Towards a General Theory of Advanced Transaction Models in the Situation Calculus (Extended Abstract)

Iluju Kiringa
Department of Computer Science
University of Toronto, Toronto, Canada
kiringai@cs.toronto.edu

Abstract

We propose a theory for describing, reasoning about, and simulating transaction models that relax some of the ACID properties of classical transactions. Such models have been proposed for database applications involving long-lived, endless, and cooperative activities. Our approach appeals to non-Markovian theories, in which one may refer to past states other than the previous one. We illustrate our framework by formalizing closed nested transactions (CNTs). We first formulate CNTs as a suitable non-Markovian theory. Then we define a legal database log as one whose actions are all possible and in which all the $Commit$ and $Rollback$ actions must occur whenever they are possible. After that, we show that the relaxed ACID constraints are properties of legal logs and logical consequences of the theory corresponding to the CNTs. Finally, we use such a specification as a background theory for transaction programs written in the language GOLOG.

1 Introduction

Transaction systems that constitute the state of the art in database systems have a flat structure defined in terms of the so-called ACID (Atomicity-Consistency-Isolation-Durability) properties. From the system point of view, a database transaction is a sequence of operations on the database state, which exhibit the ACID properties and are bracketed by $Begin$ and $Commit$ or $Begin$ and $Rollback$ ([9]).

A transaction is atomic when it either brings the database from the initial state to the final state, or it appears as it had never done any work. Consistency means that, given an initial database state that satisfies all the integrity constraints of the database, the final state also satisfies them. Two transactions are isolated when their interleaved execution yields the same result as a serial execution. Finally, durability means that from the commitment point onwards, the results of a transaction are permanent.

Various transaction models have been proposed to extend the classical flat transactions by relaxing some of the ACID properties ([7],[10]). Such extensions, generally called *advanced transaction models* (ATMs), are proposed for improving the functionality and the performance of applications involving long-lived, endless, and cooperative activities.

The ATMs, however, have been proposed in an *ad hoc* fashion, thus lacking in generality in a way that it is not obvious to compare the different ATMs, to exactly say how they extend the traditional flat model, and to formulate their properties in a way that one clearly sees which new functionality has been added, or which one has been subtracted. To address these questions, there is a need for a **general and common framework** within which to concisely specify ATMs, simulate these, specify their properties, and reason about these properties. Thus far, ACTA ([5]) seems to our knowledge the only framework addressing these questions at a high level of generality; ACTA uses a first order language to capture the semantics of any ATM.

In this paper, we present a framework for specifying database transactions at the logical level using the situation calculus ([16]). Our approach appeals to non-Markovian theories([8]), in which one may refer to past states other than the previous one. We provide the formal semantics of an ATM by specifying it as a theory of the situation calculus called *basic relational theory*, which is a set of sentences suitable for non-Markovian control in the context of database transactions. We illustrate our framework by formalizing closed nested transactions (CNTs: [17]). We first formulate CNTs as basic relational theories. We then define a legal database log as one whose actions are all possible and in which all the $Commit$ and $Rollback$ actions must occur whenever they are possible. After that, we show, by means of a few examples, that the known properties of the CNTs, including the relaxed ACID constraints, are properties of legal logs and logical consequences of the basic relational theory corresponding to the CNTs. Finally, we also indicate how to implement such a specification as a background theory for transaction programs written in the situation calculus based programming language GOLOG.

2 Logical Foundations

We use a *basic relational language*, which is a finite fragment of the situation calculus ([16],[8]) that is suitable for modeling relational database transactions. The language is a many-sorted second order language with sorts for $actions$, $situations$, and $objects$. $Actions$ are first order terms consisting of an action function symbol and its arguments (e.g., the action of the transaction t deleting the tuple $(tid, tbal)$ from the relation $tellers$ is denoted by $t\_del(tid, tbal, t)$). $Situations$ are first order terms denoting finite sequences of actions; they are represented using a binary function symbol do: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. There is a distinguished constant S_0 denoting the initial situation; S_0 stands for the empty action sequence. $Objects$ represent domain specific individuals other than actions and situations. In formalizing databases, actions correspond to the elementary database operations of inserting, deleting and updating relational tuples, and situations represent the database *log*. Relations and functions whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols and function symbols with last argument a situation term (e.g., the relation $tellers$ is represented by the fluent $tellers(tid, tbal, t, s)$). The language also includes special predicates $Poss$, and $\sqsubset$; $Poss(a, s)$ means that the action a 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. In database terms, $s \sqsubset s'$ means that s is a proper sublog of the log s'.

For simplicity, we consider only primitive update operations corresponding to insertion or deletion of tuples into relations. For each such relation $F(\vec{x}, t, s)$, a *primitive internal action* is a parameterized primitive action of the situation calculus of the form $F\_ins(\vec{x}, t)$ or $F\_del(\vec{x}, t)$. Intuitively, $F\_ins(\vec{x}, t)$ and $F\_del(\vec{x}, t)$ denote the actions of inserting the tuple $\vec{x}$ into and deleting it from the relation F by the transaction t, respectively; for convenience, we will abbreviate long symbols when necessary (e.g., $account\_ins(\vec{x}, t)$ will be abbreviated as $a\_ins(\vec{x}, t)$). Below, we will use the following abbreviation:

$$writes(a, F, t) =_{df} (\exists \vec{x}).a = F\_ins(\vec{x}, t) \vee a = F\_del(\vec{x}, t),$$

one for each fluent. We distinguish the primitive internal actions from *primitive external ac-*

tions which are $Begin(t)$, $Spawn(t,t')$, $End(t)$, $Commit(t)$, 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. Finally, by convention in this paper, a free variable will always be implicitly bound by a prenex universal quantifier.

In [16], a database domain is axiomatized in the situation calculus with axioms describing *how* and under what *conditions* the database is changing or not changing as a result of performing updates. Such axioms are called basic action theory. They comprise: domain independent foundational axioms for situations; action precondition axioms, one for each action term, stating the conditions of change; successor state axioms, one for each fluent, stating how change occurs; unique names axioms for action terms; and axioms describing the initial situation.

3 The Specification Framework

We extend the basic action theories of [16] to include a specification of relational database transactions, by giving action precondition axioms for external actions. We also give successor state axioms that state how change occurs in databases in the presence of both internal and external actions. All these axioms provide the *first dimension* of the situation calculus framework for axiomatizing transactions, namely the axiomatization of the effects of transactions on fluents; they also comprise axioms indicating which transactions are conflicting with each other, and what actions each transaction is responsible for.

A useful concept that underlies most of the ATMs is that of *responsibility* over changes operated on data items. In CNTs, a parent transaction will take responsibility of changes done by any of its committed children. The only way we can keep track of those responsibilities is to look at the transaction arguments of the actions present in the log. To that end, we introduce a fluent $responsible(t,a,s)$, which intuitively means that transaction t is responsible for the action a in the log s, which we characterize with an appropriate successor state axiom:

$$\begin{aligned}&responsible(t,a',do(a,s)) \equiv transOf(a',t,s) \wedge \neg(\exists t^*)parent(t,t^*,s) \vee\\ &\quad(\exists t^*)[parent(t,t^*,s) \wedge a = Commit(t^*) \wedge responsible(t^*,a')] \vee\\ &\quad responsible(t,a',s) \wedge \neg termAct(a,t),\end{aligned} \tag{1}$$

i.e., each transaction is considered responsible for any action whose last argument bears its name, or for the actions of its committed children, or else for those actions it was already responsible for before terminating. Here, $transOf(a,t,s)$, $parent(t,t',s)$, and $termAct(a,t)$ are defined as follows:

$$transOf(a(\vec{x},t),t,s) \equiv true, \tag{2}$$

$$\begin{aligned}&parent(t,t',do(a,s)) \equiv a = Spawn(t,t') \vee\\ &\qquad\qquad parent(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'),\end{aligned} \tag{3}$$

$$termAct(a,t) =_{df} a = Commit(t) \vee a = Rollback(t). \tag{4}$$

We use the axioms above to capture the typical relationships that hold between transactions in the hierarchy of a nested transaction.

To express *conflicts* among transactions, we need the fluents $updConflict(a,a',s)$ and $transConflict(t,t',s)$, whose intuitive meaning is that the action a is conflicting with the action a' in s, and the transaction t is conflicting with the transaction t' in s, respectively; their characterization is as follows:

$$updConflict(a,a',s) =_{df} \bigvee_{F \in \mathcal{F}} (\exists \vec{x})\neg[F(\vec{x},t,do(a,do(a',s))) \equiv F(\vec{x},t,do(a',do(a,s)))];$$

here, $\mathcal{F}$ is the set of fluents of the basic relational language; the later definition says that two internal actions a and a' conflict in the log s iff the value of the fluents depends on the order in which a and a' appear in s.

$$\begin{aligned}
&transConflictNT(t,t',do(a,s)) \equiv t \neq t' \wedge responsible(t',a,s) \wedge \\
&\quad (\exists a',s')[responsible(t,a',s) \wedge updConflict(a',a,s) \wedge do(a',s') \sqsubset s \wedge \\
&\quad \neg responsible(t,a,s) \wedge running(t',s) \wedge ((\exists t'')parent(t,t'',s) \supset \neg ancestor(t,t',s)) \vee \\
&\quad transConflictNT(t,t',s) \wedge \neg termAct(a,t);
\end{aligned} \tag{5}$$

$ancestor(t,t',s)$ is defined in the usual way using $parent(t,t',s)$ and $running(t',s)$ is defined below.

$$\begin{aligned}
&running(t,s) =_{df} (\exists s').\{do(Begin(t),s') \sqsubseteq s \wedge \\
&\quad (\forall a,s'')[do(Begin(t),s') \sqsubset do(a,s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)] \vee \\
&\quad (\exists t').do(Spawn(t',t),s') \sqsubseteq s \wedge \\
&\quad (\forall a,s'')[do(Spawn(t',t),s') \sqsubset do(a,s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)]\}.
\end{aligned} \tag{6}$$

Intuitively, (5) means that transaction t conflicts with transaction t' in the log $do(a,s)$ iff internal actions they are responsible for are conflicting in s, t' executes its internal action a after t has executed the internal action a' in the log s, t is not responsible for the action of t' it is conflicting with, t' is running; moreover, a transaction cannot conflict with actions his ancestors are responsible for; t also conflicts with t' iff both did so in s and a is not terminating t.

A further useful fluent that we provide in the general framework is $readsFrom(t,t',s)$. This is used in most transaction models as a source of dependencies among transactions, and intuitively means that the transaction t reads a value written by the transaction t' in the log s. The successor state axiom for this fluent depends on the application.

The *second dimension* of the situation calculus framework is made of *dependencies* between transactions. All the dependencies expressed in ACTA ([5]) can also be expressed in the situation calculus. As an example, we have:

Weak Rollback Dependency of t on t'

$$\begin{aligned}
&do(Rollback(t'),s') \sqsubset s^* \supset \\
&\quad \{(\forall s)do(Commit(t),s) \not\sqsubseteq do(Rollback(t'),s') \supset (\exists s'')do(Rollback(t),s'') \sqsubset s^*\};
\end{aligned}$$

i.e., if t' rolls back in a log s^*, then, whenever t does not commit before t', t must also roll back in s^*.

Further dependencies occurring in CNTs are: **Commit Dependency** of t on t', i.e., if t commits in a log s^*, then, whenever t' commits in s^*, t' commits before t; **Strong Commit Dependency** of t on t', i.e., if t' commits in a log s^*, then t must also commit in that log; and **Rollback Dependency** of t on t', i.e., if t' rolls back in a log s^*, then t must also roll back in that log. The specification of an ATM must be given in such a way that all these dependencies are properties of legal database logs of that ATM.

To control dependencies that may develop among running transactions, we use a set of predicates denoting these dependencies. For example, we use $c\_dep(t,t',s)$, $sc\_dep(t,t',s)$, $r\_dep(t,t',s)$, and $wr\_dep(t,t',s)$ to denote the commit, strong commit, rollback, and weak rollback dependencies, respectively. These are fluents whose truth value is changed by the relevant transaction models by taking into account dependencies generated by the execution of the actions of these transactions. Appropriate successor state axioms must be given for these fluents.

4 Closed Nested Transactions

A closed nested transaction is a set of transactions (called subtransactions) forming a tree structure, meaning that any given transaction, the parent, may spawn a subtransaction, the child, nested in it. A child can commit permanently only if its parent has committed; thus it cannot commit unilaterally.[1] If a parent transaction rolls back, all its children are rolled back. However, if a child rolls back, the parent may execute a recovery procedure of its own. Each subtransaction, except the root, fulfills the A, C, and I among the ACID properties. The root (level 1) of the tree structure is the only transaction to satisfy all of the ACID properties.

Notice that we do not introduce a new sort for transactions, as is the case in [3]; we treat transactions as run-time activities, whose compile-time counterparts will be GOLOG programs introduced in Section 5. We refer to transactions by their names that are of sort *object*.

In the sequel of this paper, we use a Debit/Credit example which we now describe. The application involves a basic relational language with: **fluents** $served(aid, s)$, $tellers(tid, tbal, t, s)$, $branches(bid, bbal, bname, t, s)$, and $accounts(aid, bid, abal, t, s)$; a **situation independent predicate** $requested(aid, req)$; and **actions** $report(aid)$, $t\_ins(tid, tbal, t)$, $t\_del(tid, tbal, t)$, $a\_ins(aid, bid, abal, tid, t)$, $a\_del(aid, bid, abal, tid, t)$, $b\_ins(bid, bbal, bname, t)$, and $b\_del(bid, bbal, bname, t)$. The meaning of the arguments of fluents are self explanatory; and the basic relational language also includes the external actions given in Section 2.

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

Foundational Axioms. These are constraints imposed on the structure of histories representing database logs. As they play no further role in this paper, we omit them.

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 transaction end. For example, we may enforce the following IC:

$$accounts(aid, bid, abal, tid, t, s) \wedge accounts(aid, bid', abal', tid', t', s) \supset \\ bid = bid', abal = abal', tid = tid',$$

and similar ones for the $branches$ and $tellers$ fluents; these are primary key constraints. We may verify the IC

$$accounts(aid, bid, abal, tid, t, s) \supset abal \geq 0.$$

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

$$Poss(A(\vec{x}, t), s) \equiv (\exists t')\Pi_A(\vec{x}, t', s) \wedge IC^e(do(A(\vec{x}, t), s)) \wedge running(t, s). \tag{7}$$

Here, $\Pi_A(\vec{x}, t, s)$ is a formula with free variables among $\vec{x}, t$, and s, and $IC^e(s)$ abbreviates $\bigwedge_{IC \in \mathcal{IC}_e} IC(s)$. These axioms characterize the preconditions of the update A. As an example, the following states that it is possible for the transaction t to insert a tuple $(tid, tbal)$ into the $teller$ relation relative to the database log s iff, as a result of performing the actions in the log, that tuple would not already be present in the $teller$ relation, the integrity constraints are satisfied, and transaction t is running:

$$Poss(t\_del(tid, tbal, t), s) \equiv (\exists t')teller(tid, tbal, t', s) \wedge \\ IC^e(do(t\_del(tid, tbal, t), s)) \wedge running(t, s).$$

[1] This is why the nesting considered here is called "closed", as opposed to "open" nested transactions where children may commit unilaterally ([17]).

Successor State Axioms. These have the syntactic form

$$
\begin{aligned}
&F(\vec{x}, t, do(a, s)) \equiv (\exists \vec{t'})\Phi_F(\vec{x}, a, \vec{t'}, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\
&\quad [(\exists t'').a = Rollback(t'') \wedge \neg(\exists t^*)parent(t^*, t'', s) \wedge restoreBeginPoint(F, \vec{x}, t'', s)] \vee \\
&\quad [(\exists t'').a = Rollback(t'') \wedge (\exists t^*)parent(t^*, t'', s) \wedge restoreSpawnPoint(F, \vec{x}, t'', s)],
\end{aligned} \tag{8}
$$

one for each relation of the basic relational language, where $\Phi_F(\vec{x}, a, \vec{t}, s)$ is a formula with free variables among $\vec{x}, a, \vec{t}, s$; $restoreBeginPoint(F, \vec{x}, t, s)$ is the following abbreviation:

$$
\begin{aligned}
&restoreBeginPoint(F, \vec{x}, t, s) =_{df} \\
&\quad [(\exists a^*, s^*, s', t').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \wedge writes(a^*, F, t) \wedge F(\vec{x}, t', s')] \vee \\
&\quad [(\forall a^*, s^*, s').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \supset \neg writes(a^*, F, t)] \wedge (\exists t')F(\vec{x}, t', s),
\end{aligned} \tag{9}
$$

and $restoreSpawnPoint(F, \vec{x}, t, s)$ is an abbreviation expanded in a similar way. Intuitively, $restoreBeginPoint(F, \vec{x}, t, s)$ means that the transaction t restores the value that the fluent F with arguments $\vec{x}$ had before the execution of its $Begin$ action in the log s if the transaction t has updated F; it keeps the value it had in s otherwise. Given the actual situation s, the successor state axioms characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of all the past situations. A successor state axiom for the fluent $tellers(tid, tbal, t, s)$ is as follows:

$$
\begin{aligned}
&tellers(tid, tbal, t, do(a, s)) \equiv ((\exists t_1)a = t\_ins(tid, tbal, t_1) \vee \\
&(\exists t_2)tellers(tid, tbal, t_2, s) \wedge \neg(\exists t_3)a = t\_del(tid, tbal, t_3, tid)) \wedge \neg(\exists t')a = Rollback(t') \vee \\
&(\exists t').a = Rollback(t') \wedge \neg(\exists t'')parent(t'', t', s) \wedge \\
&\qquad\qquad restoreBeginPoint(tellers, (tid, tbal, tid), t', s) \vee \\
&a = Rollback(t') \wedge (\exists t'')parent(t'', t', s) \wedge \\
&\qquad\qquad restoreSpawnPoint(tellers(tid, tbal, tid), t', s).
\end{aligned}
$$

This states that the tuple $(tid, tbal)$ will be in the $tellers$ relation relative to the log $do(a, s)$ iff the last database operation a in the log inserted it there, or it was already in the $tellers$ relation relative to the log s, and a didn't delete it; all this, provided that the operation a is not rolling the database back. If a is rolling the database back, the $tellers$ relation will get a value according to the logic of (9).

Precondition Axioms for External Actions. This is the following set of action precondition axioms for the external actions of CNTs:

$$
\begin{aligned}
Poss(Begin(t), s) \equiv{} & \neg(\exists t')parent(t', t, s) \wedge \\
& [s = S_0 \vee (\exists s', t').t \neq t' \wedge do(Begin(t'), s') \sqsubset s],
\end{aligned} \tag{10}
$$

$$
\begin{aligned}
Poss(Spawn(t, t'), s) \equiv{} & t \neq t' \wedge \\
& (\exists s', t'')[do(Begin(t), s') \sqsubset s \vee do(Spawn(t'', t), s') \sqsubset s],
\end{aligned} \tag{11}
$$

$$
Poss(End(t), s) \equiv running(t, s), \tag{12}
$$

$$
\begin{aligned}
Poss(Commit(t), s) \equiv{} & (\exists s').s = do(End(t), s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\
& (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s] \wedge \\
& (\forall t')[c\_dep(t, t', s) \wedge \neg(\exists s^*)do(Rollback(t'), s^*) \sqsubseteq s \supset \\
& \qquad (\exists s')do(Commit(t'), s') \sqsubset s)],
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
Poss(Rollback(t), s) \equiv (\exists s').s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \vee \\
(\exists t', s'').r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubset s' \vee \\
(\exists t', s^*).wr\_dep(t, t', s) \wedge do(Rollback(t'), s^*) \sqsubset s \wedge \\
\neg(\exists s^{**}) do(Commit(t), s^{**}) \sqsubset do(Rollback(t'), s^*).
\end{aligned} \tag{14}
$$

Dependency axioms. These are axioms used to capture how dependencies arise among transactions. These axioms are also used to capture the notion of *recoverability*, *avoiding cascading rollbacks*, etc, of the classical concurrency control theory ([2]). For CNTs, we have:

$r\_dep(t, t', s) \equiv transConflictNT(t, t', s)$,

$sc\_dep(t, t', s) \equiv readsFrom(t, t', s)$,

$c\_dep(t, t', do(a, s)) \equiv a = Spawn(t, t') \vee c\_dep(t, t', s) \wedge \neg termAct(a, t) \wedge \neg termAct(a, t')$,

$wr\_dep(t, t', do(a, s)) \equiv a = Spawn(t', t) \vee wr\_dep(t, t', s) \wedge \neg termAct(a, t) \wedge \neg termAct(a, t')$.

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

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

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

one for each fluent F. Here, the $\vec{C}^i$ are tuples of constants. Also, $\mathcal{D}_{S_0}$ includes unique name axioms for constants of the database, and axioms stating the conflicting updates.

The axioms above capture the notion of a situation being located in the past relative to the current situation which we express with the predicate $\sqsubset$. Thus they capture non-Markovian control ([8]). We call these axioms a *basic relational theory*, and define a relational database as a pair $(\mathfrak{R}, \mathcal{D})$, where $\mathfrak{R}$ is a basic relational language and $\mathcal{D}$ is a basic relational theory.

A fundamental property of $Rollback(t)$ and $Commit(t)$ actions is that the database system *must* execute them in any database state in which they are possible. In this sense, they are coercive actions, and we call them *system actions*:

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

Therefore, logs must be constrained to take the system actions into account, as well as the requirement that all actions in the log be possible. We capture these requirements as follows:

$$
\begin{aligned}
legal(s) =_{df} (\forall a, s^*)[do(a, s^*) \sqsubseteq s \supset Poss(a, s^*)] \wedge \\
(\forall a', a'', s', t)[systemAct(a', t) \wedge responsible(t, a') \wedge \\
responsible(t, a'') \wedge Poss(a', s') \wedge do(a'', s') \sqsubset s \supset a' = a''].
\end{aligned} \tag{15}
$$

Now we state the (relaxed) ACID properties of CNTs as sentences of the situation calculus that are logically implied by the relational theory that captures CNTs. Here, we only illustrate the A and I of the ACID properties.

Theorem 1 *(***Atomicity***) Suppose $\mathcal{D}$ is a relational theory. Then for every relational fluent F*

$$
\begin{aligned}
\mathcal{D} \models legal(s) \supset \{[s' = do(Begin(t), s_1) \vee s' = do(Spawn(t), s_1)] \wedge \\
s' \sqsubset do(a, s_2) \sqsubset s \wedge (\exists a^*, s^*)[s' \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, t)] \supset \\
[(a = Rollback(t) \supset ((\exists t_1) F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2) F(\vec{x}, t_2, s_1))) \wedge \\
((\exists t_1) F(\vec{x}, do(Commit(t), t_1, s_2)) \equiv (\exists t_2) F(\vec{x}, t_2, s_2))]\}.
\end{aligned}
$$

This says that rolling back a transaction t restores any modified fluent to the value it had just before the $Begin(t)$ or $Spawn(t', t)$ action, and committing endorses the value it had in the situation just before the $Commit(t)$ action.

Theorem 2 (No-Orphan-Commits: [5]) *Suppose $\mathcal{D}$ is a relational theory. Then, whenever a child's parent terminates before the child commits, the later, an orphan, is rolled back; i.e.,*

$$\mathcal{D} \models legal(s) \supset \{parent(t,t',s) \wedge termAct(a,t) \wedge \\ do(Commit(t'),s') \not\sqsubset do(a,s'') \sqsubset s \supset (\exists s^*)do(Rollback(t'),s^*) \sqsubset s\}.$$

This property, combined with the atomicity of all transactions of the CNT tree, leads to the fact that, should a root transaction roll back, then so must all its subtransactions. This is where the D in the ACID acronym is relaxed for subtransactions.

Finally, we turn to the important property of serializability which we express as follows:

$$transConflictNT^*(t,t',s) =_{df} (\forall C)[(\forall t)C(t,t,s) \wedge \\ (\forall s,t,t',t'')[C(t,t'',s) \wedge transConflictNT(t'',t',s) \supset C(t,t',s)] \supset C(t,t',s)],$$

$$serializableNT(s) =_{df} (\forall t).do(Commit(t),s') \sqsubset s \supset \neg transConflictNT^*(t,t,s).$$

Theorem 3 (Isolation) *Suppose $\mathcal{D}$ is a relational theory. Then*

$$\mathcal{D} \models legal(s) \supset serializableNT(s).$$

5 Simulating CNTs

GOLOG, introduced in [12] and enhanced with parallelism in [6] (ConGolog), is a language for defining complex actions in terms of primitive actions axiomatized in the situation calculus. It has the following Algol-like control structures *sequence* ($[\alpha\ ;\ \beta]$; Do action α, followed by action β); *test actions* (p?; Test the truth value of expression p in the current situation); *nondeterministic action choice* ($\alpha \mid \beta$; Do α or β); *nondeterministic argument choice* ($(\pi\ x)\alpha$; pick any value for x, and for that value of x, do action α); *conditionals* and *loops*; and *procedures*. The following are ConGolog constructs for expressing parallelism: *concurrency* ($[\alpha \parallel \beta]$; Do α and β in parallel); *concurrent iteration* ($\alpha^{||}$; Do α zero or more times in parallel).

In [6], a single-steps semantics of GOLOG programs is introduced. Single-steping a given program δ means that δ may perform one step in situation s, ending up in situation s', where a further program δ', a chunk of δ, remains to be executed. This process goes on until no chunk remains. Finally, to process programs, a ternary relation $Do(prog, S, s')$ is introduced with the intuitive meaning that s' is one of the situations reached by evaluating the GOLOG program $prog$, beginning in a given situation S. Notice that we need to modify a Do-based interpreter described in [6] to accommodate non-Markovian features and generate only legal logs.

Now we express a transaction behaviour for our Debit/Credit example as a set of procedures. For brievety, we only give a sample main procedure; this is enclosed between a $Begin(t)$ and an $End(t)$ actions to enforce ACID properties; it spawns subtransactions:

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

Here, the procedure $processReq(t, tid, aid, req)$ is used by the transaction t to process a pending debit or credit request req from a user with account aid at teller tid. Notice that a formula ϕ in a test $\phi?$ is in fact a situation suppressed formula whose situation argument is restored at run-time by the interpreter. Notice also the use of parallel iteration in the last procedure; this spawns a new parallel child transaction for each account that emitted a request, but has not yet been served. The action $report(aid)$ is used to indicate that a request emitted by the owner of the account aid has been granted. These requests are registered in the predicate $requested(aid, req)$. Finally, the following successor state axiom is used for synchronization:

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

Now we can simulate the program, say $processTrans(T)$, 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 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.

6 Discussion

ACTA ([5]) is a framework similar to ours. It allows to specify effects of transactions on objects and on other transactions. In fact, we use the same building blocks for ATMs as those used in ACTA. However, the reasoning capability of the situation calculus exceeds that of ACTA for the following reasons: (1) the database log is a first class citizen of the situation calculus, and the semantics of all external actions are defined with respect to constraints on this log; ACTA does not quantify over histories, so it has no straightforward way of expressing closed form formulas involving histories. (2) Our approach goes far beyond ACTA as it is an implementable specification, thus allowing one to automatically check many properties of the specification using an interpreter. Finally, (3) although ACTA deals with the dynamics of the database, it is not explicitly formulated as a logic for actions.

In [3], Bertossi *et al.* extend Reiter's specification of database updates to transactions. In fact, the idea of using external actions for flat transactions in the situation calculus was first introduced in [3], as was the the axiomatization of the notion of consistency verification at the transaction end. Our approach, however, provides theories that are explicitly non-Markovian; it deals with ACID properties, and goes beyond flat transactions to account for ATMs which are much more complex. Notice that our non-Markovian formalization permits a formalization of ATMs that is not verbose; this would not be the case if one appeals to a transformation of non-Markovian actions into Markovian ones (See [1] for such a transformation).

Transaction Logic ([4]) and *Statelog* ([13]) are languages for database state change that include a clean model theory. However, these approaches, unlike the situation calculus, do not view elementary updates as first order terms; they appeal to special purpose semantics to account for database transactions; finally, we are not sure of their generality to be used for modeling any given transaction model or "inventing" a new one from scratch at a sufficiently high level as is the case in ACTA and the situation calculus.

Thus far, we have given axioms that accommodate a complete initial database state. This, however, is not a requirement of the theory we are presenting. Therefore our account could, for example, accommodate initial databases with null values, open initial database states, initial databases accounting for object orientation, or initial semistructured databases.

On-going work extending our framework includes: accounting for some of the most recent ATMs, for example those reported in [10], implementing the specifications of significant ATMs, proving the correctness of the approach, accommodating initial semistructured databases, and introducing active rules ([11]).

Acknowledgments

We thank R. Reiter, A. Gabaldon, and J. Pinto for helpful discussions. Ray suggested the link to an explicit non-Markovian account. Thanks also to the anonymous referees for insightful comments. We gratefully mention the support by NSERC, IRIS (Institute for Robotics and Intelligent Systems), and ITRC (Information Technology Research Centre of Ontario).

References

[1] M. Arenas and L. Bertossi. Hypothetical temporal queries in databases. *Proceedings of the 5th KRDB*, pages 4.1–4.8, 1998.

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

[3] 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.

[4] A. Bonner and M. Kifer. Transaction logic programming. Tech. report, Univ. of Toronto, 1992.

[5] P.K. Chrysanthis. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, 1991.

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

[7] Ahmed K. Elmagarmid. *Database transaction models for advanced applications*. Morgan Kaufmann, San Mateo, CA, 1992.

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

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

[10] S. Jajodia and L. Kerschberg. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, Boston, 1997.

[11] I. Kiringa. *A Formal Account of Relational Active Databases in the Situation Calculus*. PhD thesis, Computer Science, University of Toronto, Toronto, forthcoming.

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

[13] B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. Technical Report Jun20-1, Technical Univ. of Munich, June 1996.

[14] N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. A theory of atomic transactions. In M. Gyssens, J. Parendaens, and D. Van Gucht, editors, *Proceedings of the Second International Conference on Database Theory*, pages 41–71, Berlin, 1988. Springer Verlag. LNCS 326.

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

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

[17] G. Weikum and H.J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 516–553, San Mateo, CA, 1992. Morgan Kaufmann.