The State of Change: A Survey

Anthony J. Bonner[1] and Michael Kifer[2]

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

[2] Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, U.S.A., kifer@cs.sunysb.edu

Abstract. 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 using ad hoc operators without a logical semantics. A great many works have been dedicated to developing logical theories in which the state of the underlying database can evolve with time. Many of these theories were developed with specific applications in mind, such as reasoning about actions, database transactions, program verification, etc. As a result, the different approaches have different strengths and weaknesses. In this survey, we review a number of these works, discuss their application domains, and highlight their strong and weak points.

1 Introduction

Logic-based approaches to updates can be broadly classified into two categories: those designed for database or logic programming and those designed for reasoning about programs and actions. Prolog [CM81] is an early (and the best-known) example of the languages in the first category. The situation calculus [McC63], Dynamic Logic [Har79], and Temporal Logic [Pnu77] are, in our view, the forefathers of the modern approaches to reasoning about actions. As we shall see, both the situation calculus and Dynamic Logic influenced the design of several logic-based languages for programming database transactions. Temporal Logic had a much lesser influence on the languages for database transactions, and we shall not discuss it here.[1]

In the next section, we briefly review these two approaches, and in subsequent sections, we discuss other, more-recent proposals for dealing with state changes in logic-based systems.

[1] Although the concept of time plays an important role in databases, and especially in temporal databases, these areas have adapted little from the vast knowledge that exists on Temporal Logic.

This survey is written from the point of view of logic programming and databases, with an emphasis on languages that have a logical semantics. This means that we shall focus on languages in which database queries and updates can be expressed declaratively and can be executed efficiently. It also means that we shall emphasize rule-based languages, since there is a tradition of rules in databases, and they are central to logic programming.

To make the comparison of the various approaches more concrete, we have chosen Transaction Logic [BK98a,BK96,BK95,BK94,BK93] as a standard with which other approaches can be easily compared. We find this convenient because Transaction Logic is both a full-blown logic (with a model theory and a proof theory) and, at the same time, it contains a logic programming language as a subset. We shall see, for instance, that while many rule-based languages use a logical syntax, they do not always have a logical semantics, and sometimes both their syntax and semantics are burdened by a host of ad hoc restrictions.

A pure logical semantics has two main benefits. First, it facilitates understanding. In particular, the logical semantics of a rule base is independent of the particular algorithms used to evaluate the rules. It therefore eliminates the need to understand the intricacies of such algorithms. In fact, many different algorithms can be used as long as they are consistent with the logical semantics. The best examples of this are classical logic programs and deductive databases, which can be evaluated in either a top-down or a bottom-up manner. The second benefit of a logical semantics is that it becomes possible to express properties of programs and to reason about them. For instance, one might be able to infer that a particular database program preserves the integrity constraints of a database; or that under certain conditions, a transaction program is guaranteed not to abort.

2 Ancient History

2.1 Prolog

Database transactions can be defined in Prolog via the operators *assert* and *retract*. When $retract(p)$ is executed, p is removed from the database; and when $assert(p)$ is executed, p is inserted into the database. The two examples below illustrate these operators. They also illustrate two different concerns, one associated with transaction processing, and the other with Artificial Intelligence. These very different concerns have affected the design and evolution of many of the languages surveyed in this paper. All examples in this paper use the Prolog convention that logical variables begin in upper case.

Example 1. **(Financial Transactions)** The rules below form a Prolog program (with some simplifying syntactic liberties) that defines simple banking transactions for depositing and withdrawing funds. In these rules, the balance of a bank account is represented by the predicate $balance(Acct, Amt)$. The rules define four database transactions: $change(Acct, Bal_1, Bal_2)$, which changes the balance of account $Acct$ from Bal_1 to Bal_2; $withdraw(Amt, Acct)$, which withdraws an amount from an account; $deposit(Amt, Acct)$, which deposits an amount into an account; and $transfer(Amt, Acct_1, Acct_2)$, which transfers an amount from account $Acct_1$ to account $Acct_2$.

$$transfer(Amt, Act, Act') \ :- \ withdraw(Amt, Act)\,,\ deposit(Amt, Act').$$

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

$$deposit(Amt, Act) \ :- \ balance(Act, B)\,,\ change\_balance(Act, B, B + Amt).$$

$$\begin{aligned} change\_balance(Act, B, B') \ :- \ & retract(balance(Act, B))\,, \\ & assert(balance(Act, B')). \end{aligned}$$

In addition to executing database transactions, Prolog programs can simulate actions that take place in the real world. The example below is taken from the *blocks world* of Artificial Intelligence, in which a robot arm manipulates toy blocks sitting on a table top.

Example 2. **(Robot Actions)** The following rules simulate the motion of a robot arm. In these rules, the predicate $on(X, Y)$ means that block X is sitting on block Y, and $clear(X)$ means that nothing is on block X. The rules define three robot actions: $move(X, Y)$, which means "move block X on top of block Y;" $pickup(X)$, which means "pick up block X;" and $putdown(X, Y)$, which means "put down block X on top of block Y."

$$move(X, Y) \ :- \ pickup(X)\,,\ putdown(Y).$$

$$pickup(X) \ :- \ clear(X)\,,\ on(X, Z)\,,\ retract(on(X, Z)),\ assert(clear(Z)).$$

$$putdown(X, Y) \ :- \ clear(Y)\,,\ assert(on(X, Y))\,,\ retract(clear(Y)).$$

The distinctive features of Prolog as a language for specifying updates are:

1. It is a programming language based on a Horn subset of a full logic.
2. Programs are executed by an SLD-style proof procedure. This is quite unlike most of the approaches designed for reasoning about actions, where programs are executed outside the logic by a separate run-time system.
3. Prolog updates are real, not hypothetical. In contrast, many other approaches do not update the database in the real sense. Instead, they reason about what *would* be true *if* certain actions were carried out.[2]
4. In the absence of updates (and some other non-logical control features), Prolog reduces to classical Horn logic.
5. The frame problem (described in Section 2.3) is not an issue, *i.e.*, Prolog programmers do not need to write down (or even be aware of) frame axioms.

[2] This can be seen in the examples of the situation calculus in Section 2.3.

Unfortunately, the *assert* and *retract* operators in Prolog are not covered by an adequate logical theory, so each time a programmer uses them, he moves further away from declarative programming.

In addition, Prolog does not support a basic feature of database transactions: *atomicity*—the appearance that a transaction either executes to completion or not at all. Atomicity allows a complex program to be treated as an atomic or elementary operation. In database systems, atomicity is enforced by rolling back the database to its initial state whenever a transaction fails. Because Prolog lacks this fundamental feature, the semantics of update programs is heavily dependent on rule order. It is therefore difficult to provide a logical semantics for them, and is difficult to reason about them. Moreover, they are often the most awkward of Prolog programs, and the most difficult to write correctly and to understand.

To illustrate, consider the following Prolog query:

$$:-\ transfer(Client, Seller, Price),\ \ transfer(Client, Broker, Commissions).$$

On the surface, the above query seems like a rather straightforward way of saying that a buyer must pay broker commissions to complete a transaction. However, suppose that the client's account has sufficient funds to cover the purchase price, but not to cover both the price and the commissions. In such a case, a database management system would abort the transaction and undo the changes done by the first subgoal, $transfer(Client, Seller, Price)$, after the second subgoal failed, thereby guaranteeing atomicity. Unfortunately, Prolog does not support this feature, since the changes done by *assert* and *retract* are not undone during backtracking.

Prolog's lack of atomicity can also be seen in Example 2. In particular, $move(X, Y)$ cannot be treated as an atomic action. To see this, suppose that the database consists of the atoms $\{clear(b),\ on(b, c),\ on(a, d)\}$. Then the action $pickup(b)$ can succeed, but the action $putdown(b, d)$ cannot. Consequently, if $move(b, d)$ is executed, it will fail; but the database will *not* be left in its original state. Instead, it is left in the state $\{clear(b),\ clear(c),\ on(a, d)\}$. This is because the action $pickup(b)$ deleted the atom $on(b, c)$ from the database, and inserted the atom $clear(c)$. The task of restoring the database to a meaningful state is left to the Prolog programmer.

The lack of atomicity in Prolog is not just a question of adjusting the operational semantics. For even if Prolog did support atomicity, a logical semantics would still be needed to account for it, and for updates. (See Section 3.1 for just such a semantics.)

Finally, we note that updates in Prolog are not integrated into the host logical system (*i.e.*, classical logic). It is therefore not clear how *assert* and *retract* should interact with other logical operators such as disjunction and negation. For instance, what does $assert(X) \vee assert(Y)$ mean? or $\neg assert(X)$? 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.

2.2 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.[3] Dynamic Logic reasons about the initial and the final states of program execution. For instance, one can speak about the *result* of an execution; *e.g.*, "*Variable X assumes value 0 when the program terminates.*" Process Logic extends this with the ability to reason about intermediate states. Thus, one can speak about what happens *during* execution; *e.g.*, "*Variable X assumes value 0 at some point during the computation.*" The basic syntactic blocks of these logics have the following form: $[program]\phi$. Here, ϕ is a logical formula, and *program* is an expression in a language of a completely different nature: it is procedural rather than logical, and contains the operators of sequential composition, iteration, conditionals, etc. Thus, programs are used as modal operators, while logical formulas are used as assertions.

In both Process Logic and Dynamic Logic, a model consists of a set of states, and actions cause transitions from one state to another. In Dynamic Logic, a formula $[program]\phi$ is true at a state, s_1, if ϕ is true at all states s_2 such that an execution of *program* transforms s_1 into s_2. More than one state s_2 is possible since programs in Dynamic Logic may be non-deterministic. In contrast, in Process Logic, formulas are true on *paths* rather than on states. A path is a sequence of states, which is supposed to represent all the intermediate state changes during program execution. Intuitively, a path, $s_1, s_2, ..., s_n$, is an execution path of a program if the program can start executing at state s_1, change it to s_2, then to s_3, ..., to s_n, and terminate at s_n. The path semantics of a first-order formula is simple: it is true on a path if it is true at the first state of the path. The semantics of the modal formula $[program]\phi$ is more complex. It is true on a path, $s_1, ..., s_i$, if ϕ is true on *every* path of the form $s_1, ..., s_i, ..., s_n$, where $s_i, ..., s_n$ is an execution path of *program*. As a special case, $[program]\phi$ is true on the singleton path s_1 if ϕ is true on every execution path of *program* beginning with state s_1. Although there are many differences, this emphasis on paths is similar to Transaction Logic (Section 3.1).

In a database setting, program statements in Dynamic and Process logics could be relational algebra operators, tests, and control operators, like *while*-loops. For instance, consider the following formula in Dynamic Logic:

$$\begin{array}{l}
[\,temp := connection; \\
\;flight := \emptyset; \\
\;\textbf{while}\ flight \subset temp\ \textbf{do} \\
\qquad flight := temp; \\
\qquad temp := temp \cup (temp \bowtie connection); \\
\;\textbf{od}\,] \\
\forall X, Y, Z\ flight(X, Y) \wedge flight(Y, Z) \rightarrow flight(X, Z)
\end{array} \tag{1}$$

[3] A number of different process logics have been proposed in the literature, beginning with Pratt's original work [Pra79]. The version in [HKP82] is most interesting for the purpose of this survey.

The bracketed part of the formula is a program—a naive procedure that is supposed to find the transitive closure of the *connection* relation. The last line in the formula is a first-order statement that asserts that the *flight* relation is transitively closed. As a statement in Dynamic Logic, (1) asserts that after executing the program inside the brackets, the *flight* relation is transitively closed.

Dynamic Logic can be used to reason about the *outcome* of a computation, but it is hard, if not impossible, to make assertions about what happens *during* the computation. For instance, suppose we would like to assert (or verify) that during a computation of the above program, if $flight(a, b)$ and $flight(b, c)$ become true, then $flight(a, c)$ will eventually become true as well. Note that we are not claiming that $flight(a, c)$ will be true at the *end* of the computation; so this property is not implied by the above formula in Dynamic Logic.

To reason about what happens during a computation, Process Logic [HKP82] has a path-based semantics, where a path is a sequence of states, representing a program computation. Process Logic includes several non-classical connectives for making assertions about programs and computations. For instance, the formula $\phi\, \mathbf{suf}\, \psi$ is true on a path if ψ is true on some proper suffix of the path and ϕ is true on every larger proper suffix. In the special case in which ϕ and ψ are first-order formulas, **suf** corresponds to the *until* operator of temporal logic. That is, $\phi\, \mathbf{suf}\, \psi$ is true on a path if ψ is true on some state of the path (other than the initial state), and ϕ is true on all preceding states (other than the initial state).

In Process Logic, the symbol $\mathbf{1}$ is true on every path. Thus, $\mathbf{1}\, \mathbf{suf}\, \phi$ is true on a path if ϕ is true on some proper suffix. The formula $\phi \vee (\mathbf{1}\, \mathbf{suf}\, \phi)$ is true if ϕ is true on some suffix (not necessarily a proper one). This latter formula is abbreviated as $\mathbf{some}\, \phi$, and it intuitively means that ϕ eventually becomes true. Likewise, the formula $\mathbf{some}(\phi \wedge \mathbf{some}\, \psi)$ intuitively means that ϕ eventually becomes true, and ψ becomes true some time later. Likewise, the formula $\mathbf{some}(\phi \rightarrow \mathbf{some}\, \psi)$ means that if ϕ eventually becomes true, then ψ will become true some time later.

We can use these formulas to express properties of program execution. For instance, the formula $[program]\, \mathbf{some}\, \phi$ intuitively means that during any execution of *program*, formula ϕ eventually becomes true. Now, consider the following formula:

$$
\begin{array}{l}
[\, temp := connection; \\
\;\; flight := \emptyset; \\
\;\; \mathbf{while}\ flight \subset temp\ \mathbf{do} \\
\qquad flight := temp; \\
\qquad temp := temp \cup (temp \bowtie connection); \\
\;\; \mathbf{od}\,] \\
\forall X, Y, Z\ \mathbf{some}(flight(X, Y) \wedge flight(Y, Z)\ \rightarrow\ \mathbf{some}\ flight(X, Z))
\end{array}
\tag{2}
$$

The bracketed part of this formula is the transitive-closure program given earlier in (1). Now, however, the last line of the formula says that if $flight(X, Y)$ and $flight(Y, Z)$ become true during program execution, then $flight(X, Z)$ is guaranteed to become true later in the execution. Note that, unlike formulas in Dynamic Logic, this formula says that $flight(X, Z)$ will be true at some time *during* program execution, but not necessarily at program termination.

Unlike Prolog (and related approaches, such as Transaction Logic), 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 at all, 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 contrast, in logic programming, the logic *is* the programming language; *i.e.*, logical formulas represent programs and specify database queries. Fourth, Process Logic and Dynamic Logic were not originally designed for database programming [Pra79,HKP82,Har79]. For instance, unlike the above adaptation of these logics (which we borrowed from [SWM93,Spr94]), they do not have the notions of database state or database query. In addition, they do not support defined procedures, such as subroutines and views.

2.3 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]. Recently, it has received renewed development by Reiter [Rei92a,Rei92b,Rei91].

Unlike the approaches discussed so far, the emphasis in the situation calculus is on specifying elementary actions, not on combining them into complex procedures. As such, the situation calculus does not have a repertoire of built-in actions, such as *assert* and *retract* in Prolog, or variable assignment in Dynamic Logic. Nor does it have rich control constructs, such as conditionals, iteration and subroutines. This is because the situation calculus was not designed for computer programming, but for reasoning about actions in the real world. The canonical example is the *blocks world*, in which a robot arm manipulates toy blocks sitting on a table top, as illustrated in Example 2. A typical problem is to specify the effects of lifting a block, or of putting one block down on top of another. Such actions may have several different effects. For instance, when a robot picks up a block, the block changes position, the block beneath it becomes clear, and the robot hand becomes full. Actions may also have pre-conditions on their execution. For instance, a (one-armed) robot cannot pick up a block if its hand is already holding something. Specifying all the pre-conditions and all the effects of such actions is a central aim of the situation calculus. Another aim is to treat them as *elementary* actions, not as a composition of simpler actions. Thus, a solution like that given in Example 2, in which the effects of an action are described by composing *assert* and *retract* actions, is considered unacceptable.

In the situation calculus, database states and actions are both denoted by function terms. For instance, the function term $pickup(b)$ might denote the action of a robot picking up block b. A function term denoting a database state is called a *situation*. The constant symbol s_0 denotes the initial state, before any actions have taken place. If s is a situation and a is an action, then the function term $do(a, s)$ is a situation denoting the state derived from s by applying action a. Thus, applying the actions a_1, a_2, a_3 to the initial state in that order gives rise to a state denoted by the situation $do(a_3, do(a_2, do(a_1, s_0)))$. Predicates whose truth depends on the situation are called *fluents*. For instance, the atomic formula $on(b_1, b_2, s)$ is a fluent. Intuitively, it means that block b_1 is on top of block b_2 in situation s. We adopt the convention that the last argument of a fluent is a situation. Uncertainty about database states (*e.g.*, "block a is on block b or c") is represented by more complex formulas, such as disjunctions of fluents.

Unlike database states, actions in the situation calculus must not contain any uncertainty. Thus, the action "pick up block b or block c" is not allowed. Moreover, for each action, we must say not only what it changes, but also what it does *not* change. The need to do this is illustrated in the following example, which also shows how the robot actions in Example 2 are represented in the situation calculus.

Example 3. **(Robot Actions)** Consider a simplified blocks world described by two fluents and one action, as follows:

- $clear(x, s)$: in state s, there are no blocks on top of block x.
- $on(x, y, s)$: in state s, block x is on top of block y.
- $move(x, y)$: move block x on top of block y, provided that x and y are clear.

One might think of axiomatizing the *move* action by the following rules:

$$possible(move(X,Y), S) \;\leftarrow\; clear(X,S) \;\wedge\; clear(Y,S) \;\wedge\; X \neq Y$$
$$on(X, Y, do(move(X,Y), S)) \;\leftarrow\; possible(move(X,Y), S)$$
$$clear(Z, do(move(X,Y), S)) \;\leftarrow\; possible(move(X,Y), S) \;\wedge\; on(X,Z,S)$$

The first rule describes the pre-conditions of the *move* action. It says that it is possible to execute $move(X, Y)$ in situation S if blocks X and Y are clear. The next two rules describe the effects of the action, assuming the action is possible. The second rule says that after executing $move(X, Y)$, block X is on top of block Y. The third rule says that if block X is on top of block Z in situation S, then after executing $move(X, Y)$, block Z is clear.

The above rules are called *effect axioms*, since they describe which facts are affected by the actions. These rules are not sufficient, however, because in some situations, many other formulas might be true, but they are not logically implied by the effect axioms. To see this, consider a database state described by the following atomic formulas:

$$clear(a, s_0) \qquad clear(b, s_0) \qquad clear(c, s_0) \qquad on(a, d, s_0) \qquad on(b, e, s_0) \quad (3)$$

The situation s_0 in these formulas indicates that they refer to the *initial* database state. Observe that block c is clear in this situation. Now, suppose we "execute" the action $move(a, b)$, thus changing the situation from s_0 to $do(move(a, b), s_0)$. Is block c still clear in the new situation? Our common sense says "yes," since c should not be affected by moving a onto b. However, the formula $clear(c, do(move(a, b), s_0))$ is *not* a logical consequence of the set of formulas specified so far.

In order to make $clear(c, do(move(a, b), s_0))$ a logical consequence of our blocks-world specification, we need formulas that say what fluents are *not* changed by the action $move(a, b)$. These formulas are called *frame axioms.* In general, a great many things are not changed by an action, so there will be a great many frame axioms. For instance, when a robot picks up a block, the color of the block does not change, the position of other blocks do not change, the number of blocks does not change, etc. Specifying all the invariants of an action in a succinct way is known as the *frame problem.*

Early solutions to this problem required one axiom for each action-fluent pair [Gre69]. For instance, in the example above, the following rule would be a frame axiom:

$$clear(Z, do(move(X, Y), S)) \;\leftarrow\; possible(move(X, Y), S) \wedge clear(Z, S) \wedge Z \neq Y$$

This rule pairs the action *move* with the fluent *clear*. It says that if block Z is clear in situation S, then it is also clear after moving block X onto block Y, provided that $Z \neq Y$. In general, specifying frame axioms in this way requires $O(MN)$ rules, where N is the number of actions, and M is the number of fluents.

In [Kow79], Kowalski presents a more succinct solution to the frame problem, which requires only $O(M + N)$ frame axioms. This solution relies on the Closed World Assumption (CWA). Other AI researchers continued searching for an equally simple solution, but one that does not require CWA. Recently, Reiter has found just such a solution [Rei91]: it also requires only $O(M + N)$ axioms, but is based on the Open World Assumption (OWA). We shall describe Reiter's approach here.

According to Reiter [Rei91], formalizing an action requires one axiom for each action (to describe its preconditions), and one axiom for each fluent (to describe the effect of actions on the fluent). In these axioms, all free variables are universally quantified at the top level. In Example 3, three axioms would be required. Here is the axiom for the *move* action, called a *pre-condition axiom*:

$$possible(move(X, Y), S) \;\leftrightarrow\; [clear(X, S) \;\wedge\; clear(Y, S) \;\wedge\; X \neq Y]$$

The following are the axioms for the two fluents, *on* and *clear*, called *successor-state axioms*:

$$
\begin{array}{l}
possible(A,S) \rightarrow \\
\quad [on(X,Y,do(A,S)) \leftrightarrow [A = move(X,Y) \vee \\
\qquad\qquad\qquad\qquad\qquad (on(X,Y,S) \wedge \neg \exists Z\ A = move(X,Z))]]
\end{array}
$$

$$
\begin{array}{l}
possible(A,S) \rightarrow \\
\quad [clear(Z,do(A,S)) \leftrightarrow [(\exists X,Y\ \ on(X,Z,S) \wedge A = move(X,Y)) \vee \\
\qquad\qquad\qquad\qquad\qquad (clear(Z,S) \wedge \neg \exists X\ A = move(X,Z))]]
\end{array}
$$

Successor-state axioms is a cross between the effect axioms and the frame axioms, as they replace both types of axioms.

To this, we must also add the various *unique name axioms* for actions, such as the following:

$$\forall X_1, Y_1, X_2, Y_2\ [move(X_1, Y_1) = move(X_2, Y_2) \ \leftrightarrow\ X_1 = X_2 \wedge Y_1 = Y_2]$$

whose sole purpose is to say that actions that look different syntactically are, in fact, different actions. There are $O(N^2)$ axioms of this type, so strictly speaking, Reiter's approach requires $O(M + N^2)$ axioms, not $O(M + N)$. However, Reiter argues that the unique name axioms can be ignored, since their effects can be compiled into a theorem prover [Rei91]. That is, although the theory itself has size $O(M + N^2)$, it is only necessary to write down a fragment of the theory of size $O(M + N)$ when doing reasoning. We refer to the former as the "entire theory," and to the latter as the "abbreviated theory."[4]

Let *Axioms* denote Reiter's entire theory for the simple blocks world described above plus the description of the initial state (3). From *Axioms*, we can use first-order classical logic to infer when actions are possible and when fluents are true. For example, from the pre-condition axiom above, it follows that the action $move(a, b)$ is possible in the initial state, s_0. That is, $Axioms \models possible(move(a, b), s_0)$. If we let $s_1 = do(move(a, b), s_0)$, then using the two successor-state axioms, we can infer the following facts about situation s_1:

$$Axioms\ \models\ clear(a, s_1) \wedge clear(d, s_1) \wedge clear(c, s_1) \wedge on(a, b, s_1) \wedge on(b, e, s_1)$$

Observe that only the facts $on(a, b, s_1)$ and $clear(d, s_1)$ are the direct effects of the action. The remaining facts are "carryovers" from the previous state—the effect achieved by frame axioms.

From the above, it follows that $Axioms \models possible(move(a, c), s_1)$, *i.e.*, the action $move(a, c)$ is possible in state s_1. If we let $s_2 = do(move(a, c), s_1)$, then using the two successor-state axioms above, we can infer the following facts about situation s_2:

$$Axioms\ \models\ clear(a, s_2) \wedge clear(d, s_2) \wedge clear(b, s_2) \wedge on(a, c, s_2) \wedge on(b, e, s_2)$$

[4] If we restrict our attention to Herbrand interpretations, then unique names axioms for actions are not needed. However, in this case, one looses the ability to identify different syntactic terms. For instance, formulas like $president(usa) = clinton$ and $\forall P\, husband(wife(P)) = P$ are unsatisfiable. For this reason, AI researchers often refuse to use Herbrand interpretations when reasoning about open worlds.

Just as before, $clear(b, s_2)$ and $on(a, c, s_2)$ are the direct results of the action $move(a, c)$; the other facts are carryovers from state s_1.

3 Declarative Languages for Database Transactions

At a first glance, there would seem to be many logics suitable for specifying database transactions, since many logics reason about updates, time, or 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 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 in several ways, some of which were mentioned in Section 2. Here, we generalize and elaborate on these reasons:

(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 database 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 (cf. the programs used in formulas (1) and (2) in Section 2.2). Moreover, logic is not used *inside* programs to specify database queries, but rather *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 run-time 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 run-time system, so that programs are executed by proving theorems.

(iv) Many logics of action get bogged down by the frame problem. As described in Section 2.3, this is the problem of logically specifying the large number of things that are unaffected by an action. If one is to reason about actions, then these invariants must all be specified as logical axioms (or 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. The same applies to database transactions, if they are expressed in an appropriate language. Many of the logical languages described in this section exploit this idea.[5]

Another fault line that separates the database languages described in this section is the issue of whether update literals are in the head of the rules or in the body. The languages in the "updates in the body" camp are well suited to programming, that is, to combining simple transactions into more complex transactions. The languages in the "updates in the head" camp are well suited to specifying bulk updates (which can be used as elementary transactions by the "updates in the body" languages). However, languages with updates in the head usually lack a subroutine facility so, by themselves, they do not provide the theoretical foundations for a full-blown database programming language.

The updates-in-the-body camp is represented in this survey by Transaction Logic, Dynamic Prolog, LDL, and Ultra. The updates-in-the-head camp includes the update languages of Abiteboul and Vianu, Chen's update calculus, and Datalog with State. The event calculus of Kowalski and Sergot can also be classified as an updates-in-the-head language.

3.1 Transaction Logic

Transaction Logic (abbreviated $\mathcal{TR}$) 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, with a natural model theory, 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 [BK98b]. Like classical Horn logic, the Horn fragment of $\mathcal{TR}$ has an operational semantics based on an SLD-style proof procedure, in the logic programming tradition. Interestingly, even though $\mathcal{TR}$ is a logic in which state change is a central feature, it does not rely on frame axioms when programming and executing transactions. This situation is quite similar to Prolog, but is quite unlike the situation calculus, where the frame problem has been a major issue for many years [MH69,Rei91]. Instead, frame axioms are needed in $\mathcal{TR}$ only when *reasoning* about the properties of actions, and only then does the frame problem become an issue [BK98b].

Historically, there are two versions of $\mathcal{TR}$: sequential $\mathcal{TR}$ [BK98a,BK93,BK95,BK94] and concurrent $\mathcal{TR}$ [BK96,Bon97b]. In

[5] In addition, some of the languages surveyed here assume a *fixed* set of elementary updates, such as the insertion and deletion of tuples in a relational database. In this case, it is usually easy to find a set of frame axioms that is linear in the number of predicates being updated; so the frame problem does not arise even for reasoning.

both versions, users can specify, execute and reason about logic programs with updates. In sequential $\mathcal{TR}$, the main method for combining programs is sequential composition. Concurrent $\mathcal{TR}$ extends sequential $\mathcal{TR}$ with concurrent composition. In concurrent $\mathcal{TR}$, concurrent processes can execute in isolation (like database transactions), or they may interact and communicate (like concurrent programs). From a database perspective, sequential $\mathcal{TR}$ allows users to specify transaction programs with built-in atomicity and save-points, which allow partial roll-back of transactions. Concurrent $\mathcal{TR}$ allows users to specify more-general nested transactions with intra-transaction concurrency. Because of space limitations, the rest of this section focuses on sequential $\mathcal{TR}$, which we refer to simply as $\mathcal{TR}$.

Syntactically, sequential $\mathcal{TR}$ extends first-order classical logic with a new logical operator called serial conjunction, denoted $\otimes$. Intuitively, if α and β are $\mathcal{TR}$ formulas representing programs, then the formula $\alpha \otimes \beta$ also represents a program, one that first executes α and then executes β. $\mathcal{TR}$ also interprets the connectives of classical logic in terms of action. For instance, the formula $\alpha \vee \beta$ intuitively means "*do α or do β*;" and the formula $\neg\alpha$ intuitively means "*do something other than α*." In this way, $\mathcal{TR}$ uses logical connectives to build large programs out of smaller ones. The smallest programs, called *elementary operations*, are described below.

Semantically, formulas in $\mathcal{TR}$ are evaluated on *paths*, *i.e.*, on sequences of states (as in Process Logic). Intuitively, if ϕ is a $\mathcal{TR}$ formula representing an action, then ϕ is true on a path s_1, s_2, ..., s_n if the action can execute on the path, *i.e.*, if it can change the database state from s_1 to s_2 to ... to s_n. Note that unlike Process Logic, $\mathcal{TR}$ does not use a separate, procedural language to specify programs, since programs are specified in the logic itself. $\mathcal{TR}$ is thus a single language that can be used in two ways: to represent programs (*i.e.*, logic programs), and to reason about them.

When used for reasoning, properties of programs are expressed as $\mathcal{TR}$ formulas, and a logical inference system derives properties of complex programs from those of simple programs [BK98b]. In this way, one can reason, for instance, about whether a transaction program preserves the integrity constraints of a database. One can also reason about when a transaction program will produce a given outcome, when it will abort, and about its effect on a database with null values or with other sources of indefinite information, such as disjunction. In addition, one can reason about actions in an AI context. For instance, using $\mathcal{TR}$, a user can axiomatize the elementary actions of a robot and reason about them. In AI terminology, this reasoning takes place in *open worlds*, that is, in the absence of the closed world assumption. The assumption of open worlds separates the $\mathcal{TR}$ theory of reasoning [BK98b] from the $\mathcal{TR}$ theory of logic programming [BK98a,BK93,BK95,BK94], which is based on closed worlds. The rest of this section elaborates on the theory of logic programming.

Transaction Logic Programming. Logic programs in sequential $\mathcal{TR}$ are based on *serial goals*. These are formulas of the form $a_1 \otimes a_2 \otimes ... \otimes a_n$, where each a_i is an atomic formula. Intuitively, this formula says, "First execute a_1, then execute a_2, then a_3 ... and finally a_n." A *serial-Horn rule* is a formula of the form $b \leftarrow \psi$, where b is an atomic formula, and ψ is a serial goal. This rule has the effect of making b a name of program ψ. Intuitively, it says, "To execute b, it is sufficient to execute ψ." A *serial-Horn program* (also called a $\mathcal{TR}$ *program*) is a finite set of serial-Horn rules, as illustrated in Example 4 below.

Classical Horn logic is a special case of serial-Horn $\mathcal{TR}$. To see this, observe that a serial-Horn program can be transformed into a classical Horn program 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 $b \leftarrow a_1 \wedge ... \wedge a_n$. In the absence of state changes, this transformation is a logical equivalence, and serial-Horn $\mathcal{TR}$ reduces to classical Horn logic. For instance, the following serial-Horn program

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

expresses the same database query as the following classical Horn program:

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

i.e., both programs compute the transitive closure of relation r.

As Example 4 shows, $\mathcal{TR}$ programs look very much like Prolog programs with *assert* and *retract*. Operationally, they also behave like Prolog *except* that updates are treated as database transactions; *i.e.*, they are *atomic*, and are rolled back whenever a program fails. This one difference leads to a simple and natural model theory for logic programs with destructive updates. It also improves upon Prolog with *assert* and *retract* in several ways. For instance, (*i*) the semantics of updating programs is purely logical; (*ii*) the semantics does not depend on rule order; (*iii*) programs are easier to understand, debug and maintain; and (*iv*) it extends the logic programming paradigm to a wide range of database applications, in which updates are transactional.

Specifying and executing $\mathcal{TR}$ programs is also similar to Prolog. To specify programs, the user writes a set of serial-Horn rules. These rules define transactions, including queries, updates, or a combination of both. To execute programs, the user submits a serial goal 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. Transactional features such as abort, rollback, and save-points are also handled by the theorem prover [Bon97b]. $\mathcal{TR}$ programs thus retain all the traditional features of classical logic programs, while providing a logical semantics for database updates, and an operational semantics for atomic transactions.

Example 4. **(Blocks World Revisited)**

$$stack(N,X) \leftarrow N > 0 \otimes move(Y,X) \otimes stack(N-1,Y)$$

$$stack(0,X) \leftarrow$$

$$move(X,Y) \leftarrow pickup(X) \otimes putdown(X,Y)$$

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

$$putdown(X,Y) \leftarrow wider(Y,X) \otimes clear(Y) \otimes on.ins(X,Y) \otimes clear.del(Y)$$

Here, $stack(N,X)$, $move(Y,X)$, etc., are atomic formulas just as in classical logic. However, in $\mathcal{TR}$, atomic formulas can (and in this case do) represent actions. 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 rules defining these actions combine simple actions into more complex ones. For instance, the rule defining $move(X,Y)$ can be read as follows: "*First pick up block X, and then put X down on top of block Y.*" The action $move(X,Y)$ succeeds if these two sub-actions succeed. In this case, the sub-actions themselves are defined in terms of elementary queries and updates to a relational database. For instance, the rule defining $putdown(X,Y)$ can be read as follows: "*First check that block Y is wider than block X, then check that block Y is clear, then insert the atom $on(X,Y)$ into the database, then delete the atom $clear(Y)$ from the database.*" The action $putdown(X,Y)$ succeeds if all these elementary operations succeed.

Observe that the actions *pickup* and *putdown* are deterministic, since each set of argument bindings specifies only one robot action. 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 in the first rule). If, at any step, several such blocks can be placed on top of the stack, the system arbitrarily chooses one of them.

Elementary Operations. Example 4 illustrates two kinds of elementary database operation: the insertion and deletion of atomic formulas. For instance, the expression $on.del(x,y)$ means, "Delete the atom $on(x,y)$ from the database," and the expression $clear.ins(y)$ means, "Insert the atom $clear(y)$ into the database." Here, *on.del* and *clear.ins* are predicate symbols, and $on.del(x,y)$ and $clear.ins(y)$ are atomic formulas.

This illustrates how $\mathcal{TR}$ might be used to update a relational database. In general, however, an elementary operation in $\mathcal{TR}$ can be *any* transformation on *any* kind of database state. For instance, an elementary operation could perform row operations on a set of matrices, or it could perform SQL-style bulk updates on a set of relations, or it could insert or delete a logical formula from a set of formulas. In some applications, such as workflow management, a state may be a combination of files and databases, and elementary operations may include any number of application programs and legacy systems [BSR96,DKRR98]. In all cases, however, the elementary operations are building blocks from which larger programs and software systems are built.

$\mathcal{TR}$ provides the logical foundations and a logic programming language for building such systems. To achieve the needed 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 programs. Formally, database states and elementary operations are described by a pair of oracles, called the *state oracle* and the *transition oracle*, respectively. Intuitively, the state oracle describes what must be true at states, while the transition oracle describes how the state is changed. A number of transition oracles, for tuple updates, for random sampling, and for SQL-style bulk updates, are described in detail in [BKC94,BK95].

Constraints on Execution. Transaction Logic has two kinds of conjunction: serial (denoted $\otimes$) and "classical" (denoted $\wedge$).[6] (In addition, concurrent Transactions Logic has a third kind of conjunction, called *concurrent* conjunction.)

For formulas that do not involve updates, these two types of conjunction are the same—they both reduce to the usual conjunction of classical logic. However, when updates are involved, the two types of conjunction in $\mathcal{TR}$ part their way and they extend the classical connective in two distinct ways.

As mentioned earlier, $\alpha \otimes \beta$ means that transaction α executes first and transaction β next. In contrast, $\alpha \wedge \beta$ means that both α and β must execute *along the same execution path*. In other words, $\alpha \wedge \beta$ executes along a path $\langle s_1, s_2, ..., s_n \rangle$ iff α (by itself) can cause the state transitions $s_1, s_2, ..., s_n$ *and* β (by itself) can also cause the state transitions $s_1, s_2, ..., s_n$. For instance, α could be the statement "Build a pyramid" (like the *stack* transaction defined earlier), while β could be the statement "Spell a word." Assuming that our toy blocks have letters painted on them, the formula $\alpha \wedge \beta$ represents the (possibly very non-deterministic) process of building a pyramid that spells a word.

This semantics leads to a natural interpretation of "$\wedge$" as a mechanism for applying *constraints* that prune away any undesirable executions of a nondeterministic transaction. Such constraints are developed to a greater depth in [BK95], where a rich class of temporal and dynamic constraints is discussed. Some results on executional constraints also appear in [DKRR98].

Note that it would not be possible to express constraints on transaction execution (not to mention reasoning about them) if $\mathcal{TR}$ had a semantics based on pairs of states (like Dynamic Logic) instead of paths. For instance, such capabilities are beyond the expressive power of a number of languages surveyed below simply because their semantics is based on Dynamic Logic (see, *e.g.*, Sections 3.2, 3.3, 3.6). This gain in expressive power due to a path-based semantics is similar to the benefits of Process Logic relative to Dynamic Logic, as discussed in Section 2.2.

Executional Entailment. $\mathcal{TR}$ represents elementary operations by atomic formulas. Like all formulas in $\mathcal{TR}$, elementary operations can have *both* a truth

[6] The connective "$\wedge$" is *not* the same as the corresponding connective in classical logic. It is called "classical" because of the superficial similarity between its definition in $\mathcal{TR}$ and the corresponding classical definition.

value *and* a side effect on the database. This idea is formally represented by *executional entailments*. For instance, if $p.del$ represents the deletion of atom p from the database, then this would be represented by the following entailment:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} - \{p\} \models p.del$$

This statement says that the atomic formula $p.del$ is (the name of) an update that changes the database from state $\mathbf{D}$ to state $\mathbf{D} - \{p\}$.

More generally, transaction execution is represented as executional entailment over a *sequence* of database states:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi \tag{4}$$

Here, ψ is a $\mathcal{TR}$ formula, each $\mathbf{D}_i$ is a database state, and $\mathbf{P}$ is a set of $\mathcal{TR}$ formulas. Intuitively, ψ is the main procedure of a program, $\mathbf{P}$ is a set of subroutine definitions, and $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ is the database history during program execution. In the formal semantics, statement (4) means that formula ψ is true with respect to the sequence $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$. Informally, this means that that the sequence is an execution path of program ψ. That is, if the current database state is $\mathbf{D}_0$, and if the user executes program ψ (by typing $?- \ \psi$, as in Prolog), then the database *can* 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 program terminates. We emphasize the word "can" because ψ can be a non-deterministic program. 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 in practice, 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.

Executional entailment plays an important role in Transaction Logic, as it ties together the notions of logical truth and program execution. It also provides the logical foundation for a proof procedure that executes transactions, updates the database and evaluates queries, all as a result of proving theorems [BK98a,BK96,BK95,BK94].

3.2 Dynamic Prolog

Manchanda and Warren [MW88] developed Dynamic Prolog, a logic programming language for database transactions. Although the authors were greatly influenced by Dynamic Logic in the design of the semantics, their language is very different from Dynamic Logic. In fact, Dynamic Prolog has many similarities with Transaction Logic, both in syntax and in spirit. For instance, Transaction Logic 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 its 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. Realizing this problem, Manchanda and Warren developed an interpreter whose aim was to "execute" transactions. However, this interpreter is incomplete with respect to the model theory, and 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.

While Dynamic Prolog is only slightly reminiscent of Dynamic Logic, other researchers explored the potential of this logic for database updates in much more detail. For instance, in [SWM93,Spr94], Spruit et al. proposed a database update language that is related to Dynamic Logic not only semantically but also in its syntax. In fact, this work is an adaptation of Dynamic Logic to database updates and the sample program (1) in Section 2.2 is inspired by [SWM93,Spr94].

3.3 LDL

Naqvi and Krishnamurthy have extended classical Datalog with update operators [NK88], which were later incorporated in the LDL language [NT89].

The treatment of updates in LDL is similar to that of Dynamic Prolog and Transaction Logic. In particular, updates are performed in the body of rules, and complex update procedures can be defined from simpler ones. Thus, like in $\mathcal{TR}$ and Dynamic Prolog, the LDL rule $q \leftarrow prog$ amounts to a procedure declaration, where q is the procedure name and $prog$ is the procedure definition. For instance, consider the Transaction Logic program given in Example 4. The final three rules of this program can be written in LDL as follows:

$$\begin{aligned}
&move(X,Y) \;\leftarrow\; pickup(X)\,;\, putdown(X,Y)\\
&pickup(X) \;\leftarrow\; clear(X)\,;\, on(X,Y)\,;\, -on(X,Y)\,;\, +clear(Y).\\
&putdown(X,Y) \;\leftarrow\; wider(Y,X)\,;\, clear(Y)\,;\, +on(X,Y)\,;\, -clear\,(Y).
\end{aligned}$$

Here, ";" denotes sequential composition, and the literals $+on$, $-clear$, etc., are special predicates that insert and delete tuples from the predicates on and $clear$. They are analogous to the elementary transitions $on.ins$ and $clear.del$ in Transaction Logic.

Syntax. Compared to other proposals reviewed in this survey, the syntax of LDL is quite complex. LDL distinguishes several kinds of predicate symbols, including base predicates, query predicates, update predicates, compound predicates, and unfailing predicates. In addition, there are two kinds of update predicate (corresponding to insertion and deletion), and several kinds of compound predicate (corresponding to iteration, conditionals, sequential composition, etc.).

Finally, a variety of syntactic restrictions are imposed on LDL rulebases, in order to limit recursion and to achieve a workable semantics. For instance, recursively defined updates are not allowed. Thus, the first rule in Example 4 is meaningless in LDL:

$$stack(N, X) \leftarrow N > 0\,;\, move(Y, X)\,;\, stack(N-1, Y).$$

More generally, recursion is disallowed through all procedural constructs, such as sequential composition, *if-then-else*, and *do-while*.

To limit recursion, LDL programs are stratified, which limits the set of legal programs even further. In fact, many non-recursive programs turn out to be non-stratified, and therefore are not legal. For instance, if a program contains the rule $p \leftarrow r_1; r_2$ then predicate r_1 must be fully computed before the computation of predicate r_2 begins. To enforce this, r_1 must be defined in a *strictly lower* stratum than r_2. Thus, the following program cannot be stratified, even though it is non-recursive:

$$p \leftarrow r_1\,;\, r_2 \qquad\qquad q \leftarrow r_2\,;\, r_1$$

In addition, LDL requires that each updating procedure be defined by only one rule. Thus, the following two rules cannot coexist in one program:

$$q \leftarrow +b \qquad\qquad q \leftarrow +c$$

The implementation of LDL imposes restrictions of its own, essentially to avoid the possibility of backtracking through updates. For instance, in

$$q \leftarrow a\,;\, +b\,;\, c$$

the query c might fail after inserting b, and the declarative semantics requires that the update $+b$ be undone. Since this basic transactional feature is not implemented in LDL, the programmer must ensure that all updates in an LDL program are followed by *unfailing predicates* [NT89] (predicates that never fail). Primitive updates are examples of unfailing predicates. Other unfailing predicates can be constructed from the *if-then-else* and *do-while* constructs.

Bulk Updates. Since LDL is geared towards database applications, it provides for bulk updates. As a simple example, consider the following LDL query:

$$?-\ p(X)\,;\, +q(X).$$

The intended meaning of this query is to copy all the tuples of relation p into relation q; *i.e.*, first retrieve all the tuples in relation p, and then insert them into relation q.

This treatment of programs with variables brings out a basic difference in the semantics of LDL and Transaction Logic. For instance, consider the above rules for the predicate $move(X, Y)$. In $\mathcal{TR}$, the query $?-\ move(b, Y)$ is non-deterministic: it means, "Move block b on top of *some* block, Y." In contrast, the same query in LDL is deterministic: it says, "Move block b on top of *every* block, Y."

Unfortunately, the declarative semantics of bulk updates in LDL has never been spelled out clearly. The semantics developed in [NK88,NT89] is highly propositional, with almost no mention of variables or quantifiers. Bulk updates are described operationally in terms of a "mark and update" procedure, but the declarative semantics focuses entirely on variable-free programs.

Since the semantics of updates in LDL is said to be based on *propositional* dynamic logic, it is unclear how it can be extended to deal with variables and bulk updates. In fact, some of the restrictions mentioned above seem to preclude such an extension. For instance, the following example, paraphrased from [NT89]:

$$
\begin{array}{l}
q(a). \\
p \leftarrow q(X),\ +r(X). \\
?-\ p.
\end{array}
$$

inserts a single tuple $r(a)$, and so its semantics is easy to define in propositional dynamic logic. However, if the database had two facts, $q(a)$ and $q(b)$, then the ground instantiation of the above rule is a *pair* of ground rules: $p \leftarrow q(a),\ +r(a)$ and $p \leftarrow q(b),\ +r(b)$. But this means that the same predicate (in fact, the same proposition), p, is defined by two rules—in direct violation of the syntactic restrictions imposed by LDL.

In sum, LDL syntax for updates is highly restrictive, and the implementation does not always correspond to its declarative semantics. On one hand, the implementation provides for bulk updates, while the declarative semantics does not. On the other hand, the declarative semantics accounts for the rollback of failed update programs, but the implementation does not.

3.4 The Event Calculus

Like the situation calculus, the event calculus is a methodology for encoding actions in first-order predicate logic. It was originally developed by Kowalski and Sergot for reasoning about time and events in a logic-programming setting [KS86,Kow92] and for overcoming some of the problems of the situation calculus. A comparison of the event calculus and the situation calculus is developed in [KS97].

Unlike Transaction Logic and LDL, the event calculus is not a language for programming database transactions. In fact, like the situation calculus, the event calculus does not update the database at all. Instead, it is a query language for historical databases. However, these databases are not temporal databases in the traditional sense; *i.e.*, they do not record the state of the world at various points in time. Instead, they record the events that have taken place, just as a logical database log records the transactions that have executed. For instance, a database recording robot actions might include the following information:

> On 5.feb.98, Robot Billy picks up Block 1 from the top of Block 2.
> Two days later, Billy puts down Block 1 on top of Block 3.
> Two more days pass by, and Billy picks up Block 1 again.

One goal of the event calculus is to reason about event descriptions like this. In this case, "reasoning" amounts to querying the event database, where queries are formulated as logic programs. To this end, the event calculus includes Prolog-like rules for inferring the consequences of events. Given a complete database of events, these rules can derive the state of the world at any point in time. For instance, they can infer that Block 1 is still on top of Block 3 on February 8, and that the tops of Blocks 2 and 3 are still clear on February 10.

Several variants of the event calculus have been developed [SK95]. They are all based on variants of the *"Holds" formalism* [Kow79], and they all assume that actions take place instantaneously. The original event calculus [KS86] was the most complex, partly because its ontology is based on maximal time periods. In the rest of this section, we describe the so-called *simplified event calculus* [SK95], whose ontology is based on time points.

The Simplified Event Calculus. This variant of the event calculus was designed for reasoning about *complete* event descriptions, *i.e.*, databases that contain a complete description of all relevant events. It is based on the following Prolog-like rules, in which negation is interpreted as failure:[7]

$$holds(P, T_2) \leftarrow initiated(P, T_1),\ T_1 < T_2,\ \neg terminated(P, T_1, T_2). \tag{5}$$

$$initiated(P, T) \leftarrow initiates(E, P),\ happens(E, T). \tag{6}$$

$$terminated(P, T_1, T_2) \leftarrow terminates(E, P),\ happens(E, T_3),\ T_1 < T_3 < T_2. \tag{7}$$

The predicates in these rules have the following intuitive meanings:

- $holds(P, T)$ means, "*proposition P is true at time T.*"
- $initiated(P, T)$ means, "*proposition P becomes true at time T.*"
- $terminated(P, T_1, T_2)$ means, "*proposition P becomes false at some time between T_1 and T_2.*"
- $happens(E, T)$ means, "*event E occurred at time T.*"
- $initiates(E, P)$ means, "*event E makes proposition P true.*"
- $terminates(E, P)$ means, "*event E makes proposition P false.*"

Thus, rule (5) says that P is true at time T_2 if it becomes true at time T_1 and does not become false between times T_1 and T_2. Rule (6) says that P becomes true at time T if event E makes P true, and E happens at time T. Likewise, rule (7) says that P becomes false at some time between T_1 and T_2 if event E makes P false, and E happens at time T_3, and T_3 is between T_1 and T_2. The use of negation-as-failure in rule (5) means that a fact remains true unless it is explicitly terminated by an event. Observe that explicit frame axioms are not given, and thus the simplified event calculus solves the frame problem for closed worlds.

[7] The use of negation-as-failure implies that event descriptions must be complete.

In addition to the three rules above, rules are needed to define the predicates *initiates*, *terminates* and <. For the purpose of this section, we define the latter predicate as follows:

$$5.feb.98 \ < \ 6.feb.98 \ < \ 7.feb.98 \ < \ 8.feb.98 \ < \ 9.feb.98 \ < \ 10.feb.98 \ < \ 11.feb.98$$

$$T_1 < T_2 \quad \leftarrow \quad T_1 < T_3, \ T_3 < T_1.$$

The rules for *initiates* and *terminates* depend on the kind of events in the application domain. To illustrate, we consider two kinds of event from the blocks world:

- $pickup(B_1, B_2)$, which means, "*pick up block B_1 from on top of block B_2.*"
- $putdown(B_1, B_2)$, which means, "*put down block B_1 on top of block B_2.*"

These two actions are axiomatized by the following rules:

$$\begin{aligned} initiates(E, on(B_1, B_2)) &\leftarrow act(E, putdown(B_1, B_2)) \\ terminates(E, on(B_1, B_2)) &\leftarrow act(E, pickup(B_1, B_2)) \\ initiates(E, clear(B_2)) &\leftarrow act(E, pickup(B_1, B_2)) \\ terminates(E, clear(B_2)) &\leftarrow act(E, putdown(B_1, B_2)) \end{aligned}$$

Here, the predicate $act(E, A)$ intuitively means that event E is an instance of action A. The first rule above therefore says the following: if event E puts block B_1 down on top of block B_2, then E makes the atom $on(B_1, B_2)$ true. By using the *act* predicate, different instances of the same action can take place at different times. For instance, different instances of the actions $pickup(block_1, block_2)$ and $putdown(block_1, block_2)$ can occur at several different times.

Finally, we specify the predicates *happens* and *act* as an event database. For instance,

$$\begin{array}{ll} happens(e1,\ 5.feb.98) & act(e1, pickup(block_1, block_2)) \\ happens(e2,\ 7.feb.98) & act(e2, putdown(block_1, block_3)) \\ happens(e3,\ 9.feb.98) & act(e3, pickup(block_1, block_3)) \end{array}$$

At this point, we are ready to answer queries about the state of the world. For instance, using the Prolog interpreter (with negation as failure), it is easy to derive the facts below. Observe that the first event makes $block_2$ clear, and that it remains clear as other events take place, thus illustrating that frame axioms are implicit (though not explicit) in the simplified event calculus.

$$\begin{array}{ll} holds(clear(block_2),\ 6.feb.98) & holds(clear(block_2),\ 7.feb.98) \\ holds(on(block_1, block_3),\ 8.feb.98) & holds(clear(block_2),\ 8.feb.98) \\ holds(on(block_1, block_3),\ 9.feb.98) & holds(clear(block_2),\ 9.feb.98) \\ holds(clear(block_3),\ 10.feb.98) & holds(clear(block_2),\ 10.feb.98) \\ holds(clear(block_3),\ 11.feb.98) & holds(clear(block_2),\ 11.feb.98) \end{array}$$

As seen from this example, a database in the event calculus is not like those found in traditional database applications. That is, the event calculus does not record the current state of the world, updating it whenever a transaction is executed. Nor is the event calculus like a temporal database system, as mentioned above. Instead, the event calculus queries a history of instantaneous events (or database transactions) to infer the state of the world at any given time.

This derivation can be done quite efficiently because in the simplified event calculus, all actions are write-only; *i.e.*, they represent transactions that change a database without reading it. For instance, the transaction "Set Joe's salary to $50,000" is write-only, while the transaction "Increase Joe's salary by 3%" is not, since it requires both a database read (to get the old salary) and a database write (to set the new salary). This write-only restriction follows from the limited syntax of the *initiates* and *terminates* predicates: since these predicates are independent of time, the effects of an action cannot depend on the state of the world. At best, limited information about the world can be encoded in the actions themselves, as in the action $pickup(block_1, block_2)$, which assumes (but does not verify) that $block_1$ is sitting on top on $block_2$. All of the actions considered in this section were carefully designed to be write-only. For such actions, the truth of a proposition P depends only on the most recent event to affect P. In this case, events can be indexed for fast retrieval, resulting in fast inference [Kow92].

3.5 Chen's Calculus

Chen developed a calculus and an equivalent algebra for constructing database transactions [Che95]. Like $\mathcal{TR}$, this calculus uses logical operators (including serial conjunction) to build complex transactions from elementary updates.

One interesting idea in [Che95] is a special semantics for the logical connective $\wedge$, which makes it easy to express *bulk updates* (assigning the contents of a database view to a relation). This is achieved by treating formulas of the form $a \wedge b$ as the *concurrent* composition of actions a and b, so that they are executed in parallel. Here is a typical example of the use of concurrent composition (where universal quantification is treated as an infinite conjunction of ground instantiations of the program):

$$\forall E, S \;\; +employee(E, S * 1.1) \wedge -employee(E, S) \;\leftarrow\; employee(E, S) \tag{8}$$

This rule specifies the tried-and-true transaction to "raise all employee salaries by 10%". Here, literals of the form $+p$ or $-p$ are called dynamic predicates. As in LDL, they represent the insertion and deletion of p, respectively.

Unlike LDL, dynamic predicates are allowed in rule bodies, as well as in rule heads. In the body, they refer to previously executed updates. That is, when $+p$ appears in a rule body, it intuitively means, "If p has just been inserted into the database." Likewise for $-p$. For instance, the rule $q \leftarrow -p$ means, "If p has just been deleted from the database, then infer q." In addition, the head of a rule can be a complex formula and not just a single dynamic literal, as in the following program:

$$\forall X,Y\ +path(X,Y)\ \leftarrow\ edge(X,Y)$$
$$\forall X,Y,Z\ +path(X,Y)\ \wedge\ -edge(X,Y)\ \leftarrow\ edge(X,Z)\ \wedge\ +path(Z,Y)\wedge Y\neq Z$$

This program does two things: (i) it computes the transitive closure of the *edge* relation, and (ii) it simultaneously removes edges that do not contribute to the connectivity of the graph (*i.e.*, it deletes $edge(x,y)$ if there is another path from x to y).

The semantics of such programs is determined with respect to pairs of states, as in Dynamic Logic. In addition, the semantics of concurrent actions requires a minimality principle, which roughly says that the new state should differ from the old state as little as possible. The minimality principle makes the calculus non-monotonic even in the absence of negation.

As with all "update in the head" languages, Chen's calculus lacks mechanisms for defining subroutines, which makes it unsuitable as a programming language for database transactions. However, Chen mentions in [Che95] that his calculus was intended as a language for specifying bulk and other non-trivial updates, which could then be used as elementary operations by a language like Transaction Logic.

3.6 Ultra

Wichert and Freitag [WF97] describe an update language, which later received the name Ultra. Ultra attempts to integrate several of the concepts used in other update languages, including sequential composition, concurrent composition, and bulk updates. For instance, like Chen's calculus, Ultra includes facilities for defining bulk updates declaratively; and like $\mathcal{TR}$, Ultra is an "updates in the body" language, so named subtransactions are possible. The semantics of Ultra is inspired by Dynamic Prolog.

As an example, here is an Ultra program that raises the salary of all employees by 10%:

$$\begin{aligned} raise(E,S)\ \leftarrow\ \#E,S\ employee(E,S) \gg \\ [DEL\,employee(E,S)\ :\ INS\,employee(E,S*1.1)] \end{aligned}$$

The expression $\#E,S\,employee(E,S)\gg$ in the rule body is *bulk quantification*, which is interpreted as a (possibly infinite) concurrent composition. This means that the expression following $\gg$ is executed for every tuple (E,S) in the *employee* relation. The connective ":" denotes sequential composition; so in this example, $DEL\,employee(E,S)$ is executed before $INS\,employee(E,S*1.1)$. (Of course, we could have used concurrent composition instead.)

Bulk quantification and concurrent composition are implemented by executing all the participating transactions hypothetically, and accumulating their individual update requests (which, as in Chen's calculus, requires a minimality principle). If these requests do not conflict with each other (*i.e.*, if they do not ask to insert and delete the same fact), then the updates are applied to the database. If they do conflict, then the transaction is aborted.

3.7 Datalog with State

A number of researchers have worked on adding the notion of state to Datalog programs [Zan93,LHL95]. In these works, states are represented through a special, distinct argument that is added to each updatable predicate. Updates are then modeled as state transitions. The approach contains elements reminiscent of both the situation calculus and the event calculus. For instance, like the situation calculus, states are identified with function terms; and like the event calculus, action instances (and action requests) can be stored in the database as atomic formulas. To illustrate these ideas, we sketch the approach of [LHL95], adapting it for the purpose of our presentation.

States are identified by positive integers (represented as function terms). For instance, the atom $on(b, c, 3)$ means that block b is on block c in state 3. The effects of actions are specified by Datalog-style rules. Each action begins at some state, s, and ends at some future state, $s+k$. For instance, we could define $insert\_on(X, Y, S)$ to be an action that starts at state S, ends at state $S+1$, and inserts into the database the fact that block X is on block Y. Likewise, $delete\_on(X, Y, S)$ could delete the fact that X is on Y. These actions would be defined by the following rules:

$$\begin{aligned} on(X, Y, S+1) &\leftarrow insert\_on(X, Y, S) \\ on(X, Y, S+1) &\leftarrow on(X, Y, S),\ \neg delete\_on(X, Y, S) \end{aligned}$$

The first rule says that X is on Y if this fact has just been inserted into the database. The second rule is a frame axiom. It says that X is on Y in a state if X was on Y in the previous state and this fact has not been deleted from the database by an action at state S.

To some extent, complex actions can be defined in terms of simpler actions. For instance, the following rules define the action $move$ in terms of the simpler actions $delete\_on$, $delete\_clear$, $insert\_on$, $insert\_clear$:

$$\begin{aligned} possible\_move(X, Y, S) &\leftarrow clear(X),\ clear(Y),\ X \neq Y \\ delete\_clear(Y, S) &\leftarrow possible\_move(X, Y, S),\ move(X, Y, S) \\ insert\_on(X, Y, S) &\leftarrow possible\_move(X, Y, S),\ move(X, Y, S) \\ delete\_on(X, Z, S) &\leftarrow possible\_move(X, Y, S),\ on(X, Z, S),\ move(X, Y, S) \\ insert\_clear(Z, S) &\leftarrow possible\_move(X, Y, S),\ on(X, Z, S),\ move(X, Y, S) \end{aligned}$$

The first rule says that it is possible to move block X onto block Y if both blocks are clear and if they are different blocks. The remaining rules all assume that the move action is possible. Notice that even if $move(x, y, s)$ is true, block x will not be moved onto block y unless the move action is possible in state s. This illustrates that the atom $move(x, y, s)$ is not a statement that the move action has occurred at state s; instead, it is a *request* to execute the move action at state s. Thus, in each of the last four rules, the premise requires that $move(X, Y, S)$ be possible and that it be requested. Only then will the action actually take place. In this case, the request for action $move(X, Y, S)$ effectively triggers requests for four other actions: $insert\_clear(Y, S)$, $insert\_on(X, Y, S)$, $delete\_on(X, Z, S)$, and $delete\_clear(Z, S)$. These four actions will all be executed concurrently. The ability to trigger actions in this way forms the basis of a theory of active databases [LHL95].

Although actions like *move* can be defined in terms of simpler actions, the ability of Datalog with State to combine actions is limited in comparison with languages like Transaction Logic. For instance, Datalog with State has no notion of sequential composition, and no notion of subroutine. Because of this, the applications of Datalog with State to database programming are limited. Recognizing this problem, the authors have extended Datalog with State to include nested transactions and procedures [LML96]. The extended language includes a form of sequential composition modeled on the connective of serial conjunction in Transaction Logic.

Finally, it is instructive to compare Datalog with State to Golog [LRL+97] and to Reiter's theory of database evolution [Rei95], described in Section 4.4. One important difference is that Datalog with State relies on a small, fixed set of elementary updates, so only a small, fixed set of frame axioms is needed. Another difference is that Datalog with State uses a form of closed-world semantics (XY-stratification [Zan93] or state-stratification [LHL95]), which is closer to the database tradition. Consequently, unlike Reiter's theory, Datalog with State has no problem in representing database views, recursive or otherwise. On the other hand, it is hard to see how Datalog with State might be used for reasoning about the effects of actions, which was the main result of Reiter's work.

3.8 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 comprehensive *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. These languages have much in common with Datalog with State, although they do not use state variables. Instead, each rule execution changes the database state by either inserting or deleting tuples specified in the rule head.

Like Datalog with State, the update languages of [AV91,Abi88] assume that databases are relational, and they do not support subtransactions, save-points, or partial abort and rollback. Furthermore, there is no support for subroutines. This can be seen most clearly in the procedural languages defined in [AV90], where the 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 [Bon97a].

To illustrate this approach, we adapt the program (8) previously considered in connection with Chen's calculus.

$$employee(E, S * 1.1) \wedge \neg employee(E, S) \leftarrow employee(E, S)$$

The negation sign in the rule head indicates that the corresponding atom is to be deleted; on the other hand, non-negated atoms in the rule heads are inserted into the database. If an *employee*-tuple, $\langle john, 10K \rangle$, is in the database, the above rule fires up and replaces this tuple with $\langle john, 11K \rangle$. However, the evaluation process does not stop here. The semantics of this update language is based on fixpoint computation, so the above rule is to be continuously fired up until a fixpoint is reached. So, John's salary will keep increasing indefinitely.

In general, writing update programs in such a language is not a simple matter. Abiteboul and Vianu suggest that, to be useful, update programs must be augmented with explicit control over iteration [AV91]. For instance, to prevent John from getting a raise that might dwarf the national debt, a control structure can effectively be added by using two rules, as follows:

$$employee(E, S * 1.1) \wedge \neg employee(E, S) \ \leftarrow \ employee(E, S) \wedge compute$$

$$\neg compute \ \leftarrow \ compute$$

Here, the proposition *compute* controls the execution of the salary raise rule. When the atom *compute* is inserted into the database, both rules can fire (provided that the employee relation is non-empty). Firing the first rule increases salaries, while firing the second deletes *compute* from the database, thereby disabling both rules.

The actual outcome of this process depends on the details of the operational semantics. For instance, if each enabled rule can fire non-deterministically, then John may not get any raise at all (if the second rule fires first), or he could become very rich, if the first rule is allowed to fire many times before the second rule interrupts this process.

Under a different semantics, the enabled rules fire all at once and then the corresponding changes are applied to the database, provided that different rules do not make conflicting changes, such as adding and deleting the same fact (this idea is also used in Ultra [WF97], as discussed in Section 3.6). Under this semantics, John gets the raise precisely once, after which both of the above rules are disabled.

4 Logics for Reasoning about Programs

4.1 Action Logic and Related Proposals

Pratt [Pra90] develops a logic, called *Action Logic*, that is superficially similar to Transaction Logic, but is different from it in essential ways. Like $\mathcal{TR}$, Action Logic does not distinguish between actions and propositions: actions are simply propositions that hold on intervals. However, the semantics and the intent of the two formalisms are very different. First, Action Logic is not a language for updating databases or defining transactions. In fact, it has no notion of database, no analogue of $\mathcal{TR}$'s transition oracle, and no counterpart to executional entailment. Instead, Action Logic is an extension of regular expressions, and as such, it can be viewed as a formalism for defining languages. Second, in contrast to

$\mathcal{TR}$'s semantics, which is based on sequences of states, Pratt develops a semantics based on *action algebras*. The proof theory for Action Logic is a (finite) set of equations for reasoning about these algebras.

Nevertheless, Action Logic is superficially similar to $\mathcal{TR}$. For instance, it has operators similar to $\otimes$, $\vee$, `state`, and $\neg$`path` in $\mathcal{TR}$;[8] and it has an iteration operator, which can be expressed in $\mathcal{TR}$ using recursion. What makes the comparison seem especially close, is a pair of connectives, denoted $\leftarrow$ and $\rightarrow$ in Pratt's notation, that look very similar to $\mathcal{TR}$'s connectives for serial implication.[9] Semantically, however, these connectives are very different. For instance, the following equation is an axiom of Action Logic:

$$a(a \rightarrow b) \ + \ b \ = \ b \tag{9}$$

Here, $+$ denotes ordinary disjunction, and concatenation denotes sequential composition. This equation is therefore a sequential version of modus ponens: it says that if $a(a \rightarrow b)$ is true, then b is also true. The same is not true in $\mathcal{TR}$. The analogue of equation (9) in $\mathcal{TR}$ is the following statement:

$$(a \otimes (a \Rightarrow b) \ \vee \ b) \ \leftrightarrow \ b$$

However, this statement is *not* a theorem of $\mathcal{TR}$. The intuitive reason is that the two occurrences of a in this statement do not refer to the same action. In fact, the second occurrence of a can begin only when the first occurrence ends. Hence, the truth of $a \otimes (a \Rightarrow b)$ does not imply the truth of b.

There is one more difference worth noting. Transaction Logic has *two* kinds of conjunction, classical ($\wedge$) and serial ($\otimes$). Combined with negation, they lead to two kinds of disjunction, and two kinds of implication, in a natural way. In contrast, Action Logic has *one* type of conjunction (serial conjunction), as pointed out in [Pra90]. Action algebras have a lattice-like "meet" operator that might form the model-theoretic basis for another kind of conjunction, but the meet operation is not always defined.

In [vB91], van Benthem outlines a number of logical approaches to dynamic information processing. In these approaches, actions are represented as propositions. One of these approaches is based on a *dynamic interpretation* of classical predicate logic. A dynamic interpretation associates a pair of states to each proposition, which resembles Dynamic Prolog of Manchanda and Warren [MW88]. However, the states associated to propositions by dynamic interpretations of [vB91] are not database states but rather variable assignments. van Benthem also discusses an algebraic approach to the logic of dynamic systems, which is akin to the action logic of Pratt [Pra90].

8 `state` is a proposition that is true precisely on paths of the form $\langle s \rangle$, where s is a state. In other words, `state` is true precisely at states. In contrast, `path` is a proposition that is true on every paths.

9 In Transaction Logic, the connective of serial conjunction, $\otimes$, leads naturally to a dual connective of serial disjunction, $\oplus$, where $\alpha \oplus \beta \ = \ \neg(\neg\alpha \otimes \neg\beta)$. This in turn leads naturally to the notions of left and right serial implication: $\alpha \Leftarrow \beta \ = \ \alpha \oplus \neg\beta$ and $\alpha \Rightarrow \beta \ = \ \neg\alpha \oplus \beta$. Using serial implication, one can express a wide variety of dynamic and temporal constraints in $\mathcal{TR}$ [BK95].

4.2 McCarty and Van der Meyden

In [MvdM92], McCarty and Van der Meyden develop a theory for reasoning about "indefinite" actions. However, [MvdM92] does not address action execution or the updating of databases. To give an idea of what [MvdM92] is about, consider a set consisting of exactly the following two rules in Transaction Logic:[10]

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

Here, 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 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.*

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 concurrent action, as it does in Chen's work [Che95]. In the other notion, called "satisfaction," the same symbol corresponds to constraints on action execution, as it does in $\mathcal{TR}$. However, the development of such constraints was never considered, and this promising avenue of study 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.

4.3 Reiter's Theory of Database Evolution

Although the "main stream" of AI treated 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, as described in Section 2.3. Also, unlike the original situation calculus, which was entirely first-order [McC63,MH69], Reiter's development includes an induction

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

axiom specified in second-order logic, for reasoning about sequences of transactions. Applying this approach, Reiter has developed a logical theory of database evolution [Rei95].

From the perspective of database theory [AHV95,Ull88], Reiter's approach 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, can turn into 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:[11]

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

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], briefly described earlier in this survey (Section 2.3). Indeed, 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.

4.4 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

[11] 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.

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, Golog does not include classical logic programming as a special case. That is, classical logic programs and deductive databases are *not* Golog 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 eschews. 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 characteristics that should also be mentioned. 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 many 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.

5 Conclusion

In this paper, we reviewed a number of approaches to state changes in databases and logic programming. Our thesis is that they can all be traced, to various degree, to three influential ideas proposed over twenty years ago: the situation calculus, Prolog, and Dynamic Logic. We have therefore tried to classify the works surveyed here into one or more of these three categories. In the process, we compared the approaches with each other and highlighted their differences.

Of course, there is a vast literature on time, change and action that could not be addressed in this short survey. In particular, we have excluded works based on various temporal logics, since they have had minimal influence on databases and logic programming. (Some discussion of these works appears in [BK95].) Likewise, we did not discuss proposals based on Linear Logic (such as [AP91]) or Rewriting Logic (*e.g.*, [Mes92a,Mes92b]). Nevertheless, some comparison of these approaches with Concurrent Transaction Logic can be found in [BK96].

Acknowledgments. The presentation in this paper has been greatly improved due to the referee comments. The first author was partially supported by a research grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). The second author was partially supported by the NSF grant IRI-9404629.

References

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

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

[AP91] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(4):445–473, 1991.

[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.

[BK93] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[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.

[BK98a] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[BK98b] A.J. Bonner and M. Kifer. Results on reasoning about updates in transaction logic. This volume, 1998.

[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.

[Bon97a] A.J. Bonner. The power of cooperating transactions. Manuscript, 1997.

[Bon97b] 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.

[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.

[Che95] W. Chen. Declarative updates of relational databases. *ACM Transactions on Database Systems*, 20(1):42–70, March 1995.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[DKRR98] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, June 1998.

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

[Gre69] C.C. Green. Application of theorem proving to problem solving. In *Intl. Joint Conference on Artificial Intelligence*, pages 219–240, 1969.

[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.

[Kow79] R. Kowalski. *Logic for Problem Solving.* North-Holland, Amsterdam, The Netherlands, 1979.

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

[KS86] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS97] R. Kowalski and F. Sadri. Reconciling the event calculus and the situation calculus. *Journal of Logic Programming*, 31:39–58, 1997.

[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.

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. In D. Pedreschi and C. Zaniolo, editors, *Intl. Workshop on Logic in Databases*, volume 1154 of *Lecture Notes in Computer Science*, pages 196–222. Springer-Verlag, San Miniato, Italy, July 1996.

[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*, 31:59–84, 1997.

[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.

[Mes92a] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes92b] J. Meseguer. Multiparadigm logic programming. In *Algebraic and Logic Specifications*, number 632 in Lecture Notes in Computer Science, pages 158–200. Springer-Verlag, September 1992.

[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.

[Pnu77] A. Pnueli. A temporal logic of programs. In *Intl. Conference on Foundations of Computer Science*, pages 46–57, October 1977.

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

[Pra90] V.R. Pratt. Action logic and pure induction. In *Workshop on Logics in AI*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120. Springer-Verlag, 1990.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modeling: 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.

[Rei92a] R. Reiter. Formalizing database evolution in the situation calculus. In *Conference on Fifth Generation Computer Systems*, 1992.

[Rei92b] R. Reiter. On formalizing database updates: Preliminary report. In *Proceedings of the Third Intl. Conference on Extending Database Technology*, pages 10–20. Springer-Verlag, March 1992.

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

[SK95] F. Sadri and R. Kowalski. Variants of the event calculus. In L. Sterling, editor, *Intl. Conference on Logic Programming*. MIT Press, 1995.

[Spr94] P.A. Spruit. *Logics of Database Updates*. PhD thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1994.

[SWM93] P.A. Spruit, R.J. Wieringa, and J.-J.Ch. Meyer. Dynamic database logic: The first-order case. In U.W. Lipeck and B. Thalheim, editors, *Modeling Database Dynamics*, pages 103–120. Springer-Verlag, 1993.

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

[vB91] J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic*. Elsevier Science Pub. Co., Amsterdam, New York, 1991.

[WF97] C.A. Wichert and B. Freitag. Capturing database dynamics by deferred updates. In *Intl. Conference on Logic Programming*, pages 226–240, June 1997.

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