situation s^* to reach situation s, all the actions involved must have been possible according to the semantics of ConGolog programs [27]. Therefore the outcome s must be legal, i.e. $legal(s)$ holds. ■

Theorem 5

By Definition 21, we must prove that, whenever Q is a database query, we have

$$\begin{aligned}&(\forall t, s).[(\exists s').Do(Rules^{(2,2)}(t), s, s') \wedge Q[s']] \supset\\ &\qquad\qquad [(\exists s'').Do(Rules^{(1,1)}(t), s, s'') \wedge Q[s'']].\end{aligned} \tag{77}$$

Therefore, by the definitions of $Rules^{(1,1)}(t)$ and $Rules^{(2,2)}(t)$, we need to prove that

$$\begin{array}{l}
(\forall t, s).\\
\{(\exists s').Do(\{(\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1, t]? \ ; \ (\zeta_1(\mathbf{x}_1)[R_1, t] \wedge \\
\qquad\qquad\qquad assertionInterval(t))? \ ; \ \alpha_1(\mathbf{y}_1)]| \\
\qquad\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n, t]? \ ; \ (\zeta_n(\mathbf{x}_n)[r_n, t] \wedge \\
\qquad\qquad\qquad assertionInterval(t))? \ ; \ \alpha_n(\mathbf{y}_n)]| \\
\quad \neg\{[(\exists \mathbf{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\mathbf{x})[R_1, t]) \vee \ldots \\
\qquad \vee (\exists \mathbf{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\mathbf{x}_n)[R_n, t])] \wedge \\
\qquad\qquad\qquad\qquad assertionInterval(t)\} \ ?\}, \ s, \ s') \wedge Q[s']\} \supset \\
\{(\exists s'').Do(\{(\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\mathbf{x}_1)[R_1, t]? \ ; \ \alpha_1(\mathbf{y}_1)[R_1, t]]| \\
\qquad\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\mathbf{x}_n)[R_n, t]? \ ; \ \alpha_n(\mathbf{y}_n)[R_n, t]]| \\
\quad \neg[(\exists \mathbf{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\mathbf{x})[R_1, t]) \vee \ldots \vee \\
\qquad\qquad (\exists \mathbf{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\mathbf{x}_n)[R_n, t])] \ ?\}, \ s, \ s'') \wedge Q[s'']\}.
\end{array} \tag{78}$$

By the definition (43) of Do and the semantics of ConGolog ([27]), we must prove:

$$\begin{array}{l}
(\forall t, s).\\
\{(\exists s').Trans^*(\{(\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1, t]? \ ; \ (\zeta_1(\mathbf{x}_1)[R_1, t] \wedge \\
\qquad\qquad\qquad assertionInterval(t))? \ ; \ \alpha_1(\mathbf{y}_1)]| \\
\qquad\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n, t]? \ ; \ (\zeta_n(\mathbf{x}_n)[r_n, t] \wedge \\
\qquad\qquad\qquad assertionInterval(t))? \ ; \ \alpha_n(\mathbf{y}_n)]| \\
\quad \neg\{[(\exists \mathbf{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\mathbf{x})[R_1, t]) \vee \ldots \\
\qquad \vee (\exists \mathbf{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\mathbf{x}_n)[R_n, t])] \wedge \\
\qquad\qquad\qquad\qquad assertionInterval(t)\} \ ?\}, \ s, \ nil, \ s') \wedge Q[s']\} \supset \\
\{(\exists s'').Trans^*(\{(\pi \mathbf{x}_1, \mathbf{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\mathbf{x}_1)[R_1, t]? \ ; \ \alpha_1(\mathbf{y}_1)[R_1, t]]| \\
\qquad\qquad\qquad\qquad \vdots \\
\quad (\pi \mathbf{x}_n, \mathbf{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\mathbf{x}_n)[R_n, t]? \ ; \ \alpha_n(\mathbf{y}_n)[R_n, t]]| \\
\quad \neg[(\exists \mathbf{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\mathbf{x})[R_1, t]) \vee \ldots \vee \\
\qquad\qquad (\exists \mathbf{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\mathbf{x}_n)[R_n, t])] \ ?\}, \ s, \ nil, \ s'') \wedge Q[s'']\}.
\end{array} \tag{79}$$

By the semantics of $Trans$, we may unwind the $Trans^*$ predicate in the antecedent of (79) to obtain, in each step, a formula which is a big disjunction of the form

$$\begin{array}{l}
(\exists s').[(\phi_1^1 \wedge \phi_2^1 \wedge assertionInterval(t) \wedge \phi_3^1) \vee \cdots \\
\quad \vee (\phi_1^n \wedge \phi_2^n \wedge assertionInterval(t) \wedge \phi_3^n) \vee \Phi] \wedge Q[s'],
\end{array} \tag{80}$$

where ϕ_1^i represents the formula $\tau_i[R_i, t]$, ϕ_2^i represents $\zeta_i(\mathbf{x}_i)[R_i, t]$, and ϕ_3^i represents the situation calculus formula generated from $\alpha_i(\mathbf{y}_i)$, 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 (79) to obtain, in each step, a formula which is a big disjunction of the form

$$(\exists s'').[(\phi_1^1 \wedge \phi_2^1 \wedge \phi_3^1) \vee \cdots \vee (\phi_1^n \wedge \phi_2^n \wedge \phi_3^n) \vee \Phi'] \wedge Q[s''], \tag{81}$$

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 conjunction 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 (80) implies (81). ■

Theorem 6

This proof is similar to that of Theorem 5, so we omit it. ■

Corollary 1

The proof is immediate from Theorems 5 and 6. ■