1 A LOGIC FOR PROGRAMMING DATABASE TRANSACTIONS

Anthony J. Bonner
and Michael Kifer

Abstract: We propose an extension of classical predicate calculus, called *Transaction Logic*, which provides a logical foundation for the phenomenon of state changes in logic programs and databases. Transaction Logic comes with a natural model theory and a sound and complete proof theory. The proof theory not only verifies programs, but also executes them, which makes this logic an ideal tool for declarative programming of database transactions and state-modifying logic programs. The semantics of Transaction Logic leads naturally to features whose amalgamation in a single logic has proved elusive in the past. These features include hypothetical *and* committed updates, dynamic constraints on transaction execution, non-determinism, 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 and a sound and complete SLD-style proof theory for a Horn-like subset of the logic.

1.1 INTRODUCTION

Updates are a crucial component of any database programming language. Even the simplest database transactions, such as withdrawal from a bank account, require updates. Unfortunately, updates are not accounted for by the classical Horn semantics of logic programs and deductive databases, which limits their usefulness in real-world applications. As a short-term practical solution, logic programming languages have resorted to handling updates with ad hoc operators without a logical semantics. To address this problem, this paper provides the theoretical foundations for logic programming with updates. This is accomplished in three ways: (*i*) we develop a general logic of state change, called *Transaction Logic*, including a natural model theory; (*ii*) we show that this logic has a "Horn" fragment, with both a procedural and a declarative semantics; and (*iii*) we show that programs in the logic can be executed by an SLD-style proof procedure in the logic-programming tradition. The result is a rule-based language with a purely logical semantics (and a sound-and-complete proof theory) in which users can program and execute database transactions. Moreover, in the absence of updates, this language reduces to classical Horn logic. It therefore represents a conservative extension of the logic programming paradigm.

Transaction Logic (or $\mathcal{TR}$ for short) is a general logic of state change that accounts for database updates and transactions and for important related phenomena, such as the order of update operations, transaction abort and rollback, savepoints, and dynamic constraints [BK95; Bon97c]. $\mathcal{TR}$ has applications in many areas, including databases, logic programming, workflow management, and artificial intelligence. These applications, both practical and theoretical, are discussed in detail in [BK95; Bon97c]. For instance, in logic programming, $\mathcal{TR}$ provides a clean, logical alternative to the *assert* and *retract* operators of Prolog. In relational databases, $\mathcal{TR}$ provides a logical language for programming transactions, for updating database views, and for specifying active rules. In object-oriented databases, $\mathcal{TR}$ can be combined with object-oriented logics, such as F-logic [KLW95], to provide a logical account of *methods*—procedures hidden inside objects that manipulate these objects' internal states [Kif95]. In AI, $\mathcal{TR}$ suggests a logical account of procedural knowledge and planning, and of subjunctive queries and counterfactuals.

Other Logics. On the surface, there would seem to be many other logics available for specifying database 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 [Bee92; Ban86; PDR91]. In particular, database transaction languages are not founded on action logics, the way that query languages are founded on classical logic. The main reason, we believe, is that reasoning about action is not the same thing as declarative programming, especially in a database context. This difference manifests itself it several ways:

(*i*) Most logics of action were not designed for database programming. Instead, they were intended for specifying properties of actions or relationships between actions, and for reasoning about them. For instance, one might specify that event A comes before event B, and that B comes before C, and then infer that A comes before C. Moreover, many such logics are propositional, many have no notion of database state or query, and many have no notion of named procedures (such as views and subroutines). Such logics are poor candidates for the job of formalizing database programming languages.

(*ii*) Many logics of action were designed for reasoning about programs. Such logics typically have two separate languages: a procedural language for representing programs, and a logical language for reasoning about their properties. The programs themselves are not declarative or logical at all, but are more akin to Algol. Moreover, logic is not used *inside* programs to specify database queries, but is used *outside* programs to specify program properties. This is the exact opposite of database languages and logic programs. Here, the goal is to make programming as declarative as possible, and often logic itself *is* the programming language, or a significant part of it. The result is that it is difficult to integrate action logics with database query languages and logic programs, since there is an unnatural "impedance mismatch" between them.

(*iii*) Logics of action cannot execute programs and update the database. Instead, the logics are hypothetical. At best, they can infer what *would* be true *if* a program were executed; the database itself is unchanged by such inferences. To actually execute a program and update the database, a separate runtime system is needed outside of the logic. This is contrary to the idea of logic programming, in which the logical proof theory acts as the runtime system, so that programs are executed by proving theorems.

(*iv*) Many logics of action get bogged down by the so-called *frame problem* [MH69; Rei91], the problem of logically specifying what an action does *not* do. For instance, when a robot picks up a block, many things do not change, such as the color of the block, the weight of the block, the number of blocks, etc. Of course, a great many unrelated facts also do not change, such as the mass of the Earth, the number of planets in the solar system, etc. If one is to reason about actions, then these invariants must all be specified as logical axioms (known as *frame axioms*). A great deal of research has been invested into how to do this concisely. Fortunately, frame axioms are not needed if one simply wants to *program* and *execute* transactions. For instance, C programmers do not need to specify frame axioms, and the run-time system does not reason with frame axioms when executing C programs. We show that the same applies to database transactions, if they are expressed in an appropriate logic. In this way, since there are no frame axioms, the frame problem is not an issue.

Prolog. Database transactions can be defined in Prolog via the operators *assert* and *retract*. When these two operators are used to perform updates,[1] Prolog addresses many of the problems listed above, and has many of the properties we wish to model: (*i*) it is a programming language based on a Horn subset of a full logic, (*ii*) programs are executed by an SLD-style proof procedure—not via a separate run-time system, (*iii*) updates are real, not hypothetical, (*iv*) in the absence of updates, it reduces to classical Horn logic, (*v*) the frame problem is not an issue. Unfortunately, updates in Prolog are non-logical operations; so each time a programmer uses *assert* or *retract*, he moves further away from declarative programming. Moreover, Prolog does not support some important features of database transactions, such as abort and rollback. For these reasons, state-changing procedures are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain.

In addition, updates in Prolog are not integrated into the host logical system (*i.e.*, the subset or predicate calculus used in Prolog). It is 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 $\neg assert(X)$? or $assert(X) \leftarrow retract(Y)$? Also, how does one logically account for the fact that the order of updates is important? None of these questions is addressed by Prolog's operational semantics, or by the classical theory of logic programming.

Transaction Logic. $\mathcal{TR}$ provides a general solution to the aforementioned limitations, both of Prolog and of action logics. The solution actually consists of two parts: (*i*) a general logic of state change, and (*ii*) a Horn-like fragment that supports logic programming. In the Horn fragment, users specify and execute transaction programs; and in the full logic, users can express properties of programs and reason about them [BK]. This paper first develops the syntax and semantics of the full logic. The rest of the paper then develops the Horn fragment, and shows that it provides a logic-programming language with updates. A central feature of this development is an SLD-style proof procedure based on unification, a key requirement for any practical logic-programming language. This proof procedure, which is sound and complete, executes logic programs and updates the database as it proves theorems.

The Horn fragment of $\mathcal{TR}$ allows users to combine elementary database operations into complex logic programs. Unlike many logics of action, Horn $\mathcal{TR}$ is not concerned with the axiomatization of elementary operations, but with their logical combination into programs. A logical axiomatization of elementary operations is needed only for reasoning about the properties of actions, not for programming and executing them. Such axiomatizations can be carried out in full $\mathcal{TR}$ [BK], but are not considered in this paper because they are not needed for logic programming. In logic programming and databases (as in C and Pascal), application programmers spend little if any time specifying elementary operations; instead, they devote a great deal of time to combining them into complex transactions and programs. Using logic to build programs from simple operations is thus the main focus of this paper.

Despite our focus on combining operations, the underlying set of elementary operations is an important feature of a programming language, as it determines the domain of application. In practice, elementary operations can vary widely. For example, in C and Pascal, changing the value of a variable is an elementary operation. In Prolog, asserting or retracting a clause is elementary. In database applications, SQL statements are the basic building blocks. In scientific and engineering programs, basic operations include Fourier transforms, matrix inversion, least-squares fitting, and operations on DNA sequences [GRS94; MBDH83]. In workflow management systems, elementary operations can include any number of application programs and legacy systems [BSR96]. In all cases, the elementary operations are building blocks from which larger programs and software systems are built.

Although elementary operations can vary dramatically, the logic for combining them does not. In fact, the same control features arise over-and-over again. These features include sequential composition, iterative loops, conditionals, subroutines and recursion. $\mathcal{TR}$ provides a logical framework in which these and similar control features can be expressed. This framework is *orthogonal* to the elementary operations. $\mathcal{TR}$ can therefore be used with *any* set of elementary database operations, including destructive updates. To achieve this flexibility, $\mathcal{TR}$ treats a database as a collection of abstract data types, each with its own special-purpose access methods. These methods are provided to $\mathcal{TR}$ as elementary operations, and they are combined by $\mathcal{TR}$ programs into complex transactions. This approach separates the specification of elementary operations from the logic of combining them. As we shall see, this separation has two main benefits: (*i*) it allows us to develop a logic programming language for state-changing procedures without committing to a particular theory of elementary updates; and (*ii*) it allows $\mathcal{TR}$ to accommodate a wide variety of database semantics, from classical to non-monotonic to various other non-standard logics. In this way, $\mathcal{TR}$ provides the logical foundations for extending the logic-programming paradigm to a host of new applications in which a given set of operations must be combined into larger programs or software systems.

Due to lack of space, many interesting applications cannot be described here. Likewise, that part of the logic that deals with hypothetical actions is omitted, together with the corresponding applications, such as counterfactuals. The interested reader is referred to [BK95; BKC94] for a development of these aspects of $\mathcal{TR}$. Extensions of $\mathcal{TR}$ for dealing with concurrency and communication are described in [BK96; Bon97c], and complexity results are given in [Bon].[2]

1.2 OVERVIEW AND INTRODUCTORY EXAMPLES

Specifying and executing $\mathcal{TR}$ programs is similar to using Prolog. To specify programs, the user writes a set of logical formulas. These formulas define transactions, including queries, updates, or a combination of both. To execute programs, the user submits a logical formula to a theorem-proving system, which also acts as a run-time system. This system executes transactions, updates the database, and generates query answers, all as a result of proving theorems. Other transactional features such as abort, rollback, and savepoints are also handled by the theorem prover [Bon97c]. This section provides simple examples showing how this kind of behavior can be carried out within a completely logical framework. 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 is useful 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. These choices may be implementation-dependent, but since the whole process is guided by a sound and complete inference system, the database ends up in 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 $\mathcal{TR}$:

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \psi \tag{1.1}$$

Here, ψ is a formula in $\mathcal{TR}$, the $\mathbf{D}_i$ are database states, and $\mathbf{P}$ is a set of $\mathcal{TR}$ formulas (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 database states, representing all the states of transaction execution. In the formal semantics, statement (1.1) means that formula ψ is true with respect to the sequence $\mathbf{D}_0, \ldots, \mathbf{D}_n$. Informally, this means that that the sequence 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 can be a non-deterministic transaction. As such, it can 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.

Unlike many other formalisms, Transaction Logic is neutral on the question of whether queries and updates should be syntactically distinct. $\mathcal{TR}$ is perfectly compatible with such distinctions, but it does not force them upon the user. Formally, all transaction programs in $\mathcal{TR}$ are represented by logical formulas, and there is no built-in class of query formulas or update formulas. In fact, any $\mathcal{TR}$ formula, 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 (1.1) in which $n = 0$. In this case, $\mathbf{D}_0$ is a sequence of databases of length 1.

This approach provides a flexible framework within which users can make many kinds of distinctions, if they wish. For instance, a uniform treatment of queries and updates is needed in the object-oriented domain, because object-oriented systems do not sharply distinguish between state-changing and information-retrieving methods [Kif95]. On the other hand, if a syntactic distinction is desired, then two sorts of predicates could be used, one for queries, and one for updating transactions. This philosophy is quite different from the situation calculus [MH69] and from approaches based on dynamic and process logics [Har79; HKP82], where queries and updates are represented by different classes of syntactic objects (*e.g.*, predicates vs. function terms vs. modal operators).

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. For simplicity, most of the examples are based on Horn $\mathcal{TR}$, and on the insertion and deletion of individual tuples from relational databases. We represent relational databases in the usual way as sets of ground atomic formulas. It should be noted, however, that $\mathcal{TR}$ is restricted neither to relational databases, nor to update operations based on single tuples. For instance, databases could be deductive, object-oriented, disjunctive, or a collection of scientific objects, such as matrices or DNA sequences. Likewise, database operations could include SQL-style *bulk* updates [BK95], or the insertion and deletion of rules, or complex scientific calculations, such as the Fourier transform and matrix inversion.

1.2.1 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*, and we represent them by atomic formulas. These formulas have *both* a truth value *and* a side effect on the database. Formally, we write:

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

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, $p.ins$, to represent insertions into p. Likewise, we use the predicate symbol $p.del$ to represent deletions from p.

Example 1.2.1 (Elementary Updates)
Suppose that *president* is a binary predicate symbol. Then the atoms $president.del(usa, bush)$ and $president.ins(usa, clinton)$ are elementary updates. Intuitively, $president.del(usa, bush)$ means, "delete the

atom $president(usa, bush)$ from the database." Likewise, the atom $president.ins(usa, clinton)$ means, "insert $president(usa, clinton)$ into the database." From the user's perspective, typing $?-\ president.del(usa, bush)$ to the interpreter changes the database from $\mathbf{D}$ to $\mathbf{D} - \{president(usa, bush)\}$. Likewise, typing $?-\ president.ins(usa, clinton)$ changes the database from $\mathbf{D}$ to $\mathbf{D} + \{president(usa, clinton)\}$.[3] We express this behavior formally by the following two statements, which are true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} - \{president(usa, bush)\} \models president.del(usa, bush)$$

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{president(usa, clinton)\} \models president.ins(usa, clinton)$$

□

Here we use "+" and "−" to denote set union and difference, respectively. This is sufficient for relational databases, which are sets of ground atomic formulas. For more complex databases, insertion and deletion are more complex operations [KM92]. 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, $p.ins$ and $p.del$. Our use of these names is merely a convention for purposes of illustration. In fact, p, $p.ins$ and $p.del$ are ordinary predicates of $\mathcal{TR}$, and the connection between them is established via the so-called *transition oracle*, 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:

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

This means that executing the transaction $?-\ \psi \otimes \phi$ changes the database from $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$. Here, $\mathbf{D}_0$ is the initial state, $\mathbf{D}_1$ is an intermediate state, and $\mathbf{D}_2$ is the final state.

Example 1.2.2 (Serial Conjunction)
The expression $came \otimes saw.ins \otimes conquered.ins$, where $came$, saw, and $conquered$ are ground atomic formulas, denotes a sequence of two insertions preceded by a test. This transaction means, "First check that $came$ is true;

then insert *saw* into the database; and then insert *conquered*." Thus, if the initial database is $\mathbf{D}$, and if the user issues a transaction by typing $?-\ came \otimes saw.ins \otimes conquered.ins$, then during execution, the database will change from $\mathbf{D}$ to $\mathbf{D} + \{saw\}$ to $\mathbf{D} + \{saw, conquered\}$, provided that *came* was initially true in $\mathbf{D}$. We express this behavior formally by the following statement, which holds for any transaction base, $\mathbf{P}$, for which $\mathbf{P}, \mathbf{D} \models came$ is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{saw\}, \mathbf{D} + \{saw, conquered\} \ \models \ came \otimes saw.ins \otimes conquered.ins$$

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

1.2.2 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, in this paper, 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, such as those in which actions are modal operators or function terms. 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 1.2.3 (Tossing Coins)
Let $up(coin, head)$ mean that *coin* is lying face up. Likewise, $up(coin, tail)$ means that its tail is facing up. The following transaction base $\mathbf{P}$ defines the action of flipping a coin:

$$\begin{array}{lll} toss(Coin) & \leftarrow & up(Coin, Face) \otimes up.del(Coin, Face) \otimes up.ins(Coin, head) \\ toss(Coin) & \leftarrow & up(Coin, Face) \otimes up.del(Coin, Face) \otimes up.ins(Coin, tail) \\ toss(Coin) & \leftarrow & \end{array}$$

Given a coin, say $dime1$, these rules say that there are three ways to toss $dime1$: one can first determine what side of $dime1$ is facing up; then delete this fact from the database; then either insert $up(dime1, head)$ or insert $up(dime1, tail)$. The third possibility is to do nothing at all to the coin. Thus, $?-\ toss(dime1)$ is a non-deterministic transaction. Formally, if $dime1$ initially has its head side up, then tossing $dime1$ is represented by the following three statements:

$$\mathbf{P}, \{up(dime1, head)\}, \{\}, \{up(dime1, head)\} \models toss(dime1)$$

$$\mathbf{P}, \{up(dime1, head)\}, \{\}, \{up(dime1, tail)\} \models toss(dime1)$$

$$\mathbf{P}, \{up(dime1, head)\} \models toss(dime1)$$

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

1.2.3 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.

Example 1.2.4 (Financial Transactions)
Suppose the balance of a bank account, Act, is given by the relation $balance(Act, Amt)$. To modify this relation, we are provided with a pair of elementary update operations: $balance.del(Act, Amt)$ to delete a tuple from the relation, and $balance.ins(Act, Amt)$ to insert a tuple into the relation. Using these two updates, we define four transactions: $balance.change(Act, Bal, Bal')$, to change the balance of an account; $withdraw(Amt, Act)$, to withdraw an amount from an account; $deposit(Amt, Act)$, to deposit an amount into an account; and $transfer(Amt, Act, Act')$, to transfer an amount from one account to another. These transactions are defined by the following four rules:

$$transfer(Amt, Act, Act') \leftarrow withdraw(Amt, Act) \otimes deposit(Amt, Act')$$

$$\begin{aligned} &withdraw(Amt, Act) \leftarrow \\ &\qquad balance(Act, B) \otimes B \geq Amt \otimes balance.change(Act, B, B - Amt) \end{aligned}$$

$$\begin{aligned} &deposit(Amt, Act) \leftarrow \\ &\qquad balance(Act, B) \otimes balance.change(Act, B, B + Amt) \end{aligned}$$

$$balance.change(Act, B, B') \leftarrow balance.del(Act, B) \otimes balance.ins(Act, B')$$

In the second and third rule, the atom $balance(Act, B)$ acts as a query that retrieves the balance of the specified account. All other atoms are updates. □

The next example uses robot actions to illustrate non-deterministic rules. Planning of robot actions is discussed in detail in [BK95].

Example 1.2.5 (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; $isclear(x)$, which says that nothing is on top of block x; and $wider(x, y)$, which says that x is strictly 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).

$$
\begin{aligned}
&stack(N, X) \ \leftarrow \ N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \\
&stack(0, X) \ \leftarrow \\
&move(X, Y) \ \leftarrow \ pickup(X) \otimes putdown(X, Y) \\
&pickup(X) \ \leftarrow \\
&\qquad isclear(X) \otimes on(X, Y) \otimes on.del(X, Y) \otimes isclear.ins(Y) \\
&putdown(X, Y) \ \leftarrow \\
&\qquad wider(Y, X) \otimes isclear(Y) \otimes on.ins(X, Y) \otimes isclear.del(Y)
\end{aligned}
\tag{1.2}
$$

The actions $pickup(X)$ and $putdown(X, Y)$ mean: "*pick up block X*" and "*put down block X on top of block Y, where Y must be wider than X*," respectively. 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.[4]

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

Observe that rules (1.2) can easily be rewritten in Prolog form, by replacing "$\otimes$" with "," and by replacing the elementary state transitions with *assert* and *retract*. However, the resulting, seemingly 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 $isclear(blkB)$ would stay in the database.

1.2.4 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:

"*Go to the kitchen*" $\wedge$ "*Do not 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. (A $\mathcal{TR}$ formulation of this example is given in formula (1.6) of Section 1.6.)

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 1.2.6 (Transaction Failure)
Consider a pair of sequential transactions, $?\!-\, bought.ins \otimes wanted.ins$ and $?\!-\, wanted.ins \otimes bought.ins$. Both these transactions transform the database from state $\mathbf{D}$ to state $\mathbf{D} + \{bought, wanted\}$. However, they pass through different intermediate states: the former passes through the state $\mathbf{D} + \{bought\}$, while the latter passes through the state $\mathbf{D} + \{wanted\}$. The conjunction $(bought.ins \otimes wanted.ins) \wedge (wanted.ins \otimes bought.ins)$ 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:

$$\begin{aligned} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{bought\}, \mathbf{D} + \{bought, wanted\} &\models bought.ins \otimes wanted.ins \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{wanted\}, \mathbf{D} + \{bought, wanted\} &\models wanted.ins \otimes bought.ins \end{aligned}$$

but the following statement is false for any sequence of databases $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$:[5]

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

□

Example 1.2.7 (Reducing Non-Determinism)
Consider a pair of non-deterministic transactions, $?- lost.ins \vee found.ins$ and $?- lost.ins \vee won.ins$. 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.[6] Hence, following the execution of the transaction $?- (lost.ins \vee found.ins) \wedge (lost.ins \vee won.ins)$ the final database state would be $\mathbf{D} + \{lost\}$. Formally, the following is true:[7]

$$\begin{aligned} \mathbf{P}, \mathbf{D}, \mathbf{D}' &\models (lost.ins \vee found.ins) \wedge (lost.ins \vee won.ins) \\ \textit{iff} &\quad \mathbf{D}' = \mathbf{D} + \{lost\} \end{aligned}$$

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

In [BK95], we explore the potential 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.
- $\phi \otimes \neg\psi$ means transaction ψ must *not* immediately follow transaction ϕ.
- $\neg(\phi \otimes \neg\psi)$ means 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 [BK95] we show that the full set of temporal relationships of Allen's logic of time intervals [All84] has a simple and natural representation in $\mathcal{TR}$.

Besides its $\mathcal{TR}$-specific role in 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.

1.3 SYNTAX

This section begins the formal development of $\mathcal{TR}$.

We define the alphabet of a language of $\mathcal{TR}$ to consist of the following symbols:

- $\mathcal{V}$: a countably-infinite set of variables.
- $\mathcal{F}$: a countably-infinite set of function symbols. Each symbol $f \in \mathcal{F}$ has a non-negative *arity* indicating the number of arguments f can take. There are infinitely many symbols of each arity. Constants are treated as 0-ary function symbols.
- $\mathcal{P}$: a countably-infinite set of predicate symbols. Like functions, predicate symbols have arity, and $\mathcal{P}$ has infinitely many predicate symbols for each arity. 0-ary predicate symbols are viewed as propositional constants.
- Logical connectives $\vee$, $\wedge$ (classical disjunction and conjunction), $\otimes$ (serial conjunction), $\neg$ (classical negation). Additional connectives will be defined in terms of these later.
- Quantifiers $\forall$, $\exists$.
- Auxiliary symbols, such as "(", ")", and ",".

Terms are defined as usual in first-order logic: A variable is a term; if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. When $n = 0$, we write f instead of $f()$. In this paper, we adopt the Prolog convention that variables begin in upper case, and function and constant symbols begin in lower case.

Transaction Formulas $\mathcal{TR}$ extends the syntax of first-order predicate logic with one new binary connective, $\otimes$, called *serial conjunction*. The resulting logical formulas are called *transaction formulas*.

Formally, transaction formulas are defined recursively as follows. First, 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. Second, if ϕ and ψ are transaction formulas, then so are the following expressions:

- $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \otimes \psi$, and $\neg\phi$.
- $(\forall X)\phi$ and $(\exists X)\phi$, where X is a variable.

The following are two examples of transaction formulas:

$$b(X) \otimes c(X,Y) \otimes d(Y) \qquad\qquad \forall X[a(X) \vee \neg b(X) \otimes \neg c(X,Y)]$$

Informally, the formula $\phi \otimes \psi$ means, "*Do* ϕ *and then do* ψ," or slightly more formally, "*at some point in time,* ϕ *finishes and* ψ *starts.*" Note that the classical first-order formulas are transaction formulas that do not use $\otimes$.

As in classical logic, we introduce convenient abbreviations for complex formulas. For instance, $\phi \leftarrow \psi$ is an abbreviation for $\phi \vee \neg\psi$, for all transaction formulas ϕ and ψ. It is also useful to define $\oplus$, called *serial disjunction*, the logical dual of $\otimes$: $\phi \oplus \psi = \neg(\neg\phi \otimes \neg\psi)$. Informally, $\phi \oplus \psi$ means, "*At every point in time,* ϕ *finishes or* ψ *starts.*"

1.4 ELEMENTARY OPERATIONS

Classical logic theories always come with a parameter: a language for constructing well-formed formulas. This language is not fixed, since almost any set of constants, variables and predicate symbols can be "plugged into" it. Likewise, $\mathcal{TR}$ theories are also parameterized by a language. In addition, they have another, *semantic* parameter: a pair of oracles, called the *data oracle* and the *transition oracle*, which specify elementary database operations. The data oracle specifies a set of primitive database *queries*, *i.e.*, the *static* semantics of states; and the transition oracle specifies a set of primitive database *updates*, *i.e.*, the *dynamic* semantics of states. These two oracles encapsulate elementary database operations. Like the language of the logic, the oracles are not fixed and almost any pair of oracles can be "plugged into" a $\mathcal{TR}$ theory.

1.4.1 State Data Oracles

One of the goals underlying the design of $\mathcal{TR}$ is to make it general enough to deal with any kind of database state, including relational databases and more general deductive databases. One may therefore be tempted to define a state as an arbitrary first-order formula and close the issue. However, things turn out to be more involved. For one thing, stating that a database state is a first-order formula does not determine the set of truths about that state. This is because in databases and logic programming, one usually assigns a *non-standard* semantics to database states, *e.g.*, Clark's completion, a perfect-model, or a well-founded model semantics [Llo87; VRS91; GL88]. Because of this, we have chosen to insulate the dynamic aspects of transaction execution from the static aspects pertaining to the truth at database states. Not only does this allow Transaction Logic to work with different database semantics, but it also enables us to study the dynamics and statics of databases separately.

Another problem is that logically equivalent first-order formulas may represent different database states. For instance, in databases and logic programming, $\{p \longleftarrow \neg q\}$ is viewed as a different state than $\{q \longleftarrow \neg p\}$, even though the two formulas are classically equivalent. In the first database, p is considered as true and q as false; in the second state, it is just the opposite.

To achieve the needed generality, the semantics of states is specified by a *state data oracle*. We assume a countable set of symbols, called *state identifiers*, which the oracles use to refer to database states. Note, these symbols are not part of the language of $\mathcal{TR}$, just as the oracles are not part of it. Indeed, transaction formulas in $\mathcal{TR}$ never address database states or oracles directly. However, state identifiers and oracles are used to define the semantics and the proof theory of $\mathcal{TR}$.

Definition 1.4.1 (State Data Oracle) A *state data oracle* is a mapping, $\mathcal{O}^d$, from the set of state identifiers to sets of closed first-order formulas. $\Box$

Intuitively, if i is a state identifier, then $\mathcal{O}^d(i)$ is the set of formulas considered to be all the truths known about the state. In practice, it is not necessary to materialize all these truths. Instead, given a logical formulas ϕ and a state identifier i, the proof theory for $\mathcal{TR}$ only needs to know whether $\phi \in \mathcal{O}^d(\mathbf{D})$. Thus, to do inference in $\mathcal{TR}$, an enumeration of $\mathcal{O}^d(i)$ is all that is needed.

Since the state id uniquely identifies a state, we shall use the terms "state" and "state id" interchangeably. To connote the right intuition, the examples in this paper use first-order formulas (in fact, relational databases) as state identifiers. This is appropriate because many useful oracles can be conveniently described in terms of such formulas. However, our results and definitions do not depend on this representation and, in general, any construct can be a state identifier.

To get a better grasp of the idea of an oracle-as-database-state, Section 1.4.3 provides some typical examples.

1.4.2 State Transition Oracles

The next step is to specify *elementary* changes to the database. One way to define such changes is to build them into the semantics, as in [MW88; NK88; Bon97b; Bon97a; Che91; AV90; McC83]. The problem with this approach is that adding new kinds of elementary transitions requires redefining the very notion of a model and, hence, entails a revamping of the entire theory, including the need to reprove soundness and completeness results. In other words, such theories are not extensible.

The problem is aggravated by the fact that, for arbitrary logical 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? Or, is insertion of $\{q\}$ into $\{p \longleftarrow \neg q\}$ the same as the insertion into $\{q \longleftarrow \neg p\}$? There is no "one true answer" to this question, and many solutions have been proposed (see [KM92] for a comprehensive discussion). Furthermore, Katsuno and Mendelzon [KM92] pointed out that, generally, state transitions belong to two major categories—*updates* and *revisions*—and, even within each category, several different flavours of such transitions are worth looking at. Thus, there appears to be no small, single set of elementary state transitions that is best for all purposes.

For this reason, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat elementary state transitions as a *parameter* of $\mathcal{TR}$. Each set of elementary transitions, thus, gives rise to a different version of the logic. To achieve this, elementary state transitions are specified outside $\mathcal{TR}$, using the notion of a *state transition oracle*. In this way, elementary transitions are separated from the issue of specifying complex transactions. $\mathcal{TR}$ can thus work with any procedural or declarative language for specifying elementary transitions.

Definition 1.4.2 (State Transition Oracle) A *state transition oracle*, $\mathcal{O}^t$, is a mapping from ordered pairs of state identifiers to sets of ground atomic formulas. We refer to these ground atoms as *elementary transitions*. □

Intuitively, if i_1 and i_2 are state identifiers, then $\mathcal{O}^t(i_1, i_2)$ is the set of elementary updates that can transform state i_1 into state i_2. An elementary update can thus be non-deterministic, since for each update, the transition oracle defines a binary relation on states. In practice, this relation does not have to be materialized. Instead, for a given update u, and a given state i_1, the proof theory of $\mathcal{TR}$ only needs an enumeration of the possible successor states, i_2.

Because the transition oracle returns ground atoms only, the elementary updates are completely specified. To see this, suppose that $\mathcal{O}^t(i_1, i_2)$ contained the formula $a \vee b$. Intuitively, this would mean that one of a or b transforms the database from state i_1 to state i_2, but we do not know which one.[8]

Finally, the names of elementary transitions, such as $b.ins$ and $b.del$ in Section 1.2, have *no* special status in $\mathcal{TR}$. That is, they are ordinary atomic formulas that just happen to be mentioned by the oracle. In principle, nothing prevents the user from putting the rules for $b.ins$ into the transaction base.[9] Even the fancy names of these predicates is nothing but a convention adopted in this paper for illustrative purposes.

1.4.3 Examples

This section gives examples of data and transition oracles. In the examples, a database state is a set of data items, which can be any persistent object, such as a tuple, a disk page, a file, or a logical formula. Formally, however, a database state has no structure, and our only access to it is through the two oracles. Some of the oracles below can be combined to yield more powerful oracles. Typically, such combinations are possible when oracles operate on disjoint domains of data items.

Relational Oracles. A state identifier $\mathbf{D}$ is a set of ground atomic formulas. The data oracle simply returns all these formulas. Thus, $\mathcal{O}^d(\mathbf{D}) = \mathbf{D}$. Moreover, for each predicate symbol p in $\mathbf{D}$, the transition oracle defines two new predicates, $p.ins$ and $p.del$, representing the insertion and deletion of single atoms, respectively. Formally, $p.ins(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 + \{p(\overline{x})\}$. Likewise, $p.del(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 - \{p(\overline{x})\}$. SQL-style bulk updates can also be defined by the transition oracle [BK95; BKC94], as can primitives for creating new constant symbols.

Scientific Oracles. A state is a set of square matrices. For each matrix, B, in a state, the data oracle defines two ternary relations, b and $b.dft$, representing the matrix itself and its two-dimensional discrete Fourier transform, $dft(B)$, respectively. Formally, $b(i,j,v) \in \mathcal{O}^d(\mathbf{D})$ iff $B(i,j) = v$ in $\mathbf{D}$. Likewise, $b.dft(i,j,v) \in \mathcal{O}^d(\mathbf{D})$ iff $dft(B)(i,j) = v$. In this way, the data oracle provides two built-in views of each matrix.[10] The transition oracle defines three predicates, $b.set$, $b.rswap$ and $b.cswap$, which update matrix b. The first predicate sets the value of an element of the matrix. Formally, $b.set(i,j,v) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_1$ is just like $\mathbf{D}_2$, except that $B(i,j) = v$ in state $\mathbf{D}_2$. Likewise, $b.rswap(i,j)$ swaps rows i and j of the matrix, while $b.cswap(i,j)$ swaps columns i and j. Note that for main-memory systems, these updates can be implemented with an efficiency comparable to that of variable assignment, *i.e.*, much more efficiently than assert and retract in Prolog.

Classical Oracles. A state $\mathbf{D}$ is a consistent set of variable-free, classical first-order formulas. The data oracle defines all the logical implications of these formulas. Thus $\mathcal{O}^d(\mathbf{D}) = \{\psi \mid \mathbf{D} \models^c \psi\}$, where $\models^c$ denotes classical entailment. The transition oracle defines primitives for adding and removing formulas from the database, resolving any conflicts between the new formulas and existing formulas. Such conflicts can be resolved in numerous ways, as shown by Katsuno and Mendelzon [KM92]. For instance, for each first-order formula, μ, the transition oracle could define four predicates, $update[\mu]$, $erase[\mu]$, $revise[\mu]$ and $contract[\mu]$ for doing updates, erasure, revision, and contraction as defined in [KM92].

Well-Founded Oracle. A state id $\mathbf{D}$ is a set of generalized-Horn rules,[11] and $\mathcal{O}^d(\mathbf{D})$ is the set of literals (both positive and negative) in the well-founded model of $\mathbf{D}$ [VRS91]. Such oracles can represent any rule-base with well-founded semantics, which includes Horn rule-bases, stratified rule-bases, and locally-stratified rule-bases. For advanced applications, one may want to augment $\mathcal{O}^d(\mathbf{D})$ with the rules in $\mathbf{D}$. The transition oracle provides primitives for adding and deleting clauses to/from states.

Generalized-Horn Oracles. A state id $\mathbf{D}$ is a set of generalized-Horn rules and $\mathcal{O}^d(\mathbf{D})$ is a classical Herbrand model of $\mathbf{D}$. Such oracles can represent Horn rule-bases, stratified rule-bases, locally-stratified rule-bases, rule-bases with stable-model semantics [GL88], and any rule-base whose meaning is given by a classical Herbrand model. Again, one may want to augment $\mathcal{O}^d(\mathbf{D})$ with the rules in $\mathbf{D}$. The transition oracle provides primitives for adding and deleting clauses from states.

1.4.4 The Pragmatics of Oracles

Unlike the formulas in a transaction base, we do not expect the oracles to be coded by casual users. Although the oracles allow for many different semantics of states and state changes, we envision that any logic programming system based on $\mathcal{TR}$ will likely have a carefully selected repertoire of built-in database semantics and a tightly controlled mechanism for adding new ones. This latter mechanism would not be available to ordinary programmers. For this reason, we assume in this paper that the data and transition oracles are fixed.

Unfortunately, there is no general solution to the practical problem of *how* oracles can best be implemented. For the classical oracle described above, the problem has been partly solved by Grahne, Mendelzon, and Winslett [GM95; Win88]. Winslett showed that, in general, the problem of updating propositional formulas is NP-hard. Subsequently, though, Grahne and Mendelzon proved that updating sets of ground atoms with arbitrary propositional formulas can be done in polynomial time. More importantly, this result carries over to deductive databases, in which only the extensional part is updated. In this case, as with relational databases, updates are fast and straightforward.

By design, the issue of specifying and implementing elementary operations is orthogonal to our work. In $\mathcal{TR}$, the data and transition oracles are external parameters, and all that matters practically is the existence of an algorithm to compute the outcome of an operation, or to enumerate the possible outcomes if the operation is non-deterministic. Finally, we note that transaction definitions are independent of the oracles. This latter point contributes to making $\mathcal{TR}$ a lucid and flexible language for defining transaction programs.

1.5 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*
- *Database States*
- *Executional Entailment*

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 $?- \ a.ins \otimes b.ins \otimes c.ins$ takes a relational 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 satisfies 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 $\langle \{\}, \{a\}, \{a, b\} \rangle$ satisfies the formula $a.ins \otimes b.ins$. Intuitively, the formula represents a transaction that first inserts a and then inserts b, and the path represents a complete execution of this transaction. In contrast, the shorter paths $\langle \{\}, \{a\} \rangle$ and $\langle \{a\}, \{a, b\} \rangle$ do *not* satisfy this formula, because they do not represent complete executions of the transaction. This example illustrates a general property of $\mathcal{TR}$: a formula may be true on a path, but false on all its proper subpaths.

The kind of transaction that a formula represents depends on the paths that satisfy it. If the paths are of length 1 (*i.e.*, consist of a single state), then the transaction is a query; if the paths are of length 2, then the transaction is (usually) an elementary update; and if the paths are of length greater than 2, then the transaction is a composite update. If the paths are of various lengths, then different executions of the transaction program correspond to different kinds of transaction. In this way, one model-theoretic device, paths, accounts for queries, updates, 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. Unlike modal logic, where the set of states may vary from semantic structure to semantic structure, in $\mathcal{TR}$ the set of states is determined by the data oracle. Changing the oracles can change the set of states, and thus the set of semantic structures. This is one way in which different oracles give rise to different versions of $\mathcal{TR}$.

Executional Entailment. In most logics of action, the notion of "truth" is associated with properties of actions, and formulas are evaluated at states. For instance, in Dynamic Logic, $[\alpha]\phi$ is true in the current state if ϕ is true in the state that results from executing the update α. In $\mathcal{TR}$, the notion of truth is associated with execution, and formulas are evaluated on *paths*, *i.e.*, at sequences of states. The notion of executional entailment provides a logical account of execution, and is described in more detail in Section 1.6.2.

1.5.1 Path Structures and Models

This section makes the preceding discussion precise.

The formal definition of path structures relies on the familiar notion of classical first-order semantic structures [End72]. The symbol $\models^c$ denotes satisfaction in these structures, *i.e.*, classical satisfaction. For our purposes, it is convenient to augment these classical structures with a *special*, abstract structure, denoted $\top$. We define $\top$ to satisfy every first-order formula. Even though $\top$ is not a classical structure, we shall call it "classical" because adding it to classical logic does not change the logic in any essential way—it simply adds one more model to every formula (and some notions, such as satisfiability and consistency, require minor adjustments). Having $\top$ is convenient because it provides a degree of tolerance to inconsistency that may exist between database states and views over them.[12] It is also a simple adaptation of techniques used in paraconsistent logics (*e.g.*, [KL92]), which analyze the knowledge contained in inconsistent states. The reader is referred to [BK95; BK94] for further discussion.

As described in Section 1.3, $\mathcal{TR}$ comes with a language, $\mathcal{L}$ (which determines the syntax of formulas), and with a pair of oracles, $\mathcal{O}^d$ and $\mathcal{O}^t$ (which determine the semantics of databases), and these oracles come with a set of database state identifiers (or states). We define a *path* of length k (or *k-path*) to be a finite sequence of k states, $\langle \mathbf{D}_1, \ldots, \mathbf{D}_k \rangle$, where $k \geq 1$. In the rest of this paper, the language $\mathcal{L}$, the oracles, the set of database states, and the corresponding set of paths are implicit.

In the following definition, each path structure has a domain of objects and an interpretation for all function symbols. These are used to interpret formulas on every path in the structure.

Definition 1.5.1 (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 $\langle U, I_{\mathcal{F}}, I_{path} \rangle$, 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 classical 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. In accordance with our earlier remark, we also assume that $Struct(U, I_{\mathcal{F}})$ contains the special "classical" structure $\top$.
- I_{path} is a total mapping that assigns to every path a first-order semantic structure in $Struct(U, I_{\mathcal{F}})$, subject to the following restrictions:
  - *Compliance with the data oracle*:
    $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$ for every formula $\phi \in \mathcal{O}^d(\mathbf{D})$.
  - *Compliance with the transition oracle*:
    $I_{path}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle) \models^c b$ for every atom $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$. □

The mapping I_{path} serves as the semantic link between transactions and paths: Given a path and a transaction formula, I_{path} determines whether the formula is true on the path (Definition 1.5.2, below). Intuitively, the first compliance restriction says that path $\langle \mathbf{D} \rangle$ provides a "window" onto database $\mathbf{D}$, since any formula that is true of $\mathbf{D}$ is also true of path $\langle \mathbf{D} \rangle$. The second compliance restriction says that elementary updates do what the transition oracle claims they do.

Two points about path structures are worth noting. Both points reflect a flexibility built into path structires that allows them to model the knowledge encoded in a transaction base:

1. Compliance with the oracles is one way. Thus, the formulas in $\mathcal{O}^d(\mathbf{D})$ are not the only formulas that can be true on path $\langle \mathbf{D} \rangle$. Likewise, the formulas in $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ are not the only formulas that can be true on path $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$. This is because the oracles are not the only source of

formulas. In particular, the transaction base also supplies formulas, and these formulas are true on all paths, as we shall see.

2. 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. Therefore, it is the definition of *satisfaction* that relates paths with their subpaths, as we shall see.

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 $\pi = \gamma \circ \delta$.

As in classical logic, we use variable assignments to define the semantics of open and quantified formulas. A *variable assignment* ν is a mapping, $\mathcal{V} \longmapsto U$, that takes a variable as input, and returns a domain element as output. This mapping extends 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 1.5.2 (Satisfaction) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{path} \rangle$ be a path structure, let π be a path in $\mathbf{M}$, and let ν be a variable assignment. Then,

1. $\mathbf{M}, \pi \models_\nu b$ if and only if $I_{path}(\pi) \models^c_\nu b$, where b is an atomic formula.

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

3. $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.

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

5. $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ 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.

6. The meaning of "$\oplus$" is dual to that of "$\otimes$":
$\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ if and only if $\mathbf{M}, \gamma \models_\nu \phi$ or $\mathbf{M}, \delta \models_\nu \psi$ for *every* split $\gamma \circ \delta$ of path π.

7. $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment, μ, that agrees with ν on all variables except X.

8. The meaning of "$\exists$" is dual to that of "$\forall$":
$\mathbf{M}, \pi \models_\nu (\exists X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *some* variable assignment, μ, that agrees with ν on all variables except X.

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. □

Many of the items in Definition 1.5.2 can be interpreted in terms of programming languages. For instance, item 5 establishes a relationship between a path and its subpaths. This corresponds to the relationship between a program and its components. Intuitively, the formula $\phi \otimes \psi$ is a program, and it can execute on a path if the path corresponds to an execution of ϕ followed by an execution of ψ. As another example, item 1 allows atoms to be true on arbitrary paths. In Horn $\mathcal{TR}$, these atoms play the role of subroutine calling sequences. Intuitively, if $p(t_1, \ldots, t_n)$ is an atom, then p is the subroutine name, and $t_1, \ldots, t_n$ are its arguments, exactly as in classical logic programming. Executing this subroutine corresponds to finding a path on which $p(t_1, \ldots, t_n)$ is true.

Definition 1.5.3 (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 $\oplus$, we obtain another interesting pair of serial connectives: the *left serial implication*, $\psi \Leftarrow \phi$, which stands for $\psi \oplus \neg\phi$, and the *right* serial implication, $\phi \Rightarrow \psi$, which denotes $\neg\phi \oplus \psi$. 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:

$$
\begin{array}{lcl}
(\phi \vee \psi) \otimes \eta & \longleftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\
(\phi \wedge \psi) \oplus \eta & \longleftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\
(\phi \vee \psi) \oplus \eta & \longleftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta) \\
(\phi \wedge \psi) \otimes \eta & \longrightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta)
\end{array}
\tag{1.3}
$$

The following three dualities are also easy to verify:

$$
\phi \vee \psi \leftrightarrow \neg(\neg\phi \wedge \neg\psi) \qquad \phi \oplus \psi \leftrightarrow \neg(\neg\phi \otimes \neg\psi) \qquad \exists X \phi \leftrightarrow \neg\forall X \neg\phi
$$

1.5.2 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 two distinct parts: a transaction base $\mathbf{P}$ and an initial database state $\mathbf{D}$. Of these parts, only the database is updatable. 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 (or, more precisely, what the data oracle tells $\mathcal{TR}$ about it) consists entirely of classical formulas.

Definition 1.5.4 (Executional Entailment) Let $\mathbf{P}$ be a transaction base, 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

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{1.4}$$

is true if and only if $\mathbf{M}, \langle \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \rangle \models \phi$ for every model $\mathbf{M}$ of $\mathbf{P}$. Related to this is the following statement:

$$\mathbf{P}, \mathbf{D}_0\text{---} \models \phi \tag{1.5}$$

which is true iff there is a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ that makes (1.4) true. $\square$

Intuitively, statement (1.4) means that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1$ ... to $\mathbf{D}_n$. Formally, it means that every model of $\mathbf{P}$ satisfies ϕ on the path $\langle \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \rangle$.

Normally, users issuing transactions know only the initial database state $\mathbf{D}_0$; they do not know the execution path in advance, and, in most cases, they just want to reach a final state in the execution. To account for this situation, the version of entailment in (1.5) allows us to omit the intermediate and the final database states. Intuitively, statement (1.5) 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 (1.5) is not true, we say that transaction ϕ *fails*. In Section 1.6, we present an inference system that allows us to *compute* a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ that satisfies statement (1.4) whenever a transaction succeeds.

Example 1.5.5 (Executional Entailment) Suppose $\mathbf{P}$ contains the following rules:

$$q \leftarrow r \qquad q \leftarrow s \qquad r \leftarrow a.ins \otimes b.ins \qquad s \leftarrow a.del \otimes b.del$$

Using the relational oracle described in Section 1.4.3, the following statements are all true:

$$\mathbf{P}, \{\}, \{a\}, \{a, b\} \models a.ins \otimes b.ins \qquad \mathbf{P}, \{a, b\}, \{b\}, \{\} \models a.del \otimes b.del$$
$$\mathbf{P}, \{\}, \{a\}, \{a, b\} \models r \qquad \mathbf{P}, \{a, b\}, \{b\}, \{\} \models s$$
$$\mathbf{P}, \{\}, \{a\}, \{a, b\} \models q \qquad \mathbf{P}, \{a, b\}, \{b\}, \{\} \models q$$

$$\mathbf{P}, \{\}, \{a\}, \{a, b\}, \{b\}, \{\} \models r \otimes s$$
$$\mathbf{P}, \{\}, \{a\}, \{a, b\}, \{b\}, \{\} \models q \otimes q$$

Hence, the following statements are true as well:

$$\mathbf{P}, \{\}\text{---} \models a.ins \otimes b.ins \qquad \mathbf{P}, \{a, b\}\text{---} \models a.del \otimes b.del$$
$$\mathbf{P}, \{\}\text{---} \models r \qquad \mathbf{P}, \{a, b\}\text{---} \models s$$
$$\mathbf{P}, \{\}\text{---} \models q \qquad \mathbf{P}, \{a, b\}\text{---} \models q$$
$$\mathbf{P}, \{\}\text{---} \models r \otimes s \qquad \mathbf{P}, \{\}\text{---} \models q \otimes q$$

□

Lemma 1.5.6 (Basic Properties of Executional Entailment) *For any transaction base* $\mathbf{P}$*, any database sequence* $\mathbf{D}_0, \ldots, \mathbf{D}_n$*, and any closed transaction formulas* α *and* β*, the following statements are all true:*

1. *If* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ *and* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.
2. *If* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ *and* $\mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.
3. *If* $\alpha \leftarrow \beta$ *is in* $\mathbf{P}$ *and* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.
4. *If* $\mathcal{O}^t(\mathbf{D}_0, \mathbf{D}_1) \models^c \alpha$ *then* $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.
5. *If* $\mathcal{O}^d(\mathbf{D}_0) \models^c \alpha$ *then* $\mathbf{P}, \mathbf{D}_0 \models \alpha$.

In the last two items, α *is a first-order formula and* $\models^c$ *denotes classical entailment.*

Note that Lemma 1.5.6 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 update the database, *i.e.*, in which it acts as a query. In this case, classical and serial conjunction are identical:

Lemma 1.5.7 (Conjunctive Queries) *For any transaction base* $\mathbf{P}$*, any database state* $\mathbf{D}$*, and any transaction formulas* α *and* β*,*

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

Intuitively, this lemma says that the result of evaluating a conjunctive query is the same whether the conjuncts are evaluated sequentially or in classical fashion. In fact, we can prove a stronger result to show that in the absence of updates, serial conjunction reduces to classical conjunction, serial disjunction reduces to classical disjunction, and executional entailment reduces to classical entailment:

Lemma 1.5.8 (Relationship to Classical Logic) *Let* $\mathbf{P}$ *be a transaction base, and let* $\mathbf{P}^c$ *be a set of first-order formulas derived from* $\mathbf{P}$ *be replacing each occurrence of* $\otimes$ *by* $\wedge$*, and each occurrence of* $\oplus$ *by* $\vee$*. Then, for any first-order formula* α*, and any database* $\mathbf{D}$*,*

$$\mathbf{P}, \mathbf{D} \models \alpha \quad \textit{if and only if} \quad \mathbf{P}^c \cup \mathcal{O}^d(\mathbf{D}) \models^c \alpha$$

Moreover, in the special case of classical oracles,

$$\mathbf{P}, \mathbf{D} \models \alpha \quad \textit{if and only if} \quad \mathbf{P}^c \cup \mathbf{D} \models^c \alpha$$

1.6 PROOF THEORY

Like classical logic, $\mathcal{TR}$ has a Horn subset, called *serial-Horn* $\mathcal{TR}$, which has a simple SLD-style proof theory that gives serial-Horn programs a procedural 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. Unlike classical logic programming, the proof procedure presented in this section computes new database states *as well as* query answers. A detailed development along with a proof of soundness and completeness can be found in [BK95].

Serial-Horn programs are based on the idea of a *serial goal*. A serial goal is a transaction formula of the form $a_1 \otimes a_2 \otimes ... \otimes a_n$, where each a_i is an atomic formula and $n \geq 0$. When $n = 0$, we often write $()$, which denotes the empty goal. A *serial-Horn rule* has the form $b \leftarrow a_1 \otimes a_2 \otimes ... \otimes a_n$, where the body, $a_1 \otimes a_2 \otimes ... \otimes a_n$, is a serial goal and the head, b, is an atom. All the rules in Section 1.2.3 are serial-Horn. Finally, a serial-Horn transaction base is simply a finite set of serial-Horn rules. Observe that a serial-Horn transaction base can be transformed into a classical Horn rulebase, by replacing each occurrence of $\otimes$ by $\wedge$. This transformation changes the serial-Horn rule $b \leftarrow a_1 \otimes ... \otimes a_n$ into the classical Horn rule $b \leftarrow a_1 \wedge ... \wedge a_n$. Lemma 1.5.8 thus implies that, in the absence of updates, executional entailment in Horn $\mathcal{TR}$ reduces to ordinary entailment in classical Horn logic. Classical Horn logic is thus a special case of serial-Horn $\mathcal{TR}$.

In the Horn fragment of $\mathcal{TR}$, database states are represented by the Generalized Horn Oracle described in Section 1.4.3. We say that the combination of a transaction base $\mathbf{P}$ and a generalized Horn data oracle $\mathcal{O}^d$ is *serial-Horn* if $\mathbf{P}$ is a set of serial-Horn rules satisfying the following *independence condition*:

> *For every database state* $\mathbf{D}$*, predicate symbols occurring in rule-heads in* $\mathbf{P}$ *do not occur in rule-bodies in* $\mathcal{O}^d(\mathbf{D})$*.*

Intuitively, the independence condition means that the database does not define predicates in terms of transactions. Thus, the rule $a \leftarrow b$ *cannot* be in the database if the rule $b \leftarrow c$ is in the transaction base (although the rule $b \leftarrow a$ can be in the database).

The independence condition 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.

The serial-Horn conditions support the essential features of the logic programming paradigm. For instance, like classical Horn rules, these conditions can be viewed as a demand for complete information: serial-Horn rules imply that actions are completely specified, and generalized Horn oracles imply that database states are completely specified, *i.e.*, do not contain any indefinite or disjunctive information. As described below, these conditions lead to a simple and practical, SLD-style inference system, in the logic-programming tradition. In addition, the serial-Horn conditions can be extended to accomodate negation-as-failure. This is possible because, like classical Horn rules, a set of serial-Horn rules has a unique minimal Herbrand model. In fact, much of the theory of negation in classical logic programs carries over to Transaction logic programs in a straightforward way, including familiar notions like stratification [ABW88] and local stratification [Prz88]. However, negation is not the subject of this paper, and the interested reader is referred to [BK94; BK95] for details.

Although serial-Horn $\mathcal{TR}$ is a very expressive logic, some useful programs are non-Horn. Programs with negated premises are just one example. Other examples involve dynamic constraints applied to programs that *are* serial-Horn (see Section 1.2.4). For instance, suppose that the predicate $\mathtt{goto}(L)$ is defined by a serial-Horn program that intuitively means, "Go to location L." Then, the following non-Horn formula intuitively means, "Go to the kitchen without passing through the bedroom":

$$\texttt{goto}(kitchen) \wedge \neg(\texttt{path} \otimes \texttt{at}(bedroom) \otimes \texttt{path}) \tag{1.6}$$

Here, path is an abbreviation for $p \vee \neg p$, which is true on all paths. Observe that in addition to serial conjunction, this formula involves classical conjunction, disjunction and negation. These kinds of dynamic constraints are described in detail in [BK95].

1.6.1 Inference

We now describe an inference system, called $\Im^I$, for checking statements of the form $\mathbf{P}, \mathbf{D}_0\text{---} \models (\exists)\psi$, *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 $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$. We shall see that 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 (as described later).

In [BK95], we also introduce a dual system, $\Im^{II}$, which is useful for bottom-up transaction execution. Additionally, when hypothetical transactions are allowed, [BK95] describes $\Im^{\diamond}$—a uniform inference system that amalgamates $\Im^I$ and $\Im^{II}$, and is complete even in the presence of hypothetical modal operators. These systems are all formulated as natural deduction systems. However, it is also possible to formulate them as refutation systems.

In 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 1.6.1 (Inference) If $\mathbf{P}$ is a transaction base, then $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms: $\mathbf{P}, \mathbf{D}\text{---} \vdash ()$

Inference Rules: In Rules 1–3 below, σ is a substitution, a and b are atomic formulas, and ϕ and $rest$ are serial goals.

1. *Applying transaction definitions:*
Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then

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

2. *Querying the database:*
If $b\sigma$ and $rest\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists) b\sigma$, then

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

3. *Performing elementary updates:*
If $b\sigma$ and $rest\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) b\sigma$, then

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

□

Inference system $\Im^I$ manipulates expressions of the form $\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\phi$, 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 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.

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 a. Rule 2 attaches a pre-condition, b, to the front of a transaction $rest$. Rule 3 is the only one that can change the current database state; it attaches an elementary update, b, to the front of a transaction, $rest$, so that the resulting transaction starts from state $\mathbf{D}_1$ 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 1.6.3). As in classical resolution, any instance of an answer-substitution is a valid answer to a query.

Definition 1.6.2 (General Deduction) Given an inference system, a *deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, $seq_0, seq_1, \ldots, seq_{n-1}, seq_n$, where each seq_i is an axiom or is derived from earlier sequents by an inference rule. □

Theorem 1 (Soundness and Completeness) *Under the serial-Horn conditions, the executional entailment* $\mathbf{P}, \mathbf{D}\text{---} \models (\exists)\phi$ *holds if and only if there is a deduction in* $\Im^I$ *of the sequent* $\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\phi$.

1.6.2 Execution as Deduction

Having developed the inference system for proving statements of the form $\mathbf{P}, \mathbf{D}\text{---} \models (\exists)\phi$, we come back to our original problem: proving statements of the form $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models (\exists)\phi$, where $\mathbf{D}_0$ is the initial database state, *i.e.*, the state at the moment when transaction $(\exists)\phi$ began executing. Note that at this moment, the intermediate states, $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$, and the final state, $\mathbf{D}_n$, are still *unknown*. 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 1.6.2.

Definition 1.6.3 (Executional Deduction) Let $\mathbf{P}$ be a transaction base. An *executional deduction* of a transaction, $(\exists)\phi$, is a deduction, $seq_0, \ldots, seq_n$, that satisfies the following conditions:

1. The initial sequent, seq_0, is an axiom.
2. For $i > 0$, each sequent, seq_i, 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).
3. The final sequent, seq_n, has the form $\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\phi$, for some database $\mathbf{D}$.

□

Theorem 1 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 1, 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

m.		$\mathbf{P}, \mathbf{D}_0\text{---} \vdash (\exists)\phi$	
		$\ldots$	
j+1.		$\mathbf{P}, \mathbf{D}_0\text{---} \vdash (\exists)\psi_1$	
j.	*if*	$\mathbf{P}, \mathbf{D}_1\text{---} \vdash (\exists)\phi_1$	by inference rule 3,
		$\ldots$	
i+1.		$\mathbf{P}, \mathbf{D}_1\text{---} \vdash (\exists)\psi_2$	
i.	*if*	$\mathbf{P}, \mathbf{D}_2\text{---} \vdash (\exists)\phi_2$	by inference rule 3,
		$\ldots$	
0.		$\mathbf{P}, \mathbf{D}_n\text{---} \vdash ()$	

Figure 1.1 Construction of an Executional Deduction in $\Im^I$

This deduction involves m inferences and $m+1$ sequents, where 0 is the initial sequent, and m is the final sequent. The deduction includes n changes of state ($0 \leq n \leq m$), each carried out by inference rule 3, such as the inferences from sequent i to $i+1$, and from sequent j to $j+1$.

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 1.1. 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 1.5.4).

Theorem 2 (Executional Soundness and Completeness) *Under the serial-Horn conditions, the executional entailment* $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\phi$ *holds if and only if there is an executional deduction of* $(\exists)\phi$ *whose execution path is* $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.

By constructing executional deductions, we can execute transactions. As Figure 1.1 shows, by constructing the deduction from the top, down, the database is systematically updated from $\mathbf{D}_0$ to $\mathbf{D}_1$ $\ldots$ to $\mathbf{D}_n$. We call this *forward* execution. Likewise, by constructing the deduction from the bottom, up, the database is systematically updated from $\mathbf{D}_n$ to $\mathbf{D}_{n-1}$ $\ldots$ to $\mathbf{D}_0$. We call this *reverse* execution. The process of constructing deductions and executing transactions is developed in detail in [BK95].

1.6.3 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 1.2.5, 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:

$$pickup(X) \ \leftarrow \ isclear(X) \otimes on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y)$$

Suppose we order the robot to pick up a block, by typing $?- \, pickup(X)$. 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

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

An executional deduction in which the robot picks up $blkA$ is shown in Figure 1.2. In this figure, each sequent is derived from the sequent immediately below by an inference rule, and the bottom-most sequent is an axiom. Each inference involves unifying the leftmost atom in the transaction against either a database fact (returned by the data oracle), a transaction name (defined in the transaction base), or an elementary transition (returned by the transition oracle). For example, in deriving sequent 4 from sequent 3, the inference system unifies the atom $isclear(X)$ in the transaction against the atom $isclear(blkA)$ in the database, $\mathbf{D}_0$. In this way, the system "chooses" to pick up $blkA$. Likewise, in deriving sequent 3 from sequent 2, the inference system unifies the atom $on(blkA, Y)$ in the transaction against the atom $on(blkA, blkC)$ in the database. In this way, the system retrieves $blkC$, 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 $blkA$.

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

Figure 1.2 Executional Deduction of $(\exists)pickup(X)$

1.7 RELATED WORK

There is a vast amount of related research. This section examines a selection of closely related works. A more comprehensive comparison can be found in [BK95]. For convenience, we divide the formalisms into two classes: those aimed at specifying database transactions, and those aimed at reasoning about programs.

At the outset, we should mention one feature that distinguishes $\mathcal{TR}$ from all the formalisms described below: its use of state and transition oracles, which support arbitrary notions of state and update. In contrast, many formalisms assume that a state is a relational database, and that updates are limited to the insertion and deletion of tuples. Thus, there is no support for inserting rules into a deductive database, or for inserting disjunctions into a disjunctive database. Many other formalisms are not database-oriented at all, but come from the tradition of procedural programming languages. Typically, they assume that a state is a set of program variables, and that an update changes a variable's value.

1.7.1 Declarative Languages for Database Transactions

Dynamic Prolog. Manchanda and Warren [MW88] developed Dynamic Prolog, a logic programming language for database transactions. This language is by far the most similar to $\mathcal{TR}$. For instance, $\mathcal{TR}$ and Dynamic Prolog are the only logic programming languages that account not just for updates, but for transaction abort and rollback as well. However, 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 a transaction program, the proof theory *verifies* whether the program causes the transition; but, given just the initial state and the program, it cannot *compute* the final state. In other words, the proof theory cannot execute transaction programs as $\mathcal{TR}$'s proof theory does. Apparently realizing this drawback, Manchanda and Warren developed an interpreter whose aim was to "execute" transactions. However, this interpreter is incomplete with respect to the model theory and, furthermore, it is not based on the proof theory. To a certain extent, it can be said that Manchanda and Warren have managed to formalize their intuition as a program, but not as an inference system. In addition, there are several other differences between Dynamic Prolog and $\mathcal{TR}$. For instance, Dynamic Prolog assumes that databases are relational, and it does not support bulk database updates or constraints on program execution. Furthermore, an execution path in Dynamic Prolog consists of the initial and the final state only, and it does not record the intermediate states. As a consequence, it is impossible to express constraints on transaction execution, such as those needed for advanced applications in AI, workflow management, etc. [BK95; DKRR97].

LDL. Naqvi and Krishnamurthy [NK88] extended Datalog with update operators, which were later incorporated in the LDL language [NT89]. Since LDL is geared towards database applications, this extension has bulk updates, for which an operational semantics exists. Unfortunately, the model theory presented in [NK88; NT89] is somewhat limited. First, it matches the execution model of LDL only in the propositional case, and so it does not cover bulk updates. Second, it is only defined for update-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 [NK88; NT89] is highly restrictive, making it difficult to build complex transactions out of simpler ones.

Chen's Calculus. Chen developed a calculus and an equivalent algebra for constructing transactions [Che91]. Like $\mathcal{TR}$, this calculus uses logical operators to construct database transactions from elementary updates. There are several differences however. First, the calculus is not part of a full logic. Second, it assumes that databases are relational. Third, it has a very different semantics for conjunction. Specifically, whereas $\mathcal{TR}$ uses $\wedge$ to express dynamic 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. First, the calculus cannot express the kind of dynamic constraints that $\mathcal{TR}$ can [BK95], while $\mathcal{TR}$ expresses bulk updates through other means [BKC94; BK95]. Second, parallel actions greatly complicate the semantics, since they require a minimality principle, which makes the algebra non-monotonic even in the absence of negation. Third, the syntax is not closed. For instance, negation can be applied to some formulas but not to others. In particular, if ψ is an updating transaction, then rules like $p \leftarrow \psi$ are not allowed, since it is equivalent to $p \vee \neg\psi$; indeed, this formula has no meaning in Chen's calculus. It therefore seems unlikely that this calculus can be developed into a full *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.

Abiteboul-Vianu's Update Languages. Abiteboul and Vianu developed a family of Datalog-style update languages [AV91; Abi88], including comprehensive results on complexity and expressibility. Unlike Transaction Logic, these languages are not part of a full *logic*: arbitrary logical formulas cannot be constructed, and although there is an operational semantics, there is no corresponding model theory and no logical inference system. In addition, these languages lack several features that are present in $\mathcal{TR}$. First, they assume that databases are relational. Second, they do not support subtransactions, savepoints, or partial abort and rollback. Third, there is no facility for constraining program execution, and program output is the only concern. Fourth, there is no support for subroutines. This can be seen most clearly in the procedural languages defined in [AV90]. This lack of subroutines is reflected in the PSPACE data complexity of some of the languages, since subroutines would lead to *alternating* PSPACE, that is, EXPTIME [Bon].

1.7.2 Logics for Reasoning about Programs

Dynamic Logic and Process Logic. Dynamic Logic [Har79] and Process Logic [HKP82] allow a user to express properties of procedural programs and to reason about them.[13] Dynamic Logic reasons about the initial and final states of program execution. Thus, one can speak about the *result* of an execution; *e.g.*, "When the program terminates, the value of X is less than 100." Process Logic extends this with the ability to reason about intermediate states. Thus, one can speak about what happens *during* execution; *e.g.*, "At every iteration of the loop, the value of X is less than 100." In both Process Logic and Dynamic Logic, as in $\mathcal{TR}$, a model consists of a set of states, and actions cause transitions from one state to another. Because of the emphasis on intermediate states, the semantics of $\mathcal{TR}$ is more closely related to that of Process Logic than of Dynamic Logic. For instance, in both Process Logic and $\mathcal{TR}$, formulas are evaluated *not* at states, but on *paths*, which are sequences of states.

Unlike $\mathcal{TR}$, however, Process Logic and Dynamic Logic are not logic programming languages. This difference shows up in several ways. First, Process Logic and Dynamic Logic represent programs procedurally, not as sets of logical rules. Second, they do not have an SLD-style proof procedure for executing programs. In fact, they were not intended for executing programs, but for reasoning about their properties. Third, in both Process Logic and Dynamic Logic, the logic itself is used *outside* of programs to specify their properties. In particular, the logic is used for *constructing* the programs themselves or for specifying database queries. Fourth, Process Logic and Dynamic Logic were not designed for database programming. For instance, they do not have the notions of a database or a query, and they do not support named procedures, such as subroutines and views.

McCarty and Van der Meyden. In [MvdM92], McCarty and Van der Meyden develop a theory for reasoning about "indefinite" actions. This work is orthogonal to $\mathcal{TR}$. The main similarity is that both works are concerned with defining complex actions in terms of simpler ones, and in both works, the actions may be non-deterministic (or "indefinite"). However, unlike $\mathcal{TR}$, [MvdM92] does not address action execution or the updating of databases. To give an idea of what [MvdM92] is about, consider a $\mathcal{TR}$ transaction base consisting of exactly the following two rules:[14]

$$a \leftarrow c1 \otimes c2 \otimes c3 \qquad\qquad b \leftarrow c2 \otimes c3$$

The main point is that a and b are complex actions defined in terms of the elementary actions $c1, c2, c3$. In $\mathcal{TR}$, the effects of the elementary actions are specified by an oracle, which is invoked to execute them. In contrast, [MvdM92] has no mechanism for specifying the effects of elementary actions. Instead, their work focuses on closed-world inferences of the following form:

> *If we are told that action a has occurred, then we infer, abductively, that action $c1 \otimes c2 \otimes c3$ has occurred, so action $c2 \otimes c3$ has occurred, so action b has occurred. Thus, an occurrence of action a implies an occurrence of action b.*

There are also technical differences between $\mathcal{TR}$ and [MvdM92]. For instance, [MvdM92] does not allow function symbols in the rules that define complex actions (and even so, most of their reasoning problems are undecidable and/or outside of re). In addition, [MvdM92] is committed to a particular model of states. The basic theory in [MvdM92] is also very different from that of $\mathcal{TR}$, as it is based on circumscription and second-order intuitionistic logic.

In earlier work, McCarty outlined a logic of action as part of a larger proposal for reasoning about deontic concepts [McC83]. 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 in deontic concepts, McCarty defines two notions of satisfaction. In one notion, called "strict satisfaction," the conjunction $\wedge$ corresponds to parallel action, as it does in Chen's work [Che91]. 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 was on strict satisfaction, the development of dynamic constraints was never considered. Also, there is no analogue of $\mathcal{TR}$'s transition oracle, and the only elementary updates considered correspond to insertion and deletion of atomic formulas. Unfortunately, this promising proposal 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.

Situation Calculus. The situation calculus is a methodology for specifying the effects of elementary actions in first-order classical logic. It was introduced by McCarthy [McC63] and then further developed by McCarthy and Hayes [MH69]. From a database perspective, transactions specified in the situation calculus can insert and delete atomic formulas, but not arbitrary logical formulas. Thus, such transactions cannot add new rules to a deductive database, nor can they add tuples with null values to a relational database, or disjunctions to a disjunctive database. None of these limitations apply to transactions in $\mathcal{TR}$.

As a representation language, the situation calculus is strictly less powerful than full $\mathcal{TR}$. Like $\mathcal{TR}$, the situation calculus can axiomatize the effects of elementary actions and reason about them; but unlike $\mathcal{TR}$, its ability to combine actions is very limited. For instance, it does not support loops, conditionals, subroutines, recursion or non-deterministic choice. Formally, the situation calculus is a subset of first-order classical logic, which is a subset of $\mathcal{TR}$. The situation calculus is therefore subsumed by *full* $\mathcal{TR}$. On the other hand, being a methodology within the classical logic rather than an independent logical system, the situation calculus is orthogonal to *Horn* $\mathcal{TR}$. This is because, Horn $\mathcal{TR}$ emphasizes the combination of elementary actions into complex ones, but not the specification of elementary actions themselves. The situation calculus does the reverse: it emphasizes the specification of elementary actions, but not their combination into complex actions. These two formalisms can therefore work hand-in-glove. Specifically, through its oracle mechanism, Horn $\mathcal{TR}$ can combine elementary actions specified in the situation calculus.

Another difference, as mentioned earlier, is that the frame problem does not arise in Horn $\mathcal{TR}$, since it is a language for programming and executing transactions, not for reasoning about them. In fact, the frame problem has not been an issue in any of the theory or any of the examples presented in this paper. In contrast, the frame problem is a central issue in the situation calculus [MH69; Rei91]. The frame problem is also an issue in *full* $\mathcal{TR}$ when it is used to reason about the properties of actions, a context in which frame axioms are unavoidable [BK].

Reiter's Theory of Database Evolution. Although the "main stream" of AI was treating the situation calculus as a mere curiosity for almost 30 years, it has recently received renewed development by Reiter. In particular, Reiter has developed an approach to the frame problem that does not suffer from the usual blow-up in the number of frame axioms [Rei91]. Also, unlike the original situation calculus, which was entirely first-order [McC63; MH69], Reiter's development includes an induction axiom specified in second-order logic, for reasoning about action sequences [Rei93]. Applying this approach, Reiter has developed a logical theory of database evolution [Rei95]. This theory is perfectly compatible with $\mathcal{TR}$. Through its oracle mechanism, $\mathcal{TR}$ can use the theory to specify the semantics of database states and elementary actions. Horn $\mathcal{TR}$ can then combine these actions into complex programs, and full $\mathcal{TR}$ can reason about them.

However, from the perspective of database theory [AHV95; Ull88], Reiter's theory of database evolution is quite unusual. For instance, a database state is usually modeled as set of relations or logical formulas; but in Reiter's theory, a state is identified with a sequence of actions. Thus, different transactions always terminate at different states, even if they have the same effect on the database. For example, the state resulting from the action "*insert a, then insert b*" is formally different from the state resulting from "*insert b, then insert a*."

In addition, the theory adopts the view that databases are never actually updated and transactions are never executed. Instead, the initial database state is preserved forever, and the history of database transactions is recorded in a kind of log. Thus, the current database state is not materialized, but is *virtual*. In this framework, queries to the current state are answered by querying the log and reasoning backwards through it to the initial state [Rei95]. Unfortunately, this means that simple operations, like retrieving a single tuple from the database, become long and complicated reasoning processes. Since database logs are typically large (perhaps millions of transaction records long), reasoning backwards through them is unacceptably expensive. Recognizing this problem, Reiter and his colleagues have looked at ways of materializing the current database state [Rei95; LR94; LLL$^+$94]. However, no theory has been presented showing how the materialization can be carried out within a logical framework.

Finally, Reiter's theory does not apply to logic programs and deductive databases. There are two reasons for this. First, the theory does not provide a minimal model semantics for database states. Thus, in Reiter's theory, databases do not have the semantics of logic programs. Instead, the theory requires databases to have a purely first-order classical semantics. Unfortunately, this means that much of the familiar database and logic programming methodology does not apply. For instance, although transitive closure is trivial to express in a deductive database, it cannot be expressed by the databases of Reiter's theory, since transitive closure is not first-order definable [AU79]. The lack of a minimal-model semantics also complicates the representation of relational databases. Instead of representing them as sets of ground atomic formulas in the usual way, the theory uses Clark's completion [Llo87; Rei84], which, in the case of databases, requires very large first-order formulas. In AI terminology, these complications arise because Reiter's theory is about *open worlds*, whereas databases are *closed worlds*. Unfortunately, updating open worlds is an intractable problem in general, since the result of an update may not have a finite representation in first-order logic [LR94].

Second, the theory does not protect deductive rules from database updates. In particular, updates can damage and destroy rules. For example, suppose that a deductive database consists of the single rule $p(X) \leftarrow q(X)$, and suppose that the atom $q(b)$ is inserted into this database. If this update is formalized in Reiter's theory, then the updated database would be equivalent to the following two formulas:[15]

$$q(b) \qquad\qquad p(X) \leftarrow q(X) \wedge X \neq b$$

The point here is that the rule has changed as a result of inserting $q(b)$. This change is a direct result of Reiter's approach to the frame problem [Rei91], which intuitively says that except for atoms that are explicitly inserted or deleted, all atoms must retain their old truth values. In this case, since the atom $p(b)$ was not true in the initial database, it must not be true in the final database; so the rule premise must be modified to ensure that $X \neq b$. Of course, this dictum is completely contrary to the idea of database views, in which virtual data depends on base data and can change as an indirect effect of database updates. In AI terminology, this is an example of the *ramification problem* [Fin86; Rei95].

To account for views, Reiter treats view definitions as integrity constraints that must be maintained by the transaction system. In this approach, views are not defined by Horn rules. Instead, the axioms of the transaction system are modified to treat views as stored data. For instance, in the above example, whenever a transaction inserts (or deletes) the atom $q(b)$ from the database, the modified axioms would insert (or delete) the atom $p(b)$ as well [Rei95]. In this way, the system behaves *as if* the database contained the deductive rule $p(X) \leftarrow q(X)$ (with a minimal model semantics). Unfortunately, in this approach, view definitions depend on transaction definitions. Thus, each time a transaction is modified or defined, the change must be propagated to all the view definitions. In addition, the approach requires that all views be defined directly in terms of base predicates. Thus, views cannot be recursive, and views cannot be defined in terms of other views.

In sum, the notion of database state in Reiter's theory does not allow for the fundamental features of deductive databases and logic programs, namely recursion, view composition, and minimal-model semantics. Consequently, the theory does not provide a logical account of how to query or update such databases.

Golog. Levesque et al have recently developed Golog, a procedural language for programming complex actions, including database transactions [LRL+97]. Syntactically, Golog is similar to the procedural database language QL developed by Chandra and Harel [CH80] extended with subroutines and non-deterministic choice. Semantically, however, Golog is much more complex, since the meaning of elementary actions is specified in the situation calculus, and the meaning of larger programs is specified by formulas of second-order logic. Because of this logical semantics, it is possible to express properties of Golog programs and to reason about them (to some extent).

Unfortunately, despite the claims of its developers, Golog is *not* a logic programming language. This is because having a logical semantics is not the same thing as programming in logic. Certainly, formalizing the semantics of Fortran in logic would not make Fortran a logic programming language (although it would make it possible to reason about Fortran programs). In fact, in many ways, Golog is the *opposite* of logic programming. Most obviously Golog programs are not defined by sets of Horn-like rules, but by procedural statements in an Algol-like language. Golog also does not come with an SLD-style proof procedure that executes programs and updates databases as it proves theorems. Finally, unlike Horn $\mathcal{TR}$, Golog does not include classical logic programming as a special case. That is, classical logic programs and deductive databases are *not* Golog programs, while they *are* $\mathcal{TR}$ programs.

In addition, Golog programs cannot be combined with classical logic programs, and they cannot query or update deductive databases. This is because Golog is based on Reiter's theory of database evolution, which, as described above, does not apply to logic programs and deductive databases. Even if the initial database state is described by classical Horn rules, Golog does not treat these rules as a logic program. For instance, suppose the initial state is described by the following two rules:

$$tr(X,Y) \leftarrow r(X,Y) \qquad\qquad tr(X,Z) \leftarrow r(X,Y) \wedge tr(Y,Z)$$

In a deductive database, these rules specify the transitive closure of relation r, but in Golog they do not. This is because transitive closure requires the minimal model semantics of deductive databases, which Golog lacks. In addition, Golog does not protect these rules from database updates; so, as described above, the rules are progressively damaged and destroyed as relation r is updated. Transitive closure *can* be defined in Golog, but *not* by deductive rules. Instead, the user must write an Algol-like procedure, as illustrated in [LRL+97]. In this way, Golog sacrifices the declarativeness of deductive databases for the procedurality of Algol. For the same reason, Golog has difficulty in specifying database views, especially recursive views [Rei95]. These difficulties all arise because Golog abandons the logic-programming paradigm.

Golog has numerous other differences with $\mathcal{TR}$ as well. For instance, Golog subroutines are not logical entities, but are macros specified *outside* the logic. Thus, one cannot refer to them in the logic, and in particular, one cannot quantify over them or reason about them [LRL+97]. In addition, like other logics of action, updates in Golog are hypothetical, not real. This is because Golog uses the situation calculus to reason about what *would* be true *if* an action took place. The actual execution of actions requires a separate run-time system, outside of Golog. Finally, because it is based on Reiter's theory of database evolution, there are many kinds of states that Golog cannot represent, including Prolog programs with negation-as-failure. Likewise, there are many kinds of updates that Golog cannot represent, including the insertion of rules into deductive databases, and the insertion of disjunctions into disjunctive databases.

Datalog with State. A number of researchers have worked on adding a notion of state to Datalog programs [Zan93; LHL95]. In these works, states are represented through a special, distinct argument, which is added to each updatable predicate. Updates are then modeled as state transitions. This approach can be viewed as an adaptation of the situation calculus to Datalog. As such, it has several important differences with Reiter's theory of database evolution and with Golog. First, unlike Reiter's theory, Datalog with state uses a form of closed-world semantics (XY-stratification [Zan93] or state-stratification [LHL95]), which is closer to the database tradition. Moreover, unlike Reiter's theory, Datalog with state has no problem in representing database views, recursive or otherwise. Second, actions in Datalog with state are limited to the insertion and deletion of ground atomic formulas. Because of this restriction, the frame problem is not a serious issue, since it only needs to be axiomatized for a small, fixed number of actions. Third, unlike Golog (and $\mathcal{TR}$), Datalog with state has no notion of subroutine. That is, transaction programs cannot be named and used within other programs. This approach to database updates is further discussed in Chapter ??.

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 and his approach to the issue. Mariano Consens has survived the very first draft of $\mathcal{TR}$, and many improvements are due to his sufferings. Discussions with Gösta Grahne and Peter Revesz were also helpful for the progress of this work. We would also like to thank the anonymous referees for their valuable comments.

The first author was supported in part by a Research Grant from the Natural Sciences and Engineering Research Council of Canada and by a Connaught Grant from the University of Toronto. The second author was supported in part by NSF grant IRI9404629. Support from the Computer Systems Research Institute of University of Toronto is also gratefully acknowledged.

Notes

1. As opposed to using the Event Calculus to simulate updates [Kow92].

2. Additional information on $\mathcal{TR}$, including a prototype implementation, a tutorial, and benchmark tests, is available at `www.cs.toronto.edu/~bonner/transaction-logic.html`

3. All our examples assume the *non-strict* version of *insert* and *delete*, which means that executing $p.ins$ at a state where p is true does not change the state (and likewise when $p.del$ is executed at a state where p is false).

4. Assuming that only one block can be on top of another block.

5. Assuming *bought* and *wanted* are not in $\mathbf{D}$.

6. Assuming *found* and *won* are not in $\mathbf{D}$.

7. Assuming *found* and *won* are not in $\mathbf{D}$.

8. We could allow arbitrary closed formulas like $a \vee b$ to be returned by the transition oracle, but this would complicate the theoretical development, and it is not clear what applications this generalization would have.

9. However, preventing the user from tinkering with the definitions of elementary transitions may be a good policy.

10. The discrete Fourier transform and numerous other numerical operations are typically provided as built-in operations by scientific software packages.

11. Generalized-Horn are rules with possibly negated premises.

12. For example, suppose a view is defined by the rule $p \leftarrow q$, and suppose the current database state contains the literal $\neg p$. Then, the insertion of q into the database would cause inconsistency. In most logics of action, the mere possibility of this happening would cause global inconsistency, thus rendering the entire logical system useless, even if all other database states were consistent. The use of $\top$ prevents this kind of "global collapse" in $\mathcal{TR}$ by isolating each such inconsistency to the state that causes it. Note that in the case of Horn databases, which is the main focus of this paper, inconsistency is not possible, so $\top$ does not make any difference.

13. A number of different process logics have been proposed in the literature, beginning with Pratt's original work [Pra79]. The version in [HKP82] is closer to $\mathcal{TR}$ than any other incarnation of Process Logic we are aware of.

14. Here we use the syntax of $\mathcal{TR}$, which can be translated into the original syntax of [MvdM92].

15. In both the initial and final database, we have suppressed the so-called "situation argument." Situation arguments identify a database state in the situation calculus, but are unnecessary for describing the formulas that are true in a state.

References

[Abi88] S. Abiteboul. Updates, a new frontier. In *Intl. Conference on Database Theory*, pages 1–18, 1988.

[ABW88] K.R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[All84] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, July 1984.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[Ban86] F. Bancilhon. A logic-programming/Object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, September 1986.

[Bee92] C. Beeri. New data models and languages—The challenge. In *ACM Symposium on Principles of Database Systems*, pages 1–15, New York, June 1992. ACM.

[BK] A.J. Bonner and M. Kifer. Reasoning about action in transaction logic. In preparation. Presented at the *Dagstuhl Seminar on Logic Databases and the Meaning of Change*, September 23–27 1996, International Conference and Research Center for Computer Science, Schloss Dagstuhl, Wadern, Germany.

[BK94] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[BK95] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.

[BK96] A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.

[BKC94] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[Bon] A.J. Bonner. The power of cooperating transactions. Submitted for publication.

[Bon97a] A.J. Bonner. Intuitionistic deductive databases and the polynomial time hierarchy. *Journal of Logic Programming*, 33(1):1–47, October 1997.

[Bon97b] A.J. Bonner. A logical semantics for hypothetical rulebases with deletion. *Journal of Logic Programming*, 32(2):119–170, August 1997.

[Bon97c] A.J. Bonner. Transaction Datalog: a compositional language for transaction programming. In *Proceedings of the International Workshop on Database Programming Languages*, Estes Park, Colorado, August 1997. Springer Verlag. Long version available at `http://www.cs.toronto.edu/~bonner/papers.html#transaction-logic`.

[BSR96] A. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: a database benchmark for high-throughput workflow management. In *Intl. Conference on Extending Database Technology*, number 1057 in Lecture Notes in Computer Science, pages 463–478, Avignon, France, March 25–29 1996. Springer-Verlag.

[CH80] A.K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[Che91] W. Chen. Declarative specification and evaluation of database updates. In *Intl. Conference on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, December 1991.

[DKRR97] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modelling and analysis of workflows. in preparation, October 1997.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Fin86] J. Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1986.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[GM95] G. Grahne and A.O. Mendelzon. Updates and subjunctive queries. *Information and Computation*, 116(2):241–252, February 1995.

[GRS94] Nathan Goodman, Steve Rozen, and Lincoln Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, pages 259–278. Kluwer, 1994. `ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps`.

[Har79] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[HKP82] D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.

[Kif95] M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Intl. Conference on Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Intl. Conference on Deductive and Object-Oriented databases.

[KL92] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, pages 741–843, July 1995.

[KM92] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In P. Gardenfors, editor, *Belief Revision*, volume 29 of *Cambridge Tracts in Theoretical Computer Science*, pages 183–203. Cambridge University Press, 1992.

[Kow92] R.A. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proceedings of the 7th Intl. Conference on Management of Data*, Pune, India, December 1995. Tata McGrow-Hill.

[LLL+94] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, and R. Reiter. A logical approach to high-level robot programming–a progress report. In *Control of the Physical World by Intelligent Systems, Working Notes of the 1994 AAAI Fall Symposium.* AAAI Press, New Orleans, LA, November 1994.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming (Second Edition).* Springer-Verlag, 1987.

[LR94] F. Lin and R. Reiter. How to progress a database (and why) I. Logical foundations. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 425–436, 1994.

[LRL+97] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 1997. To appear.

[MBDH83] J.N. Maksym, A.J. Bonner, C.A. Dent, and G.L. Hemphill. Machine Analysis of Acoustical Signals. *Pattern Recognition*, 16(6):615–625, 1983. Also appears in *Proceedings of the Workshop on Issues in Acoustic Signal/Image Processing and Recognition*, San Miniato, Italy, August 5–9 1982.

[McC63] J. McCarthy. Situations, actions, and clausal laws, memo 2. Stanford Artificial Intelligence Project, 1963.

[McC83] L.T. McCarty. Permissions and obligations. In *Intl. Joint Conference on Artificial Intelligence*, pages 287–294, San Francisco, CA, 1983. Morgan Kaufmann.

[MH69] 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.

[MvdM92] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 59–70, Cambridge, MA, October 1992.

[MW88] 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.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM Symposium on Principles of Database Systems*, pages 251–262, New York, March 1988. ACM.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, 1989.

[PDR91] 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.

[Pra79] V.R. Pratt. Process logic. In *ACM Symposium on Principles of Programming Languages*, pages 93–100, January 1979.

[Prz88] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 191–233. Springer-Verlag, 1984.

[Rei91] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Aritifial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarty*, pages 359–380. Academic Press, 1991.

[Rei93] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, December 1993.

[Rei95] R. Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1):53–91, October 1995.

[Ull88] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, Rockville, MD, 1988.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Win88] M. Winslett. A model based approach to updating databases with incomplete information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.

[Zan93] Carlo Zaniolo. A unified semantics for active and deductive databases. In *Proceedings of the Workshop on Rules in Database Systems*, Workshops in Computing. Springer-Verlag, Edinburgh, U.K., 1993.