# Simulation of Advanced Transaction Models Using GOLOG[*]

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

**Abstract**

We propose a logical framework for describing, reasoning about, and simulating transaction models that relax some of the ACID (Atomicity-Consistency-Isolation-Durability) properties of classical transactions. Such extensions, usually called *advanced transaction models* (ATMs), have been proposed for dealing with new 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 specify an ATM as a suitable non-Markovian theory of the situation calculus, and its properties, including the relaxed ACID properties, as formulas of the same calculus. We use our framework to formalize classical transactions and closed nested transactions. We first formulate each ATM and its properties as a theory of a certain kind and formulas of the situation calculus, respectively. 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 that the known properties of the ATM, including the relaxed ACID constraints, are properties of legal logs and logical consequences of the theory corresponding to that ATM. 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.

## 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 $Rollback$ ([10]). A transaction makes the results of its operations durable when nothing goes wrong before its normal end by executing a $Commit$ operation, upon which the database cannot be rolled back. Should anything go wrong before the commitment, the transaction rolls the database back to the state before beginning.

Various transaction models have been proposed to extend the classical flat transactions by relaxing some of the ACID properties (see [8],[11] for a collection of the best examples of these models). Such extensions, generally called *advanced transaction models* (ATMs), are proposed for dealing with new applications involving long-lived, endless, and cooperative activities. The ATMs aim at improving the functionality and the performance of the new applications.

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

[*] An extended abstract reporting on this work appears in *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB-2001), Rome, September, 2001.*

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 specify ATMs, simulate these, specify their properties, and reason about these properties. Thus far, ACTA ([6],[5]) seems to our knowledge the only framework addressing these questions at a high level of generality. In ACTA, a first order logic-like language is used to capture the semantics of any ATM.

In this paper, we address the problem of specifying database transactions at the logical level using the situation calculus ([21]). Our approach appeals to non-Markovian theories([9]), 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; the properties of the ATM, including the relaxed ACID properties, are expressed as formulas of the same calculus that logically follow from the basic relational theory. We illustrate our framework by formalizing classical transactions ([10]) and closed nested transactions ([22]). We first formulate each transaction model and its properties as a basic relational theory and formulas of the situation calculus, respectively. 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 that the known properties of the transaction model, including the relaxed ACID constraints, are properties of legal logs and logical consequences of the basic relational theory corresponding to that transaction model. 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.

Ours is an ongoing work whose main contributions reported in this paper can succintly be summarized as follows:

1. We construct logical theories called basic relational theories to formalize ATMs along the tradition set by the ACTA framework ([5]); basic relational theories are non-markovian theories in which one may explicitly refer to all past states, and not only the to the previous one. They provide the formal semantics of the corresponding ATMs. They are an extension of the classical relational theories of [19] to the database transaction setting.

2. We extend the notion of *legal database logs* introduced in [20] to accomodate transactional actions such as $Begin$, $Commit$, etc. These logs are first class citizen of the logic and properties of the ATM are expressed as formulas of the situation calculus that logically follow from the basic relational theory representing that ATM.

3. Our approach goes far beyond constructing logical theories, as it provides one with an implementable specification, thus allowing one to automatically check many properties of the specification using an interpreter. Our implementable specifications are written in an extension of GOLOG that includes parallelism ([7]). We specify an interpreter for running these specifications and show that this interpreter generates only legal logs.

## 2 Logical Foundations

We use a *basic relational language*, which is a fragment of the situation calculus ([21],[9]) 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, $situations$ are first order terms denoting finite sequences of actions, and $objects$ 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.

The language has an alphabet with variables and a finite number of constants for each sort, a finite number of function symbols called *action functions* (e.g., $a\_del(accid, branchid, accbal, tellerid, t)$), a finite number of function symbols called *functional fluents*, a finite number of function symbols called *situation independent functions*, a finite number of predicate symbols called *relational fluents* (e.g., $accounts(accid, branchid, accbal, tellerid, t, s)$), and a finite number of predicate symbols called *situation independent predicates*. Situations are represented using a binary function symbol $do$: $do(a, s)$ denotes the sequence resulting from adding the action $a$ to the sequence $s$. There is a distinguished constant $S_0$ denoting the initial situation; $S_0$ stands for the empty action sequence. 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 basic relational languages whose only primitive update operations correspond to insertion or deletion of tuples into relations. For each such relation $F(\vec{x}, t, s)$, where $\vec{x}$ is a tuple of objects, $t$ is a transaction argument, and $s$ is a situation argument, a *primitive internal action* is a parameterized primitive action of the situation calculus of the form $F\_ins(\vec{x}, t)$ 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 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. The argument $t$ is a unique transaction identifier.

A dynamic domain is axiomatized in the situation calculus with non-Markovian axioms which describe *how* and under what *conditions* the domain is changing or not changing as a result of performing actions. Such axioms are called basic action theory in [21]. They comprise the following: 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. Finally, by convention in this paper, a free variable will always be implicitly bound by a prenex universal quantifier. Basic action theories of [21] are capturing Markovian control. they have been extended to non-Markovian control in [9].

## 3 The Specification Framework

In [6], five building blocks for ATMs are identified: *history*, intertransaction *dependencies*, *visibility* of operations on database objects, *conflict* between operations, and *delegation* of responsibility for objects visible to a transaction. We now show how these building blocks are represented in the situation calculus.

In the situation calculus, the history of [6] corresponds to the log. We extend the basic action theories of [21] to include a specification of relational database transactions, by giving action precondition axioms for external actions such as $Begin(t)$, $End(t)$, $Commit(t)$, $Rollback(t)$, $Spawn(t, t')$, etc. $Commit(t)$ and $Rollback(t)$ are coercive actions that must occur whenever they are possible. 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 sublogs of the current log are visible; which visible sublogs are delegated to the transactions is expressed implicitely in successor state axioms.

A useful concept that underlies most of the ATMs is that of responsibility over changes operated on data items. For example, in a nested transaction, a parent transaction will take responsibility of changes done by any of its committed children. The only way we can keep track of those reponsibilities 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 of the form $responsible(t, a', do(a, s)) \equiv \Phi_{tm}(t, a, a', s)$, where $\Phi_{tm}(t, a, a', s)$ is a transaction model-dependent first order formula whose only free variables are among $t, a, a',$, and $s$. For example, in the flat transactions, we will have the following, simple axiom:

$$responsible(t, a, s) \equiv (\exists a')a = a'(\vec{x}, t);$$

i.e., each transaction is considered responsible for any action whose last argumment bears its name.

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

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

here, $\mathcal{F}$ is the set of fluents of the 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 $s$;

$$\begin{aligned} & transConflict(t, t', do(a, s)) \equiv t \neq t' \wedge responsible(t', a, s) \wedge \\ & \quad (\exists a', s')[responsible(t, a', s) \wedge do(a', s') \sqsubset s \wedge updConflict(a', a, s)] \vee \\ & \quad transConflict(t, t', s) \wedge \neg termAct(a, t); \end{aligned} \qquad (1)$$

i.e., transaction $t$ conflicts with transaction $t'$ in the log $s$ iff $t'$ executes an internal action $a$ after $t$ has executed an internal action $a'$ that conflicts with $a$ in the log $s$. Notice that we define $updConfilct(a, a', s)$ in terms of performing action $a$ and action $a'$ one immediately after the other and vice-versa; in the definition of $transConflict(t, t', s)$, however, we allow action $a'$ to be executed long before action $a$. This does not mean that actions that are performed between $a'$ and $a$ are irrelevant with respect to update conflicts. Rather, (3) just means that actions $a$ and $a'$ conflicts whenever executing one immediately after the other *would* results in a discrepancy in the truth value of at leat one of the relational fluents; and (1) allows for the possibility of other update conflicts arising between $a'$ and other actions before the execution of $a$.

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 axiomatizer must provide a successor state axiom for this fluent depending on the application.

The visibility of portions of the log is characterized by a transaction model-specific fluent $visible(t, s)$, which intuitively means that the transaction $t$ sees the log $s$. In general, it has the form $visible(t, s) =_{df} \mathcal{H}(t, s)$, where $\mathcal{H}(t, s)$ is a condition on the log $s$ depending on the transaction $t$. In the transaction models formalized this paper, we have $visible(t, s) \equiv true$. In the sequel, we will no longer deal with this aspect.

The *second dimension* of the situation calculus framework is made of dependencies between transactions. All the dependencies expressed in ACTA ([6]) can also be expressed in the situation calculus. As an example, we have:

**Commit Dependency** of $t$ on $t'$

$$do(Commit(t), s) \sqsubset s^* \supset$$
$$[do(Commit(t'), s') \sqsubseteq s^* \supset do(Commit(t'), s') \sqsubset do(Commit(t), s)];$$

i.e., If $t$ commits in a log $s^*$, then, whenever $t'$ also commits in $s^*$, $t'$ commits before $t$.

**Strong Commit Dependency** of $t$ on $t'$

$$(\exists s')do(Commit(t'), s') \sqsubset s^* \supset (\exists s)do(Commit(t), s) \sqsubseteq s^*;$$

i.e., If $t'$ commits in a log $s^*$, then $t$ must also commit in that log.

**Rollback Dependency** of $t$ on $t'$

$$(\exists s')do(Rollback(t'), s') \sqsubset s^* \supset (\exists s)do(Rollback(t), s) \sqsubseteq s^*;$$

i.e., If $t'$ rolls back in a log $s^*$, then $t$ must also roll back in that log.

**Weak Rollback Dependency** of $t$ on $t'$

$$do(Rollback(t'), s') \sqsubset s^* \supset$$
$$\{(\forall s)[s \sqsubset s^* \wedge do(Commit(t), s) \not\sqsubseteq do(Rollback(t'), s')] \supset$$
$$(\exists s'')do(Rollback(t), s'') \sqsubseteq s^*\};$$

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

As we shall see below, all these dependencies are properties of legal database logs of various transaction models.

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 its external actions (*external dependencies*) and those generated by the execution of its internal actions (*internal dependencies*). As an example, in the nested transaction model, we have the following successor state axiom for $wr\_dep(t, t', s)$:

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

This says that a weak rollback dependency of $t$ on $t'$ arises in $do(a, s)$ when either $a$ is the action of $t$ spawning $t'$, or that dependency existed already in $s$ and neither $t$ nor $t'$ terminated with the action $a$.

## 4 Flat Transactions

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

A sequence of database actions is a *flat transaction* iff it is a sequence $[a_1, \ldots, a_n]$, where the $a_1$ must be $Begin$, 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, as before, the argument $t$ is a unique identifier for the atomic transaction. Flat transactions can be sequenced or run in parallel. Notice that we do not introduce a term of 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 6. We refer to transactions by their names that are of sort *object*.

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:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2, \tag{2}$$
$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s), \tag{3}$$
$$\neg(s \sqsubset S_0), \tag{4}$$
$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'. \tag{5}$$

They characterize database logs as finite sequences of updates. Notice that the second axiom is a second-order induction axiom; the third and fourth axioms characterize the subsequence relation $\sqsubset$.

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

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

$$Poss(A(\vec{x}, t), s) \equiv (\exists t')\Pi_A(\vec{x}, t', s) \wedge IC^e(do(A(\vec{x}, t), s)) \wedge running(t, s). \tag{6}$$

Here, $\Pi_A(\vec{x}, t, s)$ is a formula with free variables among $\vec{x}, t$, and $s$. These axioms characterize the preconditions of the update $A$; $IC^e(s)$ and $running(t, s)$ are defined as follows:

$$IC^e(s) =_{df} \bigwedge_{IC \in \mathcal{IC}_e} IC(s). \tag{7}$$

$$\begin{aligned} running(t, s) =_{df}\ & (\exists s').do(Begin(t), s') \sqsubseteq s \wedge \\ & (\forall a, s'')[do(Begin(t), s') \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)]. \end{aligned} \tag{8}$$

In a banking Credit/Debit example formalized below, the following states that it is possible to insert a tuple 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.

$$\begin{aligned} & Poss(t\_delete(tid, tbal, t), s) \equiv (\exists t')teller(tid, tbal, t', s) \wedge \\ & \quad IC^e(do(t\_delete(tid, tbal, t), s)) \wedge running(t, s). \end{aligned} \tag{9}$$

**Successor State Axioms**. These have the syntactic form

$$\begin{aligned} & F(\vec{x}, t, do(a, s)) \equiv (\exists \vec{t'})\Phi_F(\vec{x}, a, \vec{t'}, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & \quad (\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s), \end{aligned} \tag{10}$$

where $\Phi_F(\vec{x}, a, \vec{t}, s)$ is a formula with free variables among $\vec{x}, a, \vec{t}, s$. There is one such axiom for each relational fluent $F$, and $restoreBeginPoint(F, \vec{x}, t, s)$ is defined as follows:

$$\begin{aligned} & restoreBeginPoint(F, \vec{x}, t, s) =_{df} \\ & [(\exists a^*, s', s^*, t').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \wedge writes(a^*, F, t) \wedge F(\vec{x}, t', s')] \vee \\ & [(\forall a^*, s^*, s').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \supset \neg writes(a^*, F, t)] \wedge (\exists t')F(\vec{x}, t', s). \end{aligned} \tag{11}$$

Intuitively, $restoreBeginPoint(F, \vec{x}, t, s)$ means that the system restores the value that the fluent $F$ with arguments $\vec{x}$ had before the execution of the $Begin$ action of the transaction $t$ 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. In the banking example, the following states that the tuple $(tid, tbal)$ will be in the $teller$ 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 $teller$ 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. In the case the operation $a$ is rolling the database back, the $tellers$ relation will get a value according to the logic of (11).

$$
\begin{aligned}
&tellers(tid, tbal, t, do(a, s)) \equiv ((\exists t_1)a = t\_insert(tid, tbal, t_1) \vee \\
&\quad (\exists t_2)tellers(tid, tbal, t_2, s) \wedge \neg(\exists t_3)a = t\_delete(tid, tbal, t_3)) \wedge \neg(\exists t')a = Rollback(t') \vee \\
&\quad (\exists t').a = Rollback(t') \wedge restoreBeginPoint(tellers, (tid, tbal), t', s).
\end{aligned}
$$

**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:

$$Poss(Begin(t), s) \equiv \neg(\exists s')do(Begin(t), s') \sqsubseteq s, \tag{12}$$

$$Poss(End(t), s) \equiv running(t, s), \tag{13}$$

$$
\begin{aligned}
&Poss(Commit(t), s) \equiv (\exists s').s = do(End(t), s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\
&\quad (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s],
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
&Poss(Rollback(t), s) \equiv (\exists s')[s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee \\
&\quad (\exists t', s'')[r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubseteq s].
\end{aligned} \tag{15}
$$

**Dependency axioms**. These are transaction model-dependent axioms of the form

$$dep(t, t', do(a, s)) \equiv \mathcal{C}(t, t', a, s), \tag{16}$$

where $\mathcal{C}(t, t', a, s)$ is a condition involving the conflict relation between internal actions of any two transactions $t$ and $t'$, and $dep(t, t', s)$ is one of the dependency predicates $c\_dep(t, t', s)$, $sc\_dep(t, t', s)$, etc. These axioms are used to capture the notion of *recoverability*, *avoiding cascading rollbacks*, etc, of the classical concurrency control theory ([2]). For example, to achieve recoverability, avoid cascading rollbacks, the following axioms are used, respectively:

$$r\_dep(t, t', s) =_{df} transConflict(t, t', s), \tag{17}$$

$$sc\_dep(t, t', s) =_{df} readsFrom(t, t', s). \tag{18}$$

The first axiom says that a transaction conflicting with another transaction generates a rollback dependency, and the second says that a transaction reading from another transaction generates a strong commit dependency.

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

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

$$(\forall \vec{x}, t).F(\vec{x}, t, S_0) \equiv \vec{x} = \vec{C}^{(1)} \vee \ldots \vee \vec{x} = \vec{C}^{(r)}, \tag{19}$$

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. Axioms of the form (19) say that our theories accommodate a complete initial database state, which is commonly the case in relational databases as unveiled in [19]. 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; moreover, it has the practical advantage of facilitating rapid prototyping of the ATMs using Prolog which embodies negation by failure, a notion close to the completion axioms used here.

One striking feature of our axioms is the use of the predicate $\sqsubset$ on the right hand side of action precondition axioms and successor state axioms. That is, they are capturing the notion of a situation being located in the past relative to the current situation which we express with the predicate $\sqsubset$ in the situation calculus. Thus they are capturing non-Markovian control. 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 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*:

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

We constrain legal logs to include these mandatory system actions, as well as the requirement that all actions in the log be possible:

$$\begin{aligned} &legal(s) =_{df} (\forall a, s^*)[do(a,s^*) \sqsubseteq s \supset Poss(a,s^*)] \wedge \\ &\quad (\forall a', a'', s', t)[systemAct(a',t) \wedge responsible(t,a') \wedge \\ &\qquad responsible(t,a'') \wedge Poss(a',s') \wedge do(a'',s') \sqsubset s \supset a' = a'']. \end{aligned} \tag{20}$$

Simple properties such as well-formedness of atomic transactions ([16]) can be formulated and proven.

**Theorem 1** **(Well-Formedness)** *Suppose $\mathcal{D}$ is a basic relational theory. Then no transaction may commit and then roll back, and conversely; i.e.,*

$$\begin{aligned} &\mathcal{D} \models legal(s) \supset \\ &\quad (\forall s')\{[do(Commit(t),s') \sqsubset s \supset \neg(\exists s'')do(Rollback(t),s'') \sqsubset s] \wedge \\ &\qquad [do(Rollback(t),s') \sqsubset s \supset \neg(\exists s'')do(Commit(t),s'') \sqsubset s]\}. \end{aligned}$$

These properties are similar to the fundamental axioms, applicable to all transactions, of [6]. They rule out all the ill-formed transactions such as

$[Begin(t), a\_ins(A_1, B_1, -1000, T_1),$
$\quad Commit(t), a\_del(A_1, B_1, -1000, T_1), Rollback(t)]$, etc.

**Theorem 2** *Suppose $\mathcal{D}$ is a basic relational theory. Then any legal log satisfies the strong commit and rollback dependency properties; i.e.,*

$$\begin{aligned} &\mathcal{D} \models legal(s) \supset \\ &\quad (\forall t,t')\{sc\_dep(t,t',s) \supset [(\exists s')do(Commit(t'),s') \sqsubset s \supset (\exists s^*)do(Commit(t),s^*) \sqsubseteq s] \wedge \\ &\qquad c\_dep(t,t',s) \supset [(\exists s')do(Rollback(t'),s') \sqsubset s \supset (\exists s^*)do(Rollback(t),s^*) \sqsubset s]\}. \end{aligned}$$

Now we turn to the ACID properties, which are the most important properties of flat transactions.

**Theorem 3** *(***Atomicity***) Suppose $\mathcal{D}$ is a relational theory. Then for every relational fluent $F$*

$$\begin{aligned} &\mathcal{D} \models legal(s) \supset \\ &\quad (\forall t, s_1, s_2)\{do(Begin(t),s_1) \sqsubset do(a,s_2) \sqsubset s \wedge \\ &\qquad (\exists a^*, s^*)[do(Begin(t),s_1) \sqsubset do(a^*,s^*) \sqsubset do(a,s_2) \wedge writes(a^*,F,t)] \supset \\ &\qquad\quad [(a = Rollback(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_1))) \wedge \\ &\qquad\quad (a = Commit(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_2)))]\}. \end{aligned}$$

This says that rolling back restores any modified fluent to the value it had just before the last $Begin(t)$ action, and committing endorses the value it had in the situation just before the $Commit(t)$ action.

**Theorem 4 (Consistency)** *Suppose $\mathcal{D}$ is a relational theory. Then All integrity constraints are satisfied at committed logs; i.e.,*

$$\mathcal{D} \models legal(s) \supset \{do(Commit(t), s') \sqsubseteq s \supset \bigwedge_{IC \in \mathcal{IC}_v \cup \mathcal{IC}_e} IC(do(Commit(t), s'))\}.$$

**Theorem 5** *$\mathcal{D}$ is satisfiable iff $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{IC}[S_0]$ is.*[1] *In other words, provided the constraints are consistent with the initial database state and unique names for actions, then the entire relational theory is satisfiable, and conversely.*

Some properties of transactions need the notions of committed and rolled back updates. With the predicates $committed(t, s)$ and $rolledBack(t, s)$, we express these notions in the situation calculus using the following axioms:

$$committed(a', do(a, s)) \equiv (\exists t).responsible(t, a') \wedge a = Commit(t) \vee committed(a', s);$$
$$rolledBack(a', do(a, s)) \equiv (\exists t).responsible(t, a') \wedge a = Rollback(t) \vee rolledBack(a', s).$$

**Theorem 6 (Durability)** *Suppose $\mathcal{D}$ is a relational theory. Then whenever an update is committed or rolled back by a transaction, another transaction can not change this fact:*

$$\begin{aligned}\mathcal{D} \models legal(s) \supset & \\ & \{do(Rollback(t), s') \sqsubseteq s \wedge \neg responsible(t, a) \supset \\ & \quad [Committed(a, s') \equiv Committed(a, do(Rollback(t), s'))] \wedge \\ & \quad [rolledBack(a, s') \equiv rolledBack(a, do(Rollback(t), s'))].\end{aligned}$$

**Definition 1 (Serializability)**

$$\begin{aligned} & transConflict^*(t, t', s) =_{df} (\forall C)[(\forall t)C(t, t, s) \wedge \\ & \qquad (\forall s, t, t', t'')[C(t, t'', s) \wedge transConflict(t'', t', s) \supset C(t, t', s)] \supset C(t, t', s)], \\ & serializable(s) =_{df} (\forall t).do(Commit(t), s') \sqsubset s \supset \neg transConflict^*(t, t, s).\end{aligned}$$

**Theorem 7 (Isolation)** *Suppose $\mathcal{D}$ is a relational theory. Then*

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

## 5 Closed Nested Transactions

Nested transactions ([17]) are the best known example of ATMs. A 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 commits only if its parent has committed. 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. This version of nested transactions is called closed because of this inability of subtransactions to durably commit independently of the outcome of the root transaction ([22]).

A root transaction $t$ is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where $a_1$ 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 $Begin(t)$, $Commit(t)$, and $Rollback(t)$, but including $Spawn(t, t')$, $Rollback(t')$, and

[1]Here, $\mathcal{D}_{IC}[S_0]$ is the set $\mathcal{D}_{IC}$ relativized to the situation $S_0$.

$Commit(t')$, with $t \neq t'$. A child transaction $t$ is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where $a_1$ must be $Spwan(t', t)$, 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 $Spawn(t, t')$, $Commit(t)$, and $Rollback(t)$, but including $Spawn(t^*, t^{**})$, $Rollback(t^{**})$, and $Commit(t^{**})$, with $t \neq t^{**}$. We capture the typical relationships that hold between transactions in the hierarchy of a nested transaction with the fluents $transOf(t, a, s)$, $parent(t, t', s)$ and $ancestor(t, t', s)$, which are introduced in the following successor state axiom and abbreviation, respectively:

$$transOf(t, a, s) \equiv (\exists a').a = a'(\vec{x}, t), \tag{21}$$

$$\begin{aligned} &parent(t, t', do(a, s)) \equiv a = Spawn(t, t') \vee \\ &\qquad parent(t, t', s) \wedge \neg termAct(a, t) \wedge \neg termAct(a, t'), \end{aligned} \tag{22}$$

$$\begin{aligned} &ancestor(t, t', s) =_{df} (\forall A)[(\forall t)A(t, t, s) \wedge \\ &\qquad (\forall s, t, t', t'')[A(t, t'', s) \wedge parent(t'', t', s) \supset A(t, t', s)] \supset A(t, t', s)]. \end{aligned} \tag{23}$$

Responsibility over actions that are executed and conflicts between transactions are specified with the following axioms:

$$\begin{aligned} &responsible(t, a', do(a, s)) \equiv transOf(t, a', s) \wedge \neg(\exists t^*)parent(t, t^*, s) \vee \\ &\quad (\exists t^*)[parent(t, t^*, s) \wedge a = Commit(t^*) \wedge responsible(t^*, a')] \vee \\ &\quad responsible(t, a', s) \wedge \neg termAct(a, t), \end{aligned} \tag{24}$$

$$\begin{aligned} &transConflictNT(t, t', do(a, s)) \equiv t \neq t' \wedge responsible(t', a, s) \wedge \\ &(\exists a', s')[responsible(t, a', s) \wedge updConflict(a', a, s) \wedge do(a', s') \sqsubset s \wedge \\ &\neg responsible(t, a, s) \wedge running(t', s) \wedge ((\exists t'')parent(t, t'', s) \supset \neg ancestor(t, t', s)) \vee \\ &transConflictNT(t, t', s) \wedge \neg termAct(a, t); \end{aligned} \tag{25}$$

Intuitively, (25) means that transaction $t$ conflicts with transaction $t'$ in the log $s$ iff $t$ and $t'$ are not equal, internal actions they are responsible for are conflicting in $s$, $t$ 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. Due to the presence of the new external action $Spawn$, we need to redefine $running(t, s)$ as follows:

$$\begin{aligned} &running(t, s) =_{df} (\exists s').\{do(Begin(t), s') \sqsubseteq s \wedge \\ &\quad (\forall a, s'')[do(Begin(t), s') \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)] \vee \\ &\quad (\exists t').do(Spawn(t', t), s') \sqsubseteq s \wedge \\ &\quad (\forall a, s'')[do(Spawn(t', t), s') \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)]\}. \end{aligned} \tag{26}$$

Now the external actions of nested transactions have the following precondition axioms:

$$\begin{aligned} &Poss(Begin(t), s) \equiv \neg(\exists t')parent(t', t, s) \wedge \\ &\qquad [s = S_0 \vee (\exists s', t').t \neq t' \wedge do(Begin(t'), s') \sqsubset s], \end{aligned} \tag{27}$$

$$\begin{aligned} &Poss(Spawn(t, t'), s) \equiv t \neq t' \wedge \\ &\qquad (\exists s', t'')[do(Begin(t), s') \sqsubset s \vee do(Spawn(t'', t), s') \sqsubset s], \end{aligned} \tag{28}$$

$$Poss(End(t), s) \equiv running(t, s), \tag{29}$$

$$\begin{aligned} &Poss(Commit(t), s) \equiv (\exists s').s = do(End(t), s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\ &\qquad (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s] \wedge \\ &\qquad (\forall t')[c\_dep(t, t', s) \wedge \neg(\exists s^*)do(Rollback(t'), s^*) \sqsubseteq s \supset \\ &\qquad\qquad (\exists s')do(Commit(t'), s') \sqsubset s)], \end{aligned} \tag{30}$$

$$\begin{aligned}Poss(Rollback(t),s) \equiv (\exists s').s = do(End(t),s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \vee \\ (\exists t', s'').r\_dep(t,t',s) \wedge do(Rollback(t'),s'') \sqsubset s' \vee \\ (\exists t', s^*).wr\_dep(t,t',s) \wedge do(Rollback(t'),s^*) \sqsubset s \wedge \\ \neg(\exists s^{**}) do(Commit(t),s^{**}) \sqsubset do(Rollback(t'),s^*),\end{aligned} \tag{31}$$

Dependency axioms characterizing the fluents $r\_dep$, $c\_dep$, $sc\_dep$, and $wr\_dep$ are:

$$r\_dep(t,t',s) \equiv transConflictNT(t,t',s), \tag{32}$$

$$sc\_dep(t,t',s) \equiv readsFrom(t,t',s), \tag{33}$$

$$\begin{aligned}c\_dep(t,t',do(a,s)) \equiv a = Spawn(t,t') \vee \\ c\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'),\end{aligned} \tag{34}$$

$$\begin{aligned}wr\_dep(t,t',do(a,s)) \equiv a = Spawn(t',t) \vee \\ wr\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t').\end{aligned} \tag{35}$$

As an example of what they mean, the last axiom says that a transaction spawning another transaction generates a weak rollback dependency of the later one on the first one, and this dependency ends when either transactions execute a terminating action.

The successor state axioms for nested transactions are of the form:

$$\begin{aligned}F(\vec{x},t,do(a,s)) \equiv (\exists \vec{t'})\Phi_F(\vec{x},a,\vec{t'},s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ [(\exists t'').a = Rollback(t'') \wedge \neg(\exists t^*)parent(t^*,t'',s) \wedge restoreBeginPoint(F,\vec{x},t'',s)] \vee \\ [(\exists t'').a = Rollback(t'') \wedge (\exists t^*)parent(t^*,t'',s) \wedge restoreSpawnPoint(F,\vec{x},t'',s)],\end{aligned} \tag{36}$$

one for each fluent of the relational language. Here $\Phi_F(\vec{x},a,\vec{t},s)$ is a formula with free variables among $\vec{x},a,\vec{t}$, and $s$; $restoreBeginPoint(F,\vec{x},t,s)$ is defined in (11), and $restoreSpawnPoint(F,\vec{x},t,s)$ is the following abbreviation:

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

A basic relational theory for nested transactions is defined as in Section 4, but where the relational language includes $Spawn(t,t')$ as a further action, and the axioms (27) – (28) replace axioms (12) – (15), the axioms (32) – (35) replace the axioms (17) – (18), and the set $\mathcal{D}_{ss}$ is a set of successor state axioms of the form (36). All the other axioms of Section 4 remain unchanged.

Now we state some of the properties of nested transactions as an illustration of how such properties are formulated in the situation calculus. Similarly to Theorem 2, we can show that a basic relational theory for nested transactions logically implies the commit and weak rollback dependency properties.

**Theorem 8** *(***Atomicity of Nested Transactions***) Suppose $\mathcal{D}$ is a relational theory for nested transactions. Then for every relational fluent $F$*

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

**Definition 2 (Serializability of Nested Transactions)**

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

**Theorem 9 (Isolation of Nested Transactions)** *Suppose $\mathcal{D}$ is a relational theory for nested transactions. Then*

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

## 6 Simulating ATMs

GOLOG, introduced in [13] and enhanced with parallelism in [7] (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 *Sequence* ($[\alpha\,;\,\beta]$; Do action $\alpha$, followed by action $\beta$); *Test actions* ($p?$; Test the truth value of expression $p$ in the current situation); *Nondeterministic action choice* ($\alpha \mid \beta$; Do $\alpha$ or $\beta$); *Nondeterministic choice of arguments* ($(\pi\ x)\alpha$; nondeterministically pick a value for $x$, and for that value of $x$, do action $\alpha$); *Conditionals* and *while* loops; and *Procedures,* including recursion. $nil$ represent the empty program. The following are ConGolog constructs for expressing parallelism: *Concurrency* ($[\alpha\ \ ||\ \ \beta]$; Do $\alpha$ and $\beta$ in parallel); *Concurrent iteration* ($\alpha^{||}$; Do $\alpha$ zero or more times in parallel). The purpose of this section is to show how GOLOG programs are used to capture transactions and how the semantics of this programs is used to simulate the ATMs.

### 6.1 Well-formed GOLOG Programs

GOLOG syntax is built using consructs that suppress any reference to situations in which test are evaluated. These will be restore at run time by the GOLOG interpreter. The following is a restriction to relational languages of a similar definition given in [18].

**Definition 3** *Suppose $\mathfrak{R}$ is a relational language. Then the* situation-suppressed *terms of $\mathfrak{R}$ are given by:*

1. *Any variable or constant of sort* actions, objects, *or* situations *of $\mathfrak{R}$ is a situation-suppressed term.*

2. *If $a$ is an action function symbol of $\mathfrak{R}$ other than $S_0$ and $do$, and $t_1, \cdots, t_n$ are variables or constants of $\mathfrak{R}$, then $a(t_1, \cdots, t_n)$ is a situation-suppressed term.*

3. *For any situation term $\sigma$ and any action term $a$, $do(a, \sigma)$ is a situation-suppressed term.*

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

1. *Whenever $t, t'$ are situation-suppressed terms of the same sort, then $t = t'$ is a situation-suppressed formula. Notice that a situation-suppressed formula here, contrary to [18], may mention an equality between terms of sort* situations.

2. *Whenever $t$ is a situation-suppressed term of sort* actions, *then $Poss(t)$ is a situation-suppressed formula.*

3. *Whenever $F$ is an $n+1$-ary relational fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are situation-suppressed terms of sort* objects, *then $F(t_1, \cdots, t_n)$ is a situation-suppressed formula.*

4. *Whenever $P$ is an $m$-ary situation independant predicate of $\Re$ and $t_1, \cdots, t_n$ are situation-suppressed terms of sort* objects, *then $P(t_1, \cdots, t_n)$ is a situation-suppressed formula.*

5. *Whenever $t$ and $t'$ are situation terms of $\Re$, then $t \sqsubset t'$ is a situation-suppressed formula.*

6. *Are $\phi$ and $\psi$ situation-suppressed formulas of $\Re$, 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 situation-suppressed formulas are first order and may still mention situation terms, but never as last argument of relational fluents; therefore situation-suppressed 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 a situation-suppressed formula:

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

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 4 (Well formed GOLOG Programs)** *A GOLOG program has the following syntax:*

$$\begin{aligned}\langle prog\rangle \;::=\; & \langle internal\ action\rangle \mid \langle test\ action\rangle? \mid (\langle prog\rangle; \langle prog\rangle) \mid \\ & (\langle prog\rangle | \langle prog\rangle) \mid (\langle prog\rangle \parallel \langle prog\rangle) \mid (\pi x)\langle prog\rangle \mid \\ & \langle prog\rangle^* \mid (Spawn(t, t'); \langle prog\rangle; End(t')) \mid \langle procedure\ call\rangle \mid \\ & (\textbf{\textit{proc}}\ P_1(\vec{x}_1)\langle prog\rangle\ \textbf{\textit{endProc}}\ ;\ \cdots\ ;\ \textbf{\textit{proc}}\ P_n(\vec{x}_n)\langle prog\rangle\ \textbf{\textit{endProc}}\ ;\ \langle prog\rangle)\end{aligned}$$

*Notice that*

1. $\langle internal\ action\rangle$ *is a situation-suppressed internal action term.*

2. $\langle test\ action\rangle$ *is a situation-suppressed 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 situation-suppressed terms whose sorts match those of the $n$ arguments in the declaration of $P$.*

*A well formed GOLOG program is syntactically defined as follows:*

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

## 6.2 Semantics of GOLOG Programs

With the ultimate goal of handling database transactions, it is appropriate to adopt an operational semantics of GOLOG programs based on a single-step execution of these programs; such a semantics is introduced in ([7]). First, two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that program $\delta$ may perform one step in situation $s$, ending up in situation $s'$, where program $\delta'$ remains to be executed. $Final(\delta, s)$ means that program $\delta$ 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 [7] 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. Notice that equivalences like those above are translating GOLOG constructs into formulas of the situation calculus.

Our definition of $Trans$ differs from that of [7] with respect to the handling of primitive and test actions:

**Definition 5 (Semantics of $Trans$)**

$$\begin{aligned}
&Trans(a,s,a',s') =_{df} Poss(a,s) \wedge a' = nil \wedge \\
&\quad \{(\exists a'',s'',t)[s'' = do(a,s) \wedge systemAct(a'',t) \wedge Poss(a'',s'') \wedge s' = do(a'',s'')] \vee \\
&\quad s' = do(a,s) \wedge [(\forall a'',t)systemAct(a'',t) \supset \neg Poss(a'',s)]\}, \qquad (37)\\
&Trans(\phi?,s,a',s') =_{df} Holds(\phi,s,s') \wedge a' = nil. \qquad (38)
\end{aligned}$$

In the definition 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 (37) 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 (38) says that the test action $\phi?$ may legally be performed one or more steps in the log $s$, ending in log $s'$ where $a'$ remains to be executed iff $\phi$ 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 $\delta$ in situation $s$ is the task of finding a situation $s'$ such that there is a final configuration $(\delta', s')$, for some remaining program $\delta'$, after performing a couple of transitions from $\delta, s$ to $\delta', s'$. Program execution is captured by using the abbreviation $Do(\delta,s,s')$ ([21]). In the single-step semantics, $Do(\delta,s,s')$ intuitively means that program $\delta$ is single-steped until the remainder of program $\delta$ may terminate in situation $s'$; and $s'$ is one of the logs reached by single-stepinging the program $\delta$, beginning in a given situation $s$. Formally, $Do(\delta,s,s')$ is defined as follows ([7]):

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

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 6** *We use the notation $\phi[s]$ to denote the situation calculus formula obtained from a given formula $\phi$ by restoring the situation argument $s$ in all the fluents (as their last argument) occurring in $\phi$.*

The predicate $Holds(\phi,s,s')$ captures the revised Lloyd-Topor transformations of [21]; these are transformations in the style of Lloyd-Topor([14]), but without its auxilliary predicates. The predicate $Holds(\phi,s,s')$ takes a formula $\phi$ and establish whether it holds in the log $s$ or not. If $\phi$ 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 $\phi$ until we reach literals. We capture this semantics as follows:

**Definition 7 (Semantics of $Holds$)**

$$
\begin{aligned}
&Holds(\phi, s, s') =_{df} \phi[s] \wedge s' = do(\phi, s), \text{ \textit{when} } \phi \text{ \textit{is a fluent literal}},\\
&Holds(\phi, s, s') =_{df} \phi[s] \wedge s' = s, \text{ \textit{when} } \phi \text{ \textit{is a nonfluent literal}},\\
&Holds((\phi_1 \wedge \phi_2), s, s') =_{df} (\exists s'').Holds(\phi_1, s, s'') \wedge Holds(\phi_2, s'', s'),\\
&Holds((\phi_1 \vee \phi_2)?, s, s') =_{df} Holds(\phi_1, s, s') \vee Holds(\phi_2, s, s'),\\
&Holds((\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\phi_1 \vee \phi_2, s, s'),\\
&Holds((\phi_1 \equiv \phi_2), s, s') =_{df} Holds((\phi_1 \supset \phi_2) \wedge (\phi_2 \supset \phi_1), s, s'),\\
&Holds((\forall x)\phi, s, s') =_{df} Holds(\neg(\exists x)\neg\phi, s, s'),\\
&Holds((\exists x)\phi, s, s') =_{df} Holds(\phi, s, s'),\\
&Holds(\neg\neg\phi, s, s') =_{df} Holds(\phi, s, s'),\\
&Holds(\neg(\phi_1 \wedge \phi_2), s, s') =_{df} Holds(\neg\phi_1, s, s') \vee Holds(\neg\phi_2, s, s'),\\
&Holds(\neg(\phi_1 \vee \phi_2), s, s') =_{df} (\exists s'').Holds(\neg\phi_1, s, s'') \wedge Holds(\neg\phi_2, s'', s'),\\
&Holds(\neg(\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\neg(\phi_1 \vee \phi_2), s, s'),\\
&Holds(\neg(\phi_1 \equiv \phi_2), s, s') =_{df} Holds(\neg[(\phi_1 \supset \phi_2) \wedge (\phi_2 \supset \phi_1)], s, s'),\\
&Holds(\neg(\forall x)\phi, s, s') =_{df} Holds((\exists x)\neg\phi, s, s'),\\
&Holds(\neg(\exists x)\phi, s, s') =_{df} \neg Holds(\phi, s, s').
\end{aligned}
$$

Definition 7 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 [7] and why they are needed. Our test actions differ from those of ConGolog ([7]) 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 ATMs. Here, 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 [7]. In other words, in the absence of test actions in the log, the semantics of [7] has no straightforward way to express such things as transaction $T_1$ reads data from transaction $T_2$.

## 6.3 Simulation

We use the GOLOG language as a transaction language for specifying and simulating ATMs at the logical level. To simulate a specific ATM, we first pick the appropriate basic relational theory $\mathcal{D}$ corresponding to that ATM. Then, we write a GOLOG program $T$ expressing the desired transactional behavior. Now simulating the program $T$ amounts to the theorem proving task of establishing the entailment $\mathcal{D} \models (\exists s')\ Do(prog, S_0, s')$, where $S_0$ is the initial, empty log.

A $Do$-based GOLOG interpreter for the situation calculus, written in Prolog, is described in [13]. To run our specification of transactions, we need to modify the GOLOG semantics and interpreter defined in [13] to accommodate the changes described above and also non-Markovian tests. Thus it is possible to test at the interpreter level whether a log is a sublog of another log. The interpreter provides an operator implementing the predicate $\sqsubset$ which, therefore, needs not be hand-coded by the programmer.

We consider a Debit/Credit example which we now describe to illustrate how to formulate a relational theory for nested transactions.

The database involves a relational language with:

**Fluents**: $accounts(aid, bid, abal, t, s)$, $branches(bid, bbal, bname, t, s)$, $tellers(tid, tbal, t, s)$, and $served(aid, s)$.
**Situation Independent Predicate**: $requested(aid, req)$.
**Action Functions**: $a\_insert(aid, bid, abal, tid, t)$, $a\_delete(aid, bid, abal, tid, t)$, $t\_insert(tid, tbal, t)$, $t\_delete(tid, tbal, t)$, $b\_insert(bid, bbal, bname, t)$, $b\_delete(bid, bbal, bname, t)$, and $report(aid)$.
**Constants**: $Ray$, $Iluju$, $Misha$, $Ho$, etc.

The meaning of the arguments of fluents are self explanatory; and the relational language also includes the external actions of nested transactions. Domain closure and unique name axioms are given in the usual way; thus we concentrate ourself on the remaining axioms. We enforce the following ICs ($\mathcal{IC}_e$):

$$\begin{aligned}
&accounts(aid, bid, abal, tid, t, s) \wedge accounts(aid, bid', abal', tid', t', s) \supset \\
&\qquad\qquad\qquad\qquad bid = bid', abal = abal', tid = tid', \\
&branches(bid, bbal, bname, t, s) \wedge branches(bid, bbal', bname', t', s) \supset \\
&\qquad\qquad\qquad\qquad bbal = bbal', bname = bname', \\
&tellers(tid, tbal, t, s) \wedge tellers(tid, tbal', t', s) \supset tbal = tbal';
\end{aligned}$$

and we have to verify the IC ($\mathcal{IC}_v$)

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

at transaction's end.

A sample update precondition axiom is given in (9).

A sample successor state axiom is the following for the fluent $accounts(aid, bid, abal, tid, t, s)$:

$$\begin{aligned}
&accounts(aid, bid, abal, tid, t, do(a, s)) \equiv (\exists t_1)(a = a\_insert(aid, bid, abal, tid, t_1) \vee \\
&\quad (\exists t_2) accounts(aid, bid, abal, tid, t_2, s) \wedge \neg(\exists t_3) a = a\_delete(aid, bid, abal, tid)) \wedge \\
&\qquad\qquad\qquad\qquad \neg(\exists t') a = Rollback(t') \vee \\
&\quad (\exists t').a = Rollback(t') \wedge \neg(\exists t'') parent(t'', t', s) \wedge \\
&\qquad\qquad\qquad restoreBeginPoint(F, (aid, bid, abal, tid), t', s) \vee \\
&\quad a = Rollback(t') \wedge (\exists t'') parent(t'', t', s) \wedge \\
&\qquad\qquad\qquad restoreSpawnPoint(accounts(aid, bid, abal, tid), t', s).
\end{aligned}$$

This states that the tuple $(aid, bid, abal, tid)$ will be in the $accounts$ 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 $accounts$ 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 $accounts$ relation will get a value according to the logic of (5).

Finally, the following successor state axiom 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 by indicating that a request emitted by the owner of the account $aid$ has been granted. These requests are registered in the situation independent predicate $requested(aid, req)$.

Now we give the following GOLOG 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 $processTrans(t)$
      $Begin(t);\ [(\pi\ bid, aid, abal, tid, req).$
                 $\{accounts(aid, bid, abal, tid, t) \wedge$
                   $requested(aid, req) \wedge \neg served(aid)\}?\ ; report(aid)\ ;$
                 $Spawn(t, aid)\ ;\ processReq(t, tid, aid, req)\ ;\ End(aid)]^{||}\ ;$
             $\neg((\exists\ aid, req)requested(aid, req))?\ ;\ End(t)$
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 ACI(D) 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 $\phi$ 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 the concurrent iteration in the last procedure; this spawns a new child transaction for each account that emitted a request but have 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 $processTrans(T)$, 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 5, we take particularities of system actions into account. These actions must occur whenever possible, so the interpreter must test for their possibility upon each performance of a primitive action. Definition 5 captures this requirement and allows us to show that $Do$ generates only legal situations:

**Theorem 10** *Suppose $\mathcal{D}$ is a relational theory (either for flat transactions or for CNTs), and let $T$ be a well formed GOLOG program. Then,*

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

## 7 Related Work

The inability of the classical model for concurrency control (the serializability theory) to cope with nested transactions has been addressed in [1]. This work develops a serializability theory for nested transactions. The new serializability model is articulated around the notion of computation, a generalization of the notion of history which is central to the classical serializability theory. Like the history of the classical model, a computation involves the execution of database primitive and complex operations. Unlike the history, which is a sequence of primitive operations, a computation is a tree. The interleaving of several computations constitute a partially ordered forest. Similarly to the classical case, a forest is correct (i.e. serializable) iff it is equivalent to a serial execution of the involved trees. This criterion is used to prove the correctness of concurrency control algorithms, i.e. schedulers. Correctness is considered as a property of computations generated by a scheduler. This is in spirit similar to what our Theorem 10 conveys, if we view the GOLOG interpreter as a scheduler. However, we do not go that far in this paper to deal with the proof of correctness of given schedulers. In addition to that, we still consider a linear log. This might have an implicit tree-structure that has yet to be extracted to compare our logical approach with the tree-approach of [1].

Chrysanthis and Ramamritham ([6],[5]) present a framework called ACTA which allows to specify effects of transactions on objects and on other transactions. Our framework is similar to ACTA. 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 transaction operations – $Commit$, $Rollback$, etc. – are defined with respect to constraints on this log. Nowhere have we seen a quantification over histories in ACTA, so that there is no straitforward way of expressing closed form formulas involving histories in ACTA. (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. To that end, the main implementation theorems needed are formulated in [21]. Finally, (3) although ACTA deals with the dynamics of database objects, it is never explicitly formulated as a logic for actions.

In [3], Bertossi *et al.* propose a situation calculus-based formalization of database transactions. They extend Reiter's specification of database updates to transactions. In fact, the idea of using transactional actions like $Begin$, $Rollback$, $End$, and $Commit$ for flat transactions was first introduced in [3], as was the the axiomatization of the notion of consistency verification at the transaction end. Our approach, however, is based on a situation calculus that is explicitly non-Markovian. Moreover, our work goes beyond pure flat transactions to deal with an account of notions such as serializability and atomicity, and with ATMs which are more complex.

*Transaction Logic* ([4]) and *Statelog* ([15]) 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, they are not general enough 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.

## 8 Conclusion and Future Work

One must distinguish between our approach which is a purely logical, abstract specification in which all system properties are formulated relative to the database log, and an implementation which normally materializes the database using progression ([21]). This is the distinguishing feature of our approach. The database log is a first class citizen of the logic, and the semantics of all transaction operations – $Commit$, $Rollback$, etc. – are defined with respect to this log.

As we acknowledged it in the introduction, database transaction processing is now a mature area of research. However, one needs to know whether our formalization indeed captures any existing theory, such as ACTA, at the same level of generality. Therefore, one needs to prove the correctness of the formalization. For example, we need an effective translation of our basic relational theories into ACTA axioms for a relational database and then show that the legal logs for the situation calculus basic relational theory are precisely the correct histories for its translation into a relational ACTA system.

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. These are just a examples of some of the generalizations that our initial databases could admit.

Finally, it is important to notice that the only place where the second order nature of our framework is needed is in the proof of the properties of the transaction models that rely on the second order induction principle of Section 4. For the Markovian situation calculus, it is shown in [18] that the second order nature of this language is not at all needed in simulating basic action theories. It remains to show that this is also the case for the non-Markovian setting.

The framework described in this work is currently being implemented using a regression mechanism described in [9]. On-going work extending the framework includes: accounting for some of the recent ATMs, for example those reported in [11] and open nested transactions proposed in the context of mobile computing, implementing the specifications of significant ATMs, proving the correctness of the approach, introducing on-line, that is actual execution of transactions, as opposed to off-line or hypothetical execution. We will also consider modeling active rules and different active rule processing mechanisms within the framework of this paper in the near future ([12]). Finally, we will explore ways of making this framework part of a logic-based development methodology for ATMs. Such a methodology would exhibit the important advantage of uniformity in many of its phases by using the single language of the situation calculus.

## Acknowledgments

Ray Reiter, Alfredo Gabaldon, Javier Pinto, and members of the Cognitive Robotics Group at the University of Toronto deserve many thanks for helpful discussions and encouragements. Special thanks to Ray Reiter who suggested the link between ATMs and non-Markovian theories, and Fahiem Bacchus who suggested reinforcing the simulation aspect of the framework. We are supported by NSERC, IRIS (Institute for Robotics and Intelligent Systems), and ITRC (Information Technology Research Centre of Ontario) and would like to gratefully mention them.

## References

[1] C. Beri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[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. A logic for programming database transactions. In J. Chomicki and Saake G., editors, *Logics for Databases and Information Systems*. Kluwer, 1998. Chapter 5.

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

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

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] J.W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer-Verlag, Berlin, 1988.

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

[22] 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.