Transaction Logic Programming

Anthony J. Bonner[*]
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
bonner@db.toronto.edu

Michael Kifer[†]
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11790, U.S.A.
kifer@cs.sunysb.edu

Abstract

An extension of predicate logic, called *Transaction Logic*, is proposed, which accounts in a clean and declarative fashion for the phenomenon of state changes in logic programs and databases. Transaction Logic has a natural model theory and a sound and complete proof theory, but unlike many other logics, it allows users to *program* transactions. The semantics leads naturally to features whose amalgamation in a single logic has proved elusive in the past. These features include both hypothetical *and* committed updates, dynamic constraints on transaction execution, nondeterminism, and bulk updates. Finally, Transaction Logic holds promise as a logical model of hitherto non-logical phenomena, including so-called *procedural knowledge* in AI, and the *behavior* of object-oriented databases, especially methods with side effects. This paper presents the semantics of Transaction Logic coupled with a sound and complete proof theory for a Horn-like subset of the logic.

Appears in *Proceedings of the Tenth International Conference on Logic Programming (ICLP)*, pages 257–279. MIT Press. Conference held in Budapest, Hungary, June 21–24 1993.

This paper is available at the following URL:
`ftp://db.toronto.edu/pub/bonner/papers/transaction.logic/iclp93.ps.gz`

---

[*]Work supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada and by a Connaught Grant from the University of Toronto.

[†]Supported in part by NSF grant CCR-9102159 and a grant from New York Science and Technology Foundation. Work done during sabbatical year at the University of Toronto. Support of Computer Systems Research Institute of University of Toronto is gratefully acknowledged.

1 Introduction

Transaction Logic (abbr., $\mathcal{TR}$) is a novel logic that accounts in a clean and completely first-order manner for the phenomenon of updating arbitrary logical theories, including databases and logic programs. In $\mathcal{TR}$, procedures can be specified declaratively, as logic programs, and executed by the proof theory in accordance with a model-theoretic semantics. These procedures—called *transactions*—may have side effects in the form of permanent or temporary changes to a database. This database can be as simple as a relational database or as complex as a disjunctive logic program. This paper presents the model theory and the proof theory of $\mathcal{TR}$.

$\mathcal{TR}$ was designed with several applications in mind, especially in databases, logic programming, and AI. It was therefore developed as a general logic, so that it could solve a wide range of update-related problems. Individual applications can be carved out of different fragments of the logic. These applications, both practical and theoretical, are discussed in great detail in [5]. For instance, $\mathcal{TR}$ provides a logical account of many update-related phenomena. In logic programming, this leads to a clean, logical treatment of the *assert* and *retract* operators in Prolog, which effectively extends the theory of logic programming to include updates as well as queries. In object-oriented databases, $\mathcal{TR}$ can be combined with object-oriented logics, such as F-logic [14], to provide a logical account of *methods*—procedures hidden inside objects that manipulate these objects' internal states. Thus, while F-logic covers the structural aspect of object-oriented databases, its combination with $\mathcal{TR}$ would account for the behavioral aspect as well. In AI, $\mathcal{TR}$ suggests a logical account of planning. STRIPS-like actions,[1] for instance, and many aspects of hierarchical and non-linear planning are easily expressed in $\mathcal{TR}$. In spite of the previous efforts to give these phenomena declarative semantics, until now there has been no unifying *logical* framework to account for them all.

On the surface, there would seem to be many other candidates for a logic of transactions, since many logics reason about updates or about the related phenomena of time and action. However, despite a plethora of action logics, researchers continue to complain that there is no clear declarative semantics for updates either in databases or in logic programming [4, 3, 22]. In fact—in stark contrast to classical logic—no action logic has ever become a core of databases or logic-programming, in theory or in practice. There appear to be a few simple reasons for this unsuitability of existing action logics.

One major problem is that most logics of time or action are *hypothetical*, that is, they do not permanently change the database. Instead, they reason about what *would* happen *if* certain actions took place. For instance, they might infer that *if* the pawn took the knight, then the rook *would* be threatened. Such logics are useful for reasoning about alternatives, but not for executing database updates; i.e., they do not provide executable specifications of actions. In contrast, $\mathcal{TR}$ is a logic that supports *both* real and hypothetical updates. Procedures in $\mathcal{TR}$ may commit their updates, reason about them without committing, or do any combination thereof.

Another major problem is that most logics of time or action were not designed for

[1] STRIPS was an early AI planning system that simulated the actions of a robot arm.

programming. Instead, they were intended for specifying *properties* of programs and for reasoning about them. In many of these logics, one cannot assign names to composite transactions. In their intended context (of reasoning about sequences of actions), this is not a shortcoming. However, these logics are inappropriate for programming transactions, since specifying transactions without a naming facility is like programming without subroutines. From a programming standpoint, the lack of a straightforward naming facility defeats the purpose of using logic in the first place, which is to free the user from the drudgery of low-level details.

Yet another problem is that many logics (and all relational databases) make a clear distinction between non-updating queries and actions with side effects. However, this distinction is blurred in object-oriented systems, where both queries and updates are special cases of a single idea: method invocation (or message passing). In such systems, an update can be thought of as a query with side effects. In fact, every method is simply a *program* that operates on the data. $\mathcal{TR}$ models this uniformity naturally, treating all methods equally, thereby providing a logical foundation for object-oriented databases.

Although the non-logical nature of Prolog's *assert* and *retract* operators has been extensively discussed in the literature, we point out that most of the proposals for overcoming these problems fall short of integrating updates into a complete logical system. It is therefore not clear how *assert* and *retract* should interact with other logical operators, such as disjunction and negation. For instance, what does $assert(X) \vee assert(Y)$ mean? or $\neg assert(X)$? Also, how does one logically account for the fact that the order of updates is important? Finally, what does it mean to update a database that contains arbitrary logical formulas? These questions are not addressed by Prolog's classical semantics or by subsequent works.

$\mathcal{TR}$ gives a general solution to the aforesaid limitations of Prolog and of action logics, by providing a syntax and a semantics in which updates can be combined with logical operators to build a large number of interesting and useful formulas.

Like classical logic, $\mathcal{TR}$ has a "Horn" version that is of particular interest for logic programming. In Horn $\mathcal{TR}$, a transaction is defined by Prolog-style rules in which the premise specifies a *sequence* of queries and updates. Furthermore, just as $\mathcal{TR}$ is an extension of classical first-order logic, Horn $\mathcal{TR}$ is an extension of classical Horn-clause logic. Because of its importance, much of this paper focuses on Horn $\mathcal{TR}$. Horn $\mathcal{TR}$ has a clean and simple proof theory (Section 5) that is sound and complete with respect to a model theory (Section 4).

2 Overview and Introductory Examples

From the user's point of view, using $\mathcal{TR}$ is similar to using Prolog: the user writes a logic program by specifying rules; and then he may pose queries and perform updates. In $\mathcal{TR}$, users see no obvious or immediate difference between queries and updates. An update is just a query with side effects (which can be detected only by issuing subsequent queries). In general, the user issues transactions, and the system responds by displaying answers and updating the database. This section provides simple examples of how this behavior appears to the user and how it is described formally. The examples also illustrate several dimensions of $\mathcal{TR}$'s capabilities.

One of these capabilities should be mentioned at the outset: *non-deterministic* transactions. Non-determinism has applications in many areas, but it is especially well-suited for advanced applications, such as those found in Artificial Intelligence. For instance, the user of a robot simulator might instruct the robot to build a stack of three blocks, but he may not say (or care) which blocks to use. Likewise, the user of a CAD system might request the system to run an electrical line from one point to another, without fixing the exact route, except in the form of loose constraints (*e.g.*, do not run the line too close to wet or exposed areas). In such transactions, the final state of the database is indeterminate, *i.e.*, it cannot be predicted at the outset, as it depends on choices made by the system at run time. $\mathcal{TR}$ enables users to say what choices are allowed. When a user issues a non-deterministic transaction, the system makes particular choices (which may be implementation-dependent), putting the database into one of the allowed new states.

For all but the most elementary applications, transaction execution is characterized not just by an initial and a final state, but by a sequence of *intermediate* states that the database passes through. For example, as a robot simulator piles block upon block upon block, the transaction execution will pass from state to state to state. Like the final state, intermediate states may not be uniquely determined at the start of the execution. For example, the robot may have some (non-deterministic) choice as to which block to grasp next. We call such a sequence of database states the *execution path* of the transaction. $\mathcal{TR}$ represents execution paths explicitly. By doing so, it can express a wide range of constraints on transaction execution. For example, a user may require every intermediate state to satisfy some condition, or he may forbid certain sequences of states.

Execution of transactions is formally described using statements, called *executional entailment*, that express a form of logical entailment in $\mathcal{TR}$:

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \psi \tag{1}$$

Here, ψ is a logical formula, and $\mathbf{P}$ and $\mathbf{D}_i$ are finite sets of logical formulas. $\mathbf{P}$ is called the *transaction base*. Intuitively, $\mathbf{P}$ is a set of transaction definitions, ψ is a transaction invocation, and $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is a sequence of databases, representing all the states of transaction execution. Statement (1) means that $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is an execution path of transaction ψ. That is, if the current database state is $\mathbf{D}_0$, and if the user issues the transaction ψ (by typing $?-\ \psi$, as in Prolog), then the database *may* go from state $\mathbf{D}_0$ to state $\mathbf{D}_1$, to state $\mathbf{D}_2$, etc., until it finally reaches state $\mathbf{D}_n$, after which the transaction terminates. We emphasize the word "may" because ψ may be a non-deterministic transaction. As such, it may have many execution paths beginning at $\mathbf{D}_0$. The proof theory for $\mathcal{TR}$ can derive each of these paths, but only one of them will be (non-deterministically) selected as the actual execution path; the final state, $\mathbf{D}_n$, of that path then becomes the new database.

There are two differences between the transaction base, $\mathbf{P}$, and a database, $\mathbf{D}_i$. First, a database can be updated by transactions, but the transaction base is immutable. Second, formulas in the transactions base may use the full syntax of $\mathcal{TR}$, whereas database formulas are limited to classical logic, a subset of $\mathcal{TR}$.

Unlike other formalisms, $\mathcal{TR}$ does not draw a thick line between transactions and queries. In fact, any transaction, ϕ, that does not cause a state change can be viewed as a query. This state of affairs is formally expressed by the statement $\mathbf{P}, \mathbf{D}_0 \models \phi$, a special case of statement (1) in which $n = 0$. In this case, $\mathbf{D}_0$ is a sequence of databases of length 1. This uniform treatment of transactions and queries is crucial for successful adaptation of $\mathcal{TR}$ to the object-oriented domain, because object-oriented systems do not syntactically distinguish between state-changing and information-retrieving methods.

The rest of this section illustrates our notation and the capabilities of $\mathcal{TR}$ through a number of simple examples. The examples illustrate how $\mathcal{TR}$ uses logical operators to combine simple actions into complex ones.

2.1 Simple Transactions

In $\mathcal{TR}$, all transactions are a combination of queries and updates. Queries do not change the database, and can be expressed in classical logic. In contrast, updates do change the database, and are expressed in an extension of classical logic. We call the simplest kind of updates *elementary updates* or *elementary state transitions.* In principle, there are no restrictions on the changes that an elementary update can make to a database, though in practice, we expect them to be simple and cheap. In this paper, we use the insertion and deletion of atomic formulas as canonical examples of elementary updates. However, other kinds of elementary updates may also be useful. For instance, *relational assignment* can be used to express SQL-style *bulk* updates [5]

Elementary updates are atomic in that they cannot be decomposed into simpler updates. We therefore represent them by atomic formulas. Like all atomic formulas, elementary updates have a truth value; but in addition, they have a side effect on the database. Formally, this is represented thus:

$$\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models u$$

This executional entailment says that the atomic formula u is (the name of) an update that changes the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$. Although any atomic formula can be an update, it is a good programming practice to reserve a special set of predicate symbols for this purpose. For example, in this paper, for each predicate symbol p, we use another predicate symbol, *ins:p*, to represent insertions into p. Likewise, we use the predicate symbol *del:p* to represent deletions from p.

Example 2.1 (Elementary Updates) Let *in* be a binary predicate symbol. Then the atoms $ins{:}in(pie, sky)$ and $del{:}in(pie, sky)$ are elementary updates. Intuitively, $ins{:}in(pie, sky)$ means, "insert the atom $in(pie, sky)$ into the database." Likewise, the atom $del{:}in(pie, sky)$ means, "delete $in(pie, sky)$ from the database." From the user's perspective, typing $?-$ $ins{:}in(pie, sky)$ to the interpreter changes the database from $\mathbf{D}$ to $\mathbf{D} + \{in(pie, sky)\}$. Likewise, typing $?-$ $del{:}in(pie, sky)$ changes the database from $\mathbf{D}$ to $\mathbf{D} - \{in(pie, sky)\}$. We express this behaviour formally by the following two statements, which are true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{in(pie, sky)\} \models ins{:}in(pie, sky)$$
$$\mathbf{P}, \mathbf{D}, \mathbf{D} - \{in(pie, sky)\} \models del{:}in(pie, sky)$$

□

Here we use "+" and "−" to denote set union and difference, respectively. This is sufficient for relational databases, i.e., sets of atomic formulas. For more complex databases, insertion and deletion are more complex operations [13]. Note, however, that insertion and deletion are *not* built into the semantics of $\mathcal{TR}$. In fact, $\mathcal{TR}$ is not committed to any particular set of elementary updates. Thus, there is no *intrinsic* connection between the names p, *ins:p* and *del:p*. Our use of these names is merely a convention for purposes of illustration. In fact, p, *ins:p* and *del:p* are ordinary predicates of $\mathcal{TR}$, and the connection between them is established via axioms of the so called *transition base*, as explained later.

A basic way of combining transactions is to *sequence* them, *i.e.*, to execute them one after another. For example, we may take money out of one account and then, if the withdrawal succeeds, deposit the money into another account. To combine transactions sequentially, we extend classical logic with a new binary connective, $\otimes$, called *serial conjunction.* The formula $\psi \otimes \phi$ denotes the composite transaction consisting of transaction ψ followed by transaction ϕ. Unlike elementary updates, sequential transactions often have intermediate states, as well as initial and final states. We express this behavior formally by statements like the following:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \ \models \ \psi \otimes \phi$$

That is, $?-\ \psi \otimes \phi$ changes the database from $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$. Here, $\mathbf{D}_1$ is an intermediate state.

Example 2.2 (Serial Conjunction) The expression *unlucky* $\otimes$ *ins:poor* $\otimes$ *ins:sad*, where *unlucky*, *poor*, and *sad* are ground atomic formulas, denotes a sequence of two insertions preceded by a test. This transaction means, "First check that *unlucky* is true; then insert *poor* into the database; and then insert *sad*." Thus, if the initial database is $\mathbf{D}$, and if the user issues a transaction by typing $?-$ *unlucky* $\otimes$ *ins:poor* $\otimes$ *ins:sad*, then during execution, the database will change from $\mathbf{D}$ to $\mathbf{D} + \{poor\}$ to $\mathbf{D} + \{poor, sad\}$, provided that *unlucky* was initially true in $\mathbf{D}$. We express this behaviour formally by the following statement, which holds for any transaction base $\mathbf{P}$, where $\mathbf{P}, \mathbf{D} \models unlucky$ is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{poor\}, \mathbf{D} + \{poor, sad\} \quad \models \quad unlucky \otimes ins{:}poor \otimes ins{:}sad$$

This example illustrates the use of preconditions. $\mathcal{TR}$ can express post-conditions and tests on intermediate database states as well. □

2.2 Rules and Non-deterministic Transactions

Rules are formulas of the form $p \leftarrow \phi$, where p is an atomic formula and ϕ is any $\mathcal{TR}$ formula. As in classical logic, this formula is just a convenient abbreviation for $p \vee \neg\phi$. This is the formal, declarative interpretation of rules. Operationally, the formula $p \leftarrow \phi$ means, "to execute p, it is sufficient to execute ϕ." This interpretation is important because it provides $\mathcal{TR}$ with a subroutine facility and makes logic programming possible. For instance, in the rule $p(X) \leftarrow \phi$, the predicate symbol p acts as the name of a procedure, the variable X acts as an input parameter, and the formula ϕ acts as the procedure body or definition (exactly as in Horn-clause logic programming). Although the rule-body may be any $\mathcal{TR}$ formula, it will frequently be a serial conjunction. In this case, the rule has the form $a_0 \leftarrow a_1 \otimes a_2 \otimes ... \otimes a_n$, where each a_i is an atom. With such rules, users can define transaction subroutines and write transaction logic programs. Note that this facility is possible because transactions are represented by predicates, which distinguishes $\mathcal{TR}$ from other logics of action, especially those in which actions are modal operators. In such logics, subroutines are awkward, if not impossible, to express. Finally, for notational convenience, we assume that all free variables in a rule are universally quantified outside the rule. Thus, the rule $p(X) \leftarrow \phi$ is simply an abbreviation for $\forall X\, [p(X) \leftarrow \phi]$.

Example 2.3 (Flipping Coins) Suppose the transaction base $\mathbf{P}$ contains the rules:

$$flip(X) \leftarrow ins{:}heads(X) \qquad\qquad flip(X) \leftarrow ins{:}tails(X)$$

These rules say that, to flip a coin, say $dime$, either insert $heads(dime)$ into the database or insert $tails(dime)$. Thus, $?-\ flip(dime)$ is a non-deterministic transaction. Formally, the result of flipping a dime is represented thus:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{heads(dime)\} \models flip(dime) \qquad \mathbf{P}, \mathbf{D}, \mathbf{D} + \{tails(dime)\} \models flip(dime)$$

This means that we cannot know in advance what the exact outcome of a flipping action will be. □

2.3 Transaction Bases

This section gives simple but realistic examples of transaction bases comprised of finite sets of rules. The examples show how updates can be combined with queries to define complex transactions. In each example, the body of each rule is a sequence of atomic formulas, some of which are queries and some of which are updates. The examples also illustrate the use of transaction subroutines.

Example 2.4 (Financial Transactions) Suppose the balance of a bank account is given by the relation $balance(Acnt, Amt)$. To modify this relation, we are provided with a pair of elementary update operations: $del{:}balance(Acnt, Amt)$ to delete a tuple from the relation, and $ins{:}balance(Acnt, Amt)$ to insert a tuple into the relation. Using these two updates, we define four transactions: $change{:}balance(Acnt, Bal, Bal')$ to change the balance of an account; $withdraw(Amt, Acnt)$ to withdraw an amount from an account; $deposit(Amt, Acnt)$ to deposit an amount into an account; and $transfer(Amt, Acnt, Acnt')$ to transfer an amount from one account to another. These transactions are defined by the following four rules:

$$\begin{aligned}
&transfer(Amt, Acnt, Acnt') \leftarrow withdraw(Amt, Acnt) \otimes deposit(Amt, Acnt')\\
&withdraw(Amt, Acnt) \leftarrow balance(Acnt, B) \otimes change{:}balance(Acnt, B, B - Amt)\\
&deposit(Amt, Acnt) \leftarrow balance(Acnt, B) \otimes change{:}balance(Acnt, B, B + Amt)\\
&change{:}balance(Acnt, B, B') \leftarrow del{:}balance(Acnt, B) \otimes ins{:}balance(Acnt, B')
\end{aligned}$$

In the second and third rules, the atom $balance(Act, Bal)$ is a query that retrieves the balance of the specified account. All other atoms are updates. □

Example 2.5 (Non-deterministic, Recursive Robot Actions) The following transaction base simulates the movements of a robot arm in a world of toy blocks. States of this world are defined in terms of three database predicates: $on(x, y)$, which says that block x is on top of block y; $clear(x)$, which says that nothing is on top of block x; and $wider(x, y)$, which says that x is wider than y. The rules below define four actions that change the state of the world. Each action evaluates its premises in the order given, and the action fails if any of its premises fails (in which case the database is left in its original state).

$$\begin{array}{l} stack(N, X) \leftarrow N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \\ stack(0, X) \leftarrow \\ move(X, Y) \leftarrow pickup(X) \otimes putdown(X, Y) \\ pickup(X) \leftarrow clear(X) \otimes on(X, Y) \otimes del{:}on(X, Y) \otimes ins{:}clear(Y) \\ putdown(X, Y) \leftarrow X \neq Y \otimes wider(Y, X) \otimes clear(Y) \\ \qquad\qquad\qquad\qquad\qquad \otimes ins{:}on(X, Y) \otimes del{:}clear(Y) \end{array} \tag{2}$$

The basic actions, $pickup(X)$ and $putdown(X, Y)$, mean, respectively, "pick up block X," and "put down block X on top of block Y, where Y must be wider than X." Both are defined in terms of elementary inserts and deletes to database relations. The remaining rules combine simple actions into more complex ones. For instance, $move(X, Y)$ means, "move block X to the top of block Y," and $stack(N, X)$ means, "stack N arbitrary blocks on top of block X." The actions *pickup* and *putdown* are deterministic, since each set of argument bindings specifies only one robot action.[2]

In contrast, the action *stack* is *non-deterministic*. To perform this action, the inference system searches the database for blocks that can be stacked. If, at any stage of the process, several such blocks can be placed on top of the stack, the system arbitrarily chooses one of them. □

Observe that rules (2) can easily be rewritten in Prolog form, by replacing "⊗" with "," and by replacing the elementary state transitions with *assert* and *retract*. However, the resulting, apparently innocuous, Prolog program does not execute correctly! The problem is that Prolog updates are not undone during backtracking. For instance, suppose that during a *move* action, the robot picked up $blkA$, the widest block on the table. The *move* action would then fail, since the robot cannot put $blkA$ down on the stack, since $blkA$ is too wide. In $\mathcal{TR}$, the inference system simply backtracks and then tries to find another block to pick up. Prolog, too, will backtrack, but it will leave the database in an incorrect state, since it will not undo the *pickup* action. Thus, if $blkA$ was previously on top of $blkB$, then $on(blkA, blkB)$ would remain deleted and $clear(blkB)$ would stay in the database.

2.4 Constraints

Classical conjunction constrains the non-determinism of transactions. That is, in general, the transaction $\psi \wedge \phi$ is more deterministic than either ϕ or ψ by themselves, because any execution of $\psi \wedge \phi$ must be an allowed execution of ψ *and* an allowed execution of ϕ. To illustrate, consider the following conjunction of two robot actions:

[2] Assuming that only one block can be on top of another block.

$$\text{“Go to the kitchen”} \;\wedge\; \text{“Don't pass through the bedroom”}$$

Here, each conjunct is a non-deterministic action, as there are many ways in which it can be carried out. The composite action, however, is more constrained than either of the two conjuncts alone. In this way, conjunction reduces non-determinism and allows a user to specify what is *not* to be done.

Note that classical conjunction does not cause the conjuncts to be executed as two separate transactions. Instead, it combines them into a single, more tightly constrained transaction; "$\wedge$" thus constrains the *entire execution* of a transaction, not just the final state. In general, "$\wedge$" constrains transactions in two ways: (i) by causing transactions to fail, and (ii) by forcing non-deterministic transactions to execute in certain ways.

Example 2.6 (Transaction Failure) Consider $?-\ ins{:}bought \otimes ins{:}wanted$ and $?-\ ins{:}wanted \otimes ins{:}bought$. Both these transactions transform database state $\mathbf{D}$ into state $\mathbf{D} + \{bought, wanted\}$. However, they pass through different intermediate states. The former goes through the state $\mathbf{D} + \{bought\}$, while the latter passes through the state $\mathbf{D} + \{wanted\}$. The conjunction $(ins{:}bought \otimes ins{:}wanted) \wedge (ins{:}wanted \otimes ins{:}bought)$ therefore fails, since there is no single sequence of states that is a valid execution path of both conjuncts. Formally, the following two statements are both true:

$$\begin{array}{rcl} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{bought\}, \mathbf{D} + \{bought, wanted\} & \models & ins{:}bought \otimes ins{:}wanted \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{wanted\}, \mathbf{D} + \{bought, wanted\} & \models & ins{:}wanted \otimes ins{:}bought \end{array}$$

but the following statement is false for any sequence of databases $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$:[3]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n \quad \models \quad (ins{:}bought \otimes ins{:}wanted) \wedge (ins{:}wanted \otimes ins{:}bought)$$

□

Example 2.7 (Reducing Non-Determinism) Consider a pair of non-deterministic transactions, $?-\ ins{:}lost \vee ins{:}found$ and $?-\ ins{:}lost \vee ins{:}won$. Starting from database $\mathbf{D}$, they can both follow the same path to terminate at $\mathbf{D} + \{lost\}$. In fact, this is the only database that can be reached by both transactions.[4] Hence, following the execution of the transaction $?-\ (ins{:}lost \vee ins{:}found) \wedge (ins{:}lost \vee ins{:}won)$ the final database state would be $\mathbf{D} + \{lost\}$. Formally, the following is true:[4]

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \models (ins{:}lost \vee ins{:}found) \wedge (ins{:}lost \vee ins{:}won) \quad \textit{iff} \quad \mathbf{D}' = \mathbf{D} + \{lost\}$$

In this way, classical conjunction reduces non-determinism and, in this particular example, yields a completely deterministic transaction. □

In [5], we explore the richness of $\mathcal{TR}$ for expressing constraints. Much of this expressiveness comes from serial conjunction, especially when combined with negation. For example, each of the following formulas has a natural meaning as a constraint:

- $\neg(a \otimes b \otimes c)$ means that the sequence $a \otimes b \otimes c$ is not allowed.

[3] Assuming *bought* and *wanted* are not in $\mathbf{D}$.
[4] Assuming *found* and *won* are not in $\mathbf{D}$.

- $\phi \otimes \neg\psi$ means that transaction ψ must *not* immediately follow transaction ϕ.
- $\neg(\phi \otimes \neg\psi)$ means that transaction ψ *must* immediately follow transaction ϕ.

These formulas can often be simplified by using the dual operator $\oplus$, called *serial disjunction*. For example, the last formula can be rewritten as $\neg\phi \oplus \psi$. The repertoire of executional constraints expressible in $\mathcal{TR}$ is very large. It is easy to specify that transactions must overlap, start or end simultaneously, one should terminate after the other, etc. In [5] we show that the full set of temporal relationships of Allen's logic of time intervals [2] has a simple and natural representation in $\mathcal{TR}$.

Besides its novel use for expressing constraints, "$\wedge$" has the traditional role in forming logic programs: in $\mathcal{TR}$, as in classical logic, any finite set of rules is equivalent to a conjunction of all the rules in the set.

3 Syntax

The syntax of $\mathcal{TR}$ distinguishes two kinds of formulas: *transaction formulas* and *elementary transitions*. The former define composite transactions, and the latter define elementary updates.

Transaction formulas are the formulas that most users will work with, using them to define transactions and formulate queries. Transaction formulas extend first-order formulas with a new connective, $\otimes$, called *serial conjunction*. Formally, transaction formulas are defined recursively as follows. An *atomic* transaction formula is an expression of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol, and $t_1, \ldots, t_n$ are terms (as in classical predicate calculus). If ϕ and ψ are transaction formulas, then so are $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \otimes \psi$, $\neg\phi$, $(\forall X)\phi$, and $(\exists X)\phi$, where X is a variable. The expression $a(X) \vee \neg[b(X) \otimes c(X, Y)]$ is a transaction formula. Intuitively, $\psi \otimes \phi$ means, "Do ψ and then do ϕ." A dual connective, *serial disjunction*, is also useful (Section 2.4): $\psi \oplus \phi$ is equivalent to $\neg(\neg\phi \otimes \neg\psi)$.

Transaction formulas combine simple transactions into complex ones. However, we also need a way to specify *elementary* changes to a database. One way to define such changes is to build them into the semantics as in [17, 21, 7, 1, 19]. The problem with this approach is that adding new kinds of elementary transitions leads to a redefinition of the very notion of a model and thus to a revamping of the entire theory, including proofs of soundness and completeness. This drawback is quite serious, as there appears to be no small, single set of elementary transitions that is best for all purposes [5]. Thus, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat these transitions as a *parameter* of $\mathcal{TR}$. Each set of such transitions, thus, gives rise to a different version of the logic. To achieve this, elementary transitions are defined by logical axioms.

Elementary transitions are formulas of the form $\langle\phi, \psi\rangle u$, where ϕ, ψ are closed first-order formulas and u is an atomic formula—the *name* of the transition. Intuitively, this formula says that u is an update that transforms database ϕ into database ψ. For instance, in Section 2, the predicates *ins:b* and *del:b* would be defined by an enumerable set of elementary transitions consisting of the formulas $\langle \mathbf{D}, \mathbf{D} + \{b\}\rangle\, \textit{ins:b}$ and $\langle \mathbf{D}, \mathbf{D} - \{b\}\rangle\, \textit{del:b}$ for every relational database $\mathbf{D}$.[5] Enumerable sets of such transitions are called *transition bases*. In practice, transaction bases would not be materialized, but rather generated on demand by an algorithm. The reader is referred to [5] for a more detailed discussion.

4 Model Theory

Just as the syntax is based on two basic ideas—serial conjunction and elementary transitions—the semantics is also based on a few fundamental ideas:

- *Transaction Execution Paths*: A transaction causes a sequence of database state changes;
- *Database States*: A database state is a *set* of (classical) first-order semantic structures;
- *Executional Entailment*: Transaction execution corresponds to truth over a sequence of states.

Transaction Execution Paths: When the user executes a transaction, the database may change, going from the initial state to some other state. In doing so, the execution may pass through any number of intermediate states. For example, execution of ?– $ins{:}a \otimes ins{:}b \otimes ins{:}c$ takes the database from an initial state, $\mathbf{D}$, through the intermediate states $\mathbf{D} + \{a\}$ and $\mathbf{D} + \{a, b\}$, to the final state $\mathbf{D} + \{a, b, c\}$. This idea of a sequence of states is central to our semantics. It also allows us to model a wide range of constraints. For example, we may require that every intermediate state satisfy some condition, or we may forbid certain sequences of states.

To model transactions, we start with a modal-like semantics, where each state represents a database, and each elementary update causes a transition from one state to another, thereby changing the database. At this point, however, modal logic and Transaction Logic begin to part company. The first major difference is that truth in $\mathcal{TR}$ structures does not hinge on a set of arcs between states. Instead, we focus on *paths*, that is, on sequences of states. Because of the emphasis on paths, we refer to semantic structures in $\mathcal{TR}$ as *path structures*. Second, truth in path structures is defined on paths, not states. For example, we would say that the path $\langle \mathbf{D},\ \mathbf{D}+\{a\},\ \mathbf{D}+\{a,b\} \rangle$ satisfies the formula $ins{:}a \otimes ins{:}b$, since it represents an insertion of a followed by an insertion of b. A path of length 1 corresponds to a single database state. In this way, one model-theoretic device, paths, accounts for databases, updates, queries and more general transactions.

Database States: Another difference between modal logic and Transaction Logic is in the nature of states. In modal logic, a state is basically a first-order semantic structure, since each state specifies the truth of a set of ground atomic formulas. Such structures are adequate for representing relational databases, but not for representing more general theories, like indefinite databases or general logic programs. We therefore take a more general approach. Since a database is a first-order formula, which has a *set* of first-order models, we

[5] If $\mathbf{D}$ is a general first-order formula, defining insertion and deletion is more involved [13].

define a state to be a *set* of first-order semantic structures. Each state, s, thus corresponds to a particular database—the database having precisely the models comprising s.

This approach to states provides a lot of flexibility when defining elementary updates. Such flexibility is needed since, for general databases, the semantics of elementary updates is not clear, not even for relatively simple updates like insert and delete. For example, what does it mean to insert an atom b into a database that entails $\neg b$, especially if $\neg b$ itself is not explicitly present in the database? There is no consensus on the answer to this question, and many solutions have been proposed (see [13] for a comprehensive discussion). For these reasons, we take a general approach to elementary updates. For us, an elementary update is a mapping that takes each database $\mathbf{D}_1$ to some other database $\mathbf{D}_2$, where a database is any first-order formula. More generally, an elementary update may be non-deterministic, so it is not just a mapping, but a *binary relation* on databases.

4.1 Path Structures and Models

This section makes the preceding discussion precise. In the definitions below, each path structure has a domain of objects and an interpretation for all function symbols, which are used to interpret formulas on every path in the structure.

Definition 4.1 (Path Structures) Let $\mathcal{L}$ be a first-order language with function symbols in $\mathcal{F}$ and predicate symbols in $\mathcal{P}$. A *path structure* $\mathbf{M}$ over $\mathcal{L}$ is a quadruple $\langle U, I_{\mathcal{F}}, N, I_{path}\rangle$ where

- U is the *domain* of $\mathbf{M}$.
- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$. It assigns a function $U^n \longmapsto U$ to every n-ary function symbol in $\mathcal{F}$.

  Let $Struct(U, I_{\mathcal{F}})$ denote the set of all usual first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$, where $I_{\mathcal{P}}$ is a mapping that interprets predicate symbols in $\mathcal{P}$ by relations on U.
- N is a non-empty set of *states,* where each state is a non-empty subset of $Struct(U, I_{\mathcal{F}})$. An element of N is called a state of the path structure, $\mathbf{M}$.

  A *path* of length k in $\mathbf{M}$ is any finite sequence of states, $\langle s_1, \ldots, s_k\rangle$ where $k \geq 1$ and $s_i \in N$.
- I_{path} is a mapping that assigns to every path in $\mathbf{M}$ a first-order semantic structure in $Struct(U, I_{\mathcal{F}})$, subject to the restriction that $I_{path}(\langle s\rangle) \in s$ for every state s. (Recall that s is a *set* of semantic structures.)

□

The mapping I_{path} serves as the semantic link between transactions and paths: Given a path and a transaction transaction formula, I_{path} determines whether the formula is true on the path (Definition 4.2, below). The restriction that $I_{path}(\langle s\rangle) \in s$ guarantees that any path of length 1 (*i.e.*, a view of the database state) is a model of the underlying database. Note that for an arbitrary path π, the semantic structure $I_{path}(\pi)$ is independent of the subpaths of π. Intuitively, this means that we know nothing about the relationship between transactions and their subtransactions. Such knowledge, when it exists, is encoded in the transaction base. It is therefore in the definition of satisfaction that paths and subpaths are related.

Before defining satisfaction, it is convenient to define path *splits*. Given a path, $\langle s_1, ..., s_n \rangle$, any state, s_i, on the path defines a split of the path into two parts, $\langle s_1, ..., s_i \rangle$ and $\langle s_i, ..., s_n \rangle$. If path π is split into parts γ and δ, then we write $\pi = \gamma \circ \delta$. Thus, γ is a prefix of π, and δ is a suffix of π.

As in classical logic, in order to define satisfaction for quantified formulas and open formulas, it is convenient to introduce variable assignments. A *variable assignment* ν is a mapping, $\mathcal{V} \longmapsto U$, that takes a variable as input, and returns a domain element as output. We extend the mapping from variables to terms in the usual way, *i.e.*, $\nu(f(t_1, \ldots, t_n)) = I_{\mathcal{F}}(f)(\nu(t_1), \ldots, \nu(t_n))$.

Definition 4.2 (Satisfaction) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, N, I_{path} \rangle$ be a path structure, let π be a path in $\mathbf{M}$, and let ν be a variable assignment. Then,

1. $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, where "$\models^c_\nu$" denotes classical satisfaction in first-order logic, and $p(t_1, \ldots, t_n)$ is an atomic formula.

2. $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if $\mathbf{M}, \pi \not\models_\nu \phi$.

3. $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$.
   As usual, the meaning of "$\wedge$" is dual to that of "$\vee$".

4. $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \gamma \models_\nu \phi$ and $\mathbf{M}, \delta \models_\nu \psi$ for *some* split $\gamma \circ \delta$ of path π.

5. $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X. As usual, the meaning of $(\exists X)\phi$ is dual to that of $(\forall X)\phi$.

As in classical logic, the variable assignment ν can be omitted for *sentences*, *i.e.*, for formulas with no free variables. From now on, we will deal only with sentences, unless explicitly stated otherwise. □

In $\mathcal{TR}$, atoms like $p(t_1, \ldots, t_n)$ play the role of "subroutine calling sequences" in programming-language parlance. Intuitively, executing the subroutine corresponds to finding a path on which $p(t_1, \ldots, t_n)$ is true. Also, item 4 establishes a relationship between a path and its subpaths, which corresponds to the relationship between a transaction and its subtransactions. In particular, an atom, p, may be true on a path but false on all proper subpaths (and vice-versa). Intuitively, this means that transaction p does not call itself recursively.

Definition 4.3 (Models of Transaction Formulas) A path structure $\mathbf{M}$ is a *model* of a $\mathcal{TR}$-formula ϕ, denoted $\mathbf{M} \models \phi$, if and only if $\mathbf{M}, \pi \models \phi$ for every path π in $\mathbf{M}$. A path structure is a model of a set of formulas if and only if it is a model of every formula in the set. □

As usual in first-order logic, we define $\phi \leftarrow \psi$ and $\psi \rightarrow \phi$ to mean $\phi \vee \neg\psi$, resp., and $\phi \leftrightarrow \psi$ to mean $(\phi \leftarrow \psi) \wedge (\phi \rightarrow \psi)$. By replacing $\vee$ with $\oplus$ (the dual of $\otimes$), we obtain another interesting pair of serial connectives: the *left serial implication*, $\psi \Leftarrow \phi$, standing for $\psi \oplus \neg\phi$, and the *right* serial implication, $\phi \Rightarrow \psi$, denoting $\neg\phi \oplus \psi$. Intuitively, these formulas say that, "action ϕ must be immediately preceded (resp., followed) by action ψ." Unlike "$\leftarrow$" and "$\rightarrow$", these connectives are not identical, *i.e.*, $\phi \Leftarrow \psi$ is *not* equivalent to $\psi \Rightarrow \phi$; rather, $\phi \Leftarrow \psi$ is equivalent to $\neg\phi \Rightarrow \neg\psi$. It is easy to verify that every path structure is a model of the following formulas, which are analogous to De Morgan's laws:

$$\begin{array}{lcl}
(\phi \vee \psi) \otimes \eta & \leftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\
(\phi \wedge \psi) \oplus \eta & \leftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\
(\phi \wedge \psi) \otimes \eta & \longrightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta) \\
(\phi \vee \psi) \oplus \eta & \longleftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta)
\end{array} \tag{3}$$

Definition 4.3 tells us what it means for a path structure to be a model of a transaction formula ϕ. Such formulas are used to define complex transactions in terms of simpler ones. In addition, we must define what it means to be a model of an elementary state transition, $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$. Intuitively, this formula means that u is an update that changes database $\mathbf{D}_1$ into database $\mathbf{D}_2$. The next two definitions make this idea precise.

Definition 4.4 (Correspondence) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, N, I_{path} \rangle$ be a path structure. For each first-order formula $\mathbf{D}$, the expression $\mathbf{D} \rightsquigarrow s$ means, "s is the set of *all* (first-order) models of $\mathbf{D}$ in $Struct(U, I_{\mathcal{F}})$." We say that s *corresponds* to database $\mathbf{D}$. $\Box$

Note that the meaning of $\mathbf{D} \rightsquigarrow s$ depends on $\mathbf{M}$ (its domain and its interpretation of function symbols). The path structure will always be clear from the context.

Definition 4.5 (Models of Transition Bases) Let $\mathbf{M}$ be a path structure, let $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ be an elementary state transition, and suppose that $\mathbf{D}_1 \rightsquigarrow s_1$ and $\mathbf{D}_2 \rightsquigarrow s_2$. Then, the transition is *satisfied* in $\mathbf{M}$, denoted $\mathbf{M} \models \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$, if and only if s_1 and s_2 are states of $\mathbf{M}$ and $\mathbf{M}, \langle s_1, s_2 \rangle \models u$. $\Box$

In this definition, $\langle s_1, s_2 \rangle$ is a path of length 2, and the entailment relation "$\models$" is from Definition 4.2. Intuitively, the statement $\mathbf{M} \models \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ means that u is true on the arc from s_1 to s_2 in $\mathbf{M}$. Note that unlike transaction formulas, which are true on paths, truth of an elementary transition is determined with respect to the entire path structure. We say that $\mathbf{M}$ is a *model* of a transition base $\mathcal{B}$ if and only if $\mathbf{M}$ satisfies every elementary state transition in $\mathcal{B}$.

4.2 Execution as Entailment

We now define *executional entailment*, a concept that connects model theory with transaction execution. Recall that a program in $\mathcal{TR}$ consists of three distinct parts: a transaction base $\mathbf{P}$, a database $\mathbf{D}$, and a transition base $\mathcal{B}$. Each part plays a distinct role in defining executional entailment. Of these three parts, only the database is updatable. The other two parts specify procedures for updating the database and answering queries. The transition base defines elementary updates (state transitions), and the transaction base contains logical rules that define complex queries and transactions; normally, it will be composed of formulas containing the serial connectives $\otimes$ or $\oplus$, though classical first-order formulas are also allowed. In contrast, the database consists entirely of classical formulas.

Definition 4.6 (Executional Entailment) Let $\mathcal{B}$ be a transition base, and $\mathbf{P}$ be a transaction base. Let ϕ be a transaction formula, and let $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of databases (first-order formulas). Then, the following statement

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{4}$$

is true if and only if for every model $\mathbf{M}$ of $\mathbf{P}$ and $\mathcal{B}$, there is a path $\langle s_0, s_1, \ldots, s_n \rangle$ in $\mathbf{M}$ such that $\mathbf{D}_i \rightsquigarrow s_i$, for $i = 0, 1, ..., n$, and $\mathbf{M}, \langle s_0, s_1, \ldots, s_n \rangle \models \phi$. Related to this is the following statement:

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{5}$$

that is true iff there is a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ that makes (4) true. □

Intuitively, Statement (4) means that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$. Formally, it means that every model of $\mathbf{P}$ and $\mathcal{B}$ has a path corresponding to $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ that satisfies ϕ.

Normally, users issuing transactions know only the initial database state $\mathbf{D}_0$. To account for this situation, the version of entailment in (5) allows us to omit the intermediate and the final database states. Intuitively, Statement (5) means that transaction ϕ can execute successfully starting from database $\mathbf{D}_0$. When the context is clear, we simply say that transaction ϕ *succeeds*. Likewise, when statement (5) is not true, we say that transaction ϕ *fails*. In Section 5, we present an inference system that allows us to *compute* a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ that satisfies Statement (4) whenever a transaction succeeds.

Lemma 4.7 (Basic Properties of Executional Entailment) *For any transition base $\mathcal{B}$, any transaction base $\mathbf{P}$, any database sequence $\mathbf{D}_0, \ldots, \mathbf{D}_n$, and any transaction formulas α and β, the following statements are all true:*

1. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$
   then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.

2. *If* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$
   then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.

3. *If* $\alpha \leftarrow \beta \in \mathbf{P}$ *and* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.

4. *If* $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$ *is in* $\mathcal{B}$, *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.

5. *If* $\mathbf{D}_0 \models^c \psi$ *then* $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \psi$, *where ψ is a first-order formula,*
   and $\models^c$ denotes classical entailment.

Note that Lemma 4.7 suggests a simple inference system, in which items 4 and 5 are axioms, and items 1–3 are inference rules. Also, $n = 0$ corresponds to the special case in which a transaction does not affect a database, *i.e.*, in which it acts as a query. In this case, classical and serial conjunction are identical:

Lemma 4.8 (Conjunctive Queries) *For any transition base $\mathcal{B}$, any transaction base* $\mathbf{P}$*, any database* $\mathbf{D}$*, and any transaction formulas α and β,*

$$\mathcal{B}, \mathbf{P}, \mathbf{D} \models \alpha \wedge \beta \quad \textit{if and only if} \quad \mathcal{B}, \mathbf{P}, \mathbf{D} \models \alpha \otimes \beta$$

Intuitively, this lemma says that the result of evaluating a conjunctive query is the same whether the conjuncts are evaluated sequentially or in parallel.

5 Proof Theory

$\mathcal{TR}$ has a sound and complete proof procedure [6]. This procedure, however, is relatively complex and is beyond the scope of this paper. Fortunately, like classical logic, $\mathcal{TR}$ has a "Horn" version—called *serial-Horn* $\mathcal{TR}$—which has a substantially-simpler proof theory. Like classical Horn programs, serial-Horn programs have both a procedural and a declarative semantics. It is this property that allows a user to *program* transactions within the logic. This section defines the serial Horn subset of $\mathcal{TR}$ and develops its proof theory, a practical SLD-style proof procedure based on unification. Unlike classical logic programming, the proof procedure presented in this section computes new database states *as well as* query answers.

Serial-Horn programs are based on the idea of a *serial conjunction*. A serial conjunction is a transaction formula of the form $a_1 \otimes a_2 \otimes ... \otimes a_n$, where each a_i is an atomic formula. A *serial-Horn rule* has the form $a_0 \leftarrow a_1 \otimes a_2 \otimes ... \otimes a_n$, where the body, $a_1 \otimes a_2 \otimes ... \otimes a_n$, is a serial conjunction and the head, a_0, is an atom. All the rules in Section 2.3 are serial Horn. If $a_1 ,..., a_n$ are non-updating queries, then the expression $a_1 \otimes ... \otimes a_n$ is equivalent to $a_1 \wedge ... \wedge a_n$. Thus, classical Horn rules are a special case of serial-Horn rules. Finally, a *serial-Horn program* is a transaction base, $\mathbf{P}$, a transition base, $\mathcal{B}$, and a database, $\mathbf{D}$, that satisfy the following:

(i) The transaction base, $\mathbf{P}$, must be a set of serial-Horn rules.

(ii) The database must be a set of classical Horn rules.

(iii) The transaction (query) must be an existential serial conjunction, *i.e.*, a formula of the form $(\exists \overline{X})\phi$, where ϕ is a serial conjunction and $\overline{X}$ is a list of all free variables in ϕ.

(iv) The database must be *independent* of the transaction base $\mathbf{P}$. Intuitively, this means that the database must not define predicates (views) in terms of transactions. Formally, predicate symbols that occur in rule-heads in $\mathbf{P}$ cannot occur in rule-bodies in $\mathbf{D}$.

Conditions (ii) and (iv) apply not only to the current database, $\mathbf{D}$, but to every database mentioned in the transition base, $\mathcal{B}$.

The last condition, independence, arises naturally in two situations: (*i*) when the database is relational (a set of atomic formulas), and (*ii*) when a conceptual distinction is desired between updating actions and non-updating queries. In the former case, the database is trivially independent of $\mathbf{P}$, since each database atom has an empty premise. In the latter case, the logic would have two sorts of predicates, query predicates and action predicates. Action predicates would be defined only in the transaction base, and query predicates would be defined only in the database. Action predicates could be defined in terms of query predicates (e.g., to express pre-conditions and post-conditions), but not vice-versa.

Although serial Horn $\mathcal{TR}$ is a very expressive logic, some useful programs are not Horn. Such programs typically arise when the user applies constraints to the execution of programs that *are* serial-Horn (see Section 2.4). Although constraints can be handled by a general-purpose proof theory of $\mathcal{TR}$ [6], there is a more efficient, special-purpose inference system, which will be reported elsewhere.

5.1 Inference

We now develop an inference system, called $\Im^I$, for checking that $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \models \psi$, *i.e.*, that a transaction, $(\exists)\,\psi$, can successfully execute starting from state $\mathbf{D}_0$. The inference succeeds if and only if it finds an execution path for the transaction ψ, that is, a sequence of databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$. As we shall see, certain inference strategies generate the execution path in a way that corresponds to the intuitive notion of transaction execution. In particular, top-down inference corresponds to forward execution (the normal kind), and bottom-up inference corresponds to reverse execution (i.e., starting from the current state, compute new states by executing all updates in reverse).

In [5], we also introduce a dual system, $\Im^{II}$, which is useful for bottom-up transaction execution. Additionally, when hypothetical transactions are allowed, [5] describes $\Im^{\diamond}$—an elegant inference system that amalgamates $\Im^I$ and $\Im^{II}$, and is complete even in the presence of hypothetical modal operators.

Because we are dealing with the serial Horn case, the transaction ψ is an existential serial conjunction, that is, a formula of the form $(\exists\overline{X})\,(a_1 \otimes a_2 \otimes \ldots \otimes a_m)$, where each a_i is atomic. Since *all* free variables in ψ are assumed to be existentially quantified, we often omit the $\overline{X}$; though as a reminder, we leave $(\exists)$ in front of many transactions. Note that this existential quantification is consistent with the traditions of logic programming and databases. The inference rules below all focus on the left end of such transactions. To highlight this focus, we write serial conjunctions as $\phi \otimes rest$, where ϕ is the piece of the conjunction that the inference system is currently focussed on, and $rest$ is the rest of the conjunction.

Definition 5.1 (Inference) Suppose $\mathcal{B}$ is a transition base, and $\mathbf{P}$ is a transaction base. If they satisfy the serial-Horn conditions, then $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ can be any database mentioned in $\mathcal{B}$.

Axioms: $\quad \mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$ for any $\mathbf{D}$.

Inference Rules: In rules 1–3 below, a and b are atomic formulas that unify with mgu σ, and $rest$ is a serial conjunction of atoms.

1. *Applying transaction definitions:* If $b \leftarrow \phi$ is a rule in $\mathbf{P}$, then

$$\frac{\mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(a \otimes rest)}$$

2. *Querying the database:* If $\mathbf{D} \vdash^c (\exists)\, b\sigma$, where $\vdash^c$ denotes classical provability,[6] then

$$\frac{\mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, rest\,\sigma}{\mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(a \otimes rest)}$$

3. *Performing elementary updates:* If $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle b$ is a transition in $\mathcal{B}$, then

$$\frac{\mathcal{B}, \mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\,\sigma}{\mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(a \otimes rest)}$$

The above presents $\Im^I$ as a natural deduction system. It is not difficult to see that $\Im^I$ has an SLD-style refutational companion. □

This system manipulates expressions of the form $\mathcal{B}, \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi$, called *sequents.* The informal meaning of such a sequent is that the transaction $(\exists)\,\phi$ can *succeed* from $\mathbf{D}$, *i.e.*, it can be executed on a path emanating from the database $\mathbf{D}$. Each inference rule consists of two sequents, one above the other, and has the following interpretation: If the upper sequent can be inferred, then the lower sequent can also be inferred. Starting from the axiom-sequents, the system repeatedly applies the inference rules to infer more sequents.

Notice that Rule 3 changes the database. Since the transition base, $\mathcal{B}$, is infinite, any implementation of this rule would include an algorithm for enumerating the elements of the transition base. Implementing an infinite axiom base by an algorithm in this way is akin to algorithmic implementations of inference rules, such as the resolution rule, which have no finite representation in classical logic. In system $\Im^I$, the enumeration algorithm would take an update, b, and a database state, D, as input, from which it would enumerate all possible successor states, *i.e.*, states, D', such that $\langle D, D' \rangle b$ is in the transition base. For instance, given *ins:d* and $\{a, b, c\}$ as input, the algorithm would return $\{a, b, c, d\}$ as output. Thus, procedures that perform elementary updates are treated as black boxes that can be "plugged into" the inference system.

To understand the inference system, first note that the axioms describe the empty transaction, "( )". This transaction does nothing and always succeeds. The three inference rules describe more complex transactions, capturing the roles of the transaction base, the database, and the transition base, respectively. We can interpret these rules as follows: Rule 1 replaces a subroutine definition, ϕ, by its calling sequence, b; Rule 2 attaches a pre-condition, b, to the front of a transaction, *rest*; and Rule 3 attaches an elementary update, b, to the front of a transaction, *rest*, so that the resulting transaction starts from $\mathbf{D}_1$ instead of $\mathbf{D}_2$. The unifier, σ, makes $\Im^I$ a practical, SLD-style inference system, one that returns most-general-unifiers as answers, as Prolog does (see Section 5.3). As in classical resolution, any instance of an answer-substitution is a valid answer to a query.

[6] Here, we are invoking classical provability as an oracle. Since the database is Horn, the oracle only needs to examine the least fixpoint of the database, since $\mathbf{D} \vdash^c (\exists)\, b$ if and only if b unifies with some atom in *lfp*($\mathbf{D}$). This can be done using SLD-resolution or some other suitable mechanism.

Definition 5.2 (General Deduction) Given an inference system, a *deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, seq_0, seq_1, ... , seq_{n-1}, seq_n, where each seq_i is an axiom or is derived from earlier sequents by an inference rule. □

Theorem 5.3 (Soundness and Completeness) *Let $\mathcal{B}$ be a transition base,* $\mathbf{P}$ *a transaction base, and $(\exists)\,\phi$ a transaction formula. If $\mathcal{B}$,* $\mathbf{P}$*,* $\mathbf{D}$*, and $(\exists)\,\phi$ satisfy the serial-Horn conditions, then the executional entailment $\mathcal{B}, \mathbf{P}, \mathbf{D}\text{ --- } \models (\exists)\,\phi$ holds iff there is a deduction in $\Im^I$ of the sequent $\mathcal{B}, \mathbf{P}, \mathbf{D}\text{ --- } \vdash (\exists)\,\phi$.*

5.2 Execution as Deduction

Having developed the inference system, we must remind ourselves that our original goal was not so much proving statements of the form $\mathcal{B}, \mathbf{P}, \mathbf{D}\text{ --- } \models (\exists)\,\phi$, but rather of the form $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models (\exists)\,\phi$, where $\mathbf{D}_0$ is the database state at the time the user issues the transaction ?– $(\exists)\phi$. Note that the intermediate states, $\mathbf{D}_1$, ... , $\mathbf{D}_{n-1}$, and the final state, $\mathbf{D}_n$, are *unknown* at this time. An important task for the inference system is to compute these states. The general notion of deduction is not tight enough to do this conveniently, since a general deduction may record the execution of many unrelated transactions, mixed up in a haphazard way. Since we are interested in the execution of a particular transaction, we introduce a more specialized notion of *executional deduction* that—without sacrificing completeness—defines a narrower range of deductions than does Definition 5.2.

Definition 5.4 (Executional Deduction) Let $\mathcal{B}$ be a transition base, and $\mathbf{P}$ be a transaction base. An *executional deduction* of a transaction, $(\exists)\,\phi$, is a deduction, seq_0, ... , seq_n, that satisfies the following conditions:

1. The initial sequent, seq_0, has the form $\mathcal{B}, \mathbf{P}, \mathbf{D}'\text{ --- } \vdash (\,)$, for some database $\mathbf{D}'$.
2. The final sequent, seq_n, has the form $\mathcal{B}, \mathbf{P}, \mathbf{D}\text{ --- } \vdash (\exists)\,\phi$, for some database $\mathbf{D}$.
3. Each sequent, seq_i (for $i > 0$), is obtained from the *previous* sequent, seq_{i-1}, by one of the inference rules of system $\Im^I$ (i.e., seq_{i-1} is the numerator of the rule, and seq_i is the denominator).

□

Theorem 5.3 remains valid even if deductions are required to be executional. However, because we now have a stronger form of deduction, we can prove stronger results about it. Theorem 5.3, for instance, does not specify the execution path of the transaction. With executional deduction, we can.

Execution paths can easily be extracted from executional deductions. The key observation is that system $\Im^I$ applies elementary transitions exactly when inference rule 3 is invoked. Invoking this rule during inference is the proof-theoretic analogue of executing an elementary transition. Thus, we need only pick out those points in an executional deduction where inference rule 3 is applied, as in Figure 1. In this figure, we define the *execution path* of the deduction to be the sequence $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$. The next theorem provides a model-theoretic meaning for execution paths. It also relates executional deduction to executional entailment (Definition 4.6).

$$
\begin{array}{llll}
 & \mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\,\phi & & \\
 & \quad \ldots & & \\
 & \mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\,\psi_1 & & \\
 \textit{if} & \mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,\phi_1 & \text{by inference rule 3,} \\
 & \quad \ldots & & \\
 & \mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,\psi_2 & & \\
 \textit{if} & \mathcal{B}, \mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\,\phi_2 & \text{by inference rule 3,} \\
 & \quad \ldots & & \\
 & \mathcal{B}, \mathbf{P}, \mathbf{D}_n \text{---} \vdash (\,) & &
\end{array}
$$

Figure 1: An Executional Deduction in $\Im^I$

Theorem 5.5 (Executional Soundness and Completeness) *Given a transition base, $\mathcal{B}$, and a transaction base, $\mathbf{P}$, there is an executional deduction of $(\exists)\,\phi$ whose execution path is $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ iff the following statement is true:*

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ \ldots\ , \mathbf{D}_n \ \models \ (\exists)\,\phi \tag{6}$$

By constructing executional deductions, we can execute transactions. As Figure 1 shows, by constructing the deduction from the top, down, the database is systematically updated from $\mathbf{D}_0$ to $\mathbf{D}_1$ ... to $\mathbf{D}_n$. We call this *normal* execution. Likewise, by constructing the deduction from the bottom, up, the database is systematically updated from $\mathbf{D}_n$ to $\mathbf{D}_{n-1}$ ... to $\mathbf{D}_0$. We call this *reverse* execution. The process of constructing deductions and executing transactions is developed in detail in [5].

5.3 Example: Inference with Unification

As in Prolog, our inference system returns a substitution. The substitution specifies values of the free variables for which the transaction succeeds. The example here is similar to Example 2.5, in which a robot simulator moves blocks around a table top. We consider a transaction base, $\mathbf{P}$, containing the following rule, which describes the effect of picking up a block, X:

$$\mathit{pickup}(X) \ \leftarrow \ \mathit{clear}(X) \otimes \mathit{on}(X,Y) \otimes \mathit{del{:}on}(X,Y) \otimes \mathit{ins{:}clear}(Y)$$

Suppose we order the robot to pick up a block, by typing $?-\ \mathit{pickup}(X)$. Since the block is left unspecified, the transaction is non-deterministic. The inference system attempts to find a value for X that enables the transaction to succeed, updating the database in the process. To illustrate, suppose the initial database represents an arrangement of three blocks, where *blkA* is on top of *blkC*, and *blkB* stands alone. If the robot picks up *blkA*, the database changes from state $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$, where

$$
\begin{aligned}
\mathbf{D}_0 &= \{\mathit{clear}(\mathit{blkA}),\, \mathit{clear}(\mathit{blkB}),\, \mathit{on}(\mathit{blkA}, \mathit{blkC})\} \\
\mathbf{D}_1 &= \{\mathit{clear}(\mathit{blkA}),\, \mathit{clear}(\mathit{blkB})\} \\
\mathbf{D}_2 &= \{\mathit{clear}(\mathit{blkA}),\, \mathit{clear}(\mathit{blkB}),\, \mathit{clear}(\mathit{blkC})\}
\end{aligned}
$$

Rule	Unifier	Sequent
1		6. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{ --- } \vdash (\exists) pickup(X)$
2		5. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{ --- } \vdash (\exists)[clear(X) \otimes on(X,Y) \otimes del{:}on(X,Y) \otimes ins{:}clear(Y)]$
2	$X/blkA$	4. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{ --- } \vdash (\exists)[on(blkA,Y) \otimes del{:}on(blkA,Y) \otimes ins{:}clear(Y)]$
3	$Y/blkC$	3. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{ --- } \vdash (\exists)[del{:}on(blkA, blkC) \otimes ins{:}clear(blkC)]$
3		2. $\mathcal{B}, \mathbf{P}, \mathbf{D}_1 \text{ --- } \vdash ins{:}clear(blkC)$
		1. $\mathcal{B}, \mathbf{P}, \mathbf{D}_2 \text{ --- } \vdash (\,)$

Figure 2: Executional Deduction of $(\exists)\, pickup(X)$

An executional deduction in which the robot picks up $blkA$ is shown in Figure 2. In this figure, each sequent is derived from the sequent immediately below by an inference rule. Each inference involves unifying the leftmost atom in the transaction against an atom in the database, the transaction base, or the transition base. For example, in deriving sequent 5 from sequent 4, the inference system unifies the atom $clear(X)$ in the transaction against the atom $clear(blkA)$ in the database, $\mathbf{D}_0$. In this way, the system "chooses" to pick up $blkA$. Likewise, in deriving sequent 4 from sequent 3, the inference system unifies the atom $on(blkA, Y)$ in the transaction against the atom $on(blkA, blkC)$ in the database, thus retrieving $blkC$, the block on which $blkA$ is resting. Note that each line in the table shows three items: a numbered sequent, the inference rule used in deriving it from the sequent below, and the unifying substitution. The answer substitution is obtained by composing all the unifiers, which yields $\{X/blkA, Y/blkC\}$, and then projecting onto the substitution for X, which yields $\{X/blkA\}$. The operational interpretation of this proof is that the robot has picked up $blkA$.

6 Comparison with Other Work

As far as logic programming is concerned, we are not aware of any other approach to logics for updates that is as comprehensive as $\mathcal{TR}$. In particular, none of the works discussed below is capable of expressing constraints on the execution of complex transactions. Likewise, none of them can *seamlessly* accommodate hypothetical state transitions with transitions that actually commit and permanently change the current database state; and, with the exception of [13], all of the works are limited to updating sets of ground atomic facts. A more extensive comparison can be found in [5].

Winslett [24] did foundational work on the meaning of updates to general logical theories. Later, Grahne, Katsuno and Mendelzon [13, 10] axiomatized various theories of state transition and studied tractable cases of what we call "elementary state transitions." Our approach to state transitions is inspired by these results.

Manchanda and Warren [17] introduce Dynamic Prolog—a logic system where update transactions "work right," *i.e.*, when failed, they do not leave a residue in the database. Like $\mathcal{TR}$, their logic can be used to update views, and transactions can be nondeterministic.

However, they distinguish between update predicates and query predicates—a drawback if we keep an eye on object-oriented applications, as explained in the introduction. Furthermore, bulk updates, constraints on transaction execution, and the insertion and deletion of rules cannot be expressed, due to the chosen semantics. In addition, the proof theory for Dynamic Prolog is impractical for carrying out updates, since one must know the final database state *before* inference begins. This is because this proof theory is a verification system: Given an initial state, a final state, and an update procedure, the proof theory can *verify* that the procedure causes the transition; but, given just the initial state and the procedure, it cannot *compute* the final state; *i.e.*, it cannot execute procedures, as $\mathcal{TR}$'s proof theory does. Apparently, realizing this drawback, Manchanda and Warren developed an interpreter for "executing" transactions. However, this interpreter is incomplete with respect to the model theory and, furthermore, it is not based on the proof theory of Dynamic Prolog. To a certain extent, it can be said that Manchanda and Warren have managed to formalize their intuition procedurally, but not as an inference system.

Naqvi and Krishnamurthy [21] extended Datalog with update operators, which were later incorporated in the LDL language. Since LDL is geared towards database applications, this extension has bulk updates, for which an operational semantics exists. Unfortunately, the model theory presented in [21] is somewhat limited. First, it matches the operational semantics only in the propositional case, and so does not cover bulk updates. Second, it is only defined for programs in which commutativity of elementary updates can be assumed. For sequences of updates in which this does not hold, the semantics turns out to be rather tricky and certainly does not qualify as "model theoretic." Third, the definition of "legal" programs in [21] is highly restrictive, making it difficult to build complex transactions out of simpler ones.

Chen has developed a calculus and an equivalent algebra for constructing transactions [7]. Like $\mathcal{TR}$, this calculus uses logical operators to construct actions from elementary updates. There are two main differences however. First, the calculus has no analogue of $\mathcal{TR}$'s transition base for specifying elementary updates, so it is restricted to the insertion and deletion of single tuples. The second difference is in the semantics of conjunction, "$\wedge$": Whereas $\mathcal{TR}$ uses "$\wedge$" to express constraints, Chen's calculus uses it to express parallel actions. The main motivation here is that parallel actions make bulk updates easy to express, which is an important database feature. However, there are several disadvantages in the way this is achieved. For one, the calculus cannot express the kind of sophisticated constraints that $\mathcal{TR}$ can. At the same time, sequential updates often achieve the same effect as parallel updates. The way parallel actions are introduced in [7] is undesirable, since it complicates the semantics greatly, making it non-monotonic even in the absence of negation. $\mathcal{TR}$ achieves the same effect in a much simpler, monotonic way [5]. Finally, the syntax of the calculus is not closed. For instance, negation can be applied to some formulas but not others. In particular, if ψ is an updating transaction, then rules like $p \leftarrow \psi$ have no meaning, since this is equivalent to $p \vee \neg\psi$. It therefore seems unlikely that this calculus can be developed into a full-blown *logic* in a straightforward or satisfying way. Furthermore, the calculus itself is very limited as a programming language, since it has no mechanism for defining recursion or subroutines.

McCarty has outlined a logic of action as part of a larger proposal for reasoning about deontic concepts [19]. His proposal contains three distinct layers, each with its own logic: first-order predicate logic, a logic of action, and a logic of permission and obligation. In some ways, the first two layers are similar to $\mathcal{TR}$, especially since the action layer uses logical operators to construct complex actions from elementary actions. Because of his interest on deontic concepts, McCarty defines two notions of satisfaction. In one notion, called "strict satisfaction," conjunction, $\wedge$, corresponds to parallel action, as it does in Chen's work [7]. In the other notion, called "satisfaction," the same symbol corresponds to constraints, as it does in $\mathcal{TR}$. However, since the focus of this work is on strict satisfaction, the development of path constraints was never considered. Also, there is no analogue of $\mathcal{TR}$'s transition base, and the only elementary updates considered correspond to insertion and deletion of atomic formulas. Although obviously interesting, this action logic was not developed in detail. For instance, although a model theory based on sequences of partial states is presented, there is no sound-and-complete proof theory, and no mechanism is presented for executing actions or updating the database. In contrast, the recent work of McCarty and Van der Meyden [20] is much more detailed, but is very different from $\mathcal{TR}$ and is not intended for updating databases.

Many AI workers represent actions using classical predicate logic (situation calculus [18]). Much of this research focuses on general reasoning *about* actions and, as such, is complementary to $\mathcal{TR}$, which focuses on defining transactions and executing them. In these AI formalisms, actions are usually hypothetical, while $\mathcal{TR}$-actions can be *both* hypothetical *and* real. Another important difference is that $\mathcal{TR}$ avoids the infamous *frame problem* of AI, which complicates so much of the work on reasoning about actions. This is possible because $\mathcal{TR}$ performs real updates on materialized databases, just as ordinary programs do. Finally, although the above approaches can perform very general reasoning, they have difficulty with basic database features such as views and post-conditions, especially when recursion is involved [23]. Advanced features of $\mathcal{TR}$, such as path constraints and non-deterministic actions, also seem difficult to formalize in the situation calculus.

Georgeff and Lansky develop a formalism called Procedural Logic whose intent is very similar to that of $\mathcal{TR}$ [9]. However, there are important differences, both practically and philosophically. These difference arise largely because Procedural Logic is actually two separate things: An implementation (called PRS), which provides the functionality; and a formalization, which accounts for only a fraction of that functionality. It is probably fair to say that $\mathcal{TR}$ is a formalization that accounts for most of PRS. The shortcomings of Procedural Logic can be summarized as follows: (*i*) the formalization does not include a database, so updates cannot be defined, (*ii*) the declarative semantics is still very procedural, and seems to anticipate a Prolog-style backtracking interpreter, and (*iii*) the operational semantics is based on an *interpreter*, not a logical inference system, and moreover, this interpreter is incomplete with respect to the "declarative" semantics.

Our path structures are reminiscent of the "path models" in Process Logic [12]. However, there are important differences. First, Process Logic was intended for reasoning about properties of programs, not for updating databases or for programming and executing transactions. Indeed, Process Logic lacks a subroutine facility for defining composite actions, and there is no facility, like $\mathcal{TR}$'s transition base, for defining elementary actions. Second, state changes in Process Logic are hypothetical, and there seems to be no easy to make them permanent.

Abiteboul and Vianu developed a family of update languages [1], and provided impressive results on complexity and expressibility. However, these languages lack several features that are present in $\mathcal{TR}$. First, they apply only to relational databases, not to arbitrary sets of first-order formulas. Second, there is no facility for constraining transaction execution. Indeed, transaction output is the only concern. Third, these languages are not part of a full-blown *logic*: arbitrary logical formulas cannot be constructed, and although there is an operational semantics, there is no model theory and no logical inference system. It is therefore unlikely that these languages have the flexibility to find applications in other domains, such as AI. Finally, these languages do not support transaction subroutines. That is, a transaction cannot be given a name, and it cannot be invoked repeatedly from within the language, facilities that any practical programming language must provide. This lack of subroutines is reflected in the data complexity of some of the languages: they are in PSPACE, whereas recursive subroutines require *alternating* PSPACE, that is, EXPTIME.

The works [22, 8] are related to [1] in that they borrow much of the syntax from deductive databases and yet their semantics is operational (although inspired by logical model theory). As such, these languages are in a different league than $\mathcal{TR}$.

We should also mention works based on Allen's logic of time intervals [2], such as the event calculus of Kowalski and Sergot [16, 15] and the logic of Halpern and Shoham [11]. While the former present a methodology for programming Allen's time intervals in a classical setting, Halpern and Shoham develop a propositional modal logic for them. Although by following these approaches, one can represent some aspects of transaction programming, these approaches are not completely adequate for the task and, at least [11], pursues a different goal. For instance, none of the features listed at the beginning of this section is supported in a straightforward way (and none of these approaches has a straightforward subroutine facility). However, just as in the case of situation calculus, it is likely that the methodology of event calculus could be used in conjunction with $\mathcal{TR}$.

Acknowledgments: Alberto Mendelzon provided us with many insights regarding updates of logic theories. Thanks to Ray Reiter for commenting on various aspects of this paper, especially on the issues related to the frame problem. Mariano Consens took a very close look at the first draft and many improvements are due to his comments. Discussions with Gösta Grahne and Peter Revesz were also helpful for the progress of this work.

References

[1] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 240–250, New York, 1988. ACM.

[2] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, July 1984.

[3] F. Bancilhon. A logic-programming/Object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, September 1986.

[4] C. Beeri. New data models and languages—The challenge. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–15, San Diego, CA, June 1992. ACM.

[5] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-270, University of Toronto, April 1992. Revised: February 1994. *ftp://csri-technical-reports/270/report.ps*.

[6] A.J. Bonner and M. Kifer. A general logic of state change. Technical report, CSRI, University of Toronto, 1995. In preparation.

[7] W. Chen. Declarative specification and evaluation of database updates. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, volume 566 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, December 1991.

[8] C. de Maindreville and E. Simon. Non-deterministic queries and updates in deductive databases. In *Intl. Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, 1988.

[9] M.P. Georgeff and A.L. Lansky. Procedural knowledge. In *Proc. IEEE Special Issue on Knowledge Representation*, volume 74, pages 1384–1398, 1986.

[10] G. Grahne and A.O. Mendelzon. Updates and subjunctive queries. Technical Report KRR-TR-91-4, CSRI, University of Toronto, July 1991.

[11] J.Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 279–292, 1986.

[12] D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.

[13] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 387–394, Boston, Mass., April 1991.

[14] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, May 1995.

[15] R.A. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.

[16] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[17] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[18] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

[19] L.T. McCarty. Permissions and obligations. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 287–294, San Francisco, CA, 1983. Morgan Kaufmann.

[20] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 59–70, Cambridge, MA, October 1992.

[21] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 251–262, New York, March 1988. ACM.

[22] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, New York, 1991. ACM.

[23] R. Reiter. On formalizing database updates: Preliminary report. In *Proc. 3rd Intl. Conf. on Extending Database Technology*, March 1992.

[24] M. Winslett. A model based approach to updating databases with incomplete information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.