Transaction Logic: Unifying Declarative and Procedural Knowledge – Extended Abstract –

Anthony J. Bonner* and **Michael Kifer†**

1 Introduction

This paper presents AI applications of the recently proposed *Transaction Logic* (abbr., $\mathcal{TR}$) [2]. Transaction Logic is a novel formalism that accounts in a clean and declarative fashion for the phenomenon of updating first-order knowledge bases, most notably, databases and logic programs. Transaction Logic has a natural model theory and a sound-and-complete proof theory. Unlike many other logics, $\mathcal{TR}$ allows users to *program* transactions that modify the state of a knowledge base. This is possible because, like classical logic, $\mathcal{TR}$ has a "Horn" version which has *both* a procedural and a declarative semantics, as well as an efficient SLD-style proof procedure. As a result, $\mathcal{TR}$ is a unifying, *logical* formalism for specifying both *declarative* and *procedural* knowledge. Furthermore, for a wide range of practical problems, the *frame problem* [18] is not an issue for $\mathcal{TR}$. This is because $\mathcal{TR}$ performs real updates on materialized databases, much as procedural languages like Pascal do. A key contribution of $\mathcal{TR}$ is capturing these procedural updates in a logical framework with an efficient proof theory. A full development of the proof theory, a discussion of the frame problem, and applications to database systems can be found in [2, 3]. This paper presents the model theory of $\mathcal{TR}$, and then focuses on applications of $\mathcal{TR}$ to problems in AI, especially planning, temporal reasoning, constraint satisfaction, hypothetical and counterfactual reasoning, and the representation and use of procedural knowledge.

The importance of procedural knowledge has been extensively argued in the AI literature (see *e.g.*, [8]). For instance, the well-known SHRDLU program [26] is largely based on procedural knowledge. In fact, Winograd argues in [26] that procedural knowledge is inherent in automated natural language understanding. For example, the meaning of "the" is a collection of procedures that check the context and then prescribe different sequences of actions, depending on the outcome [26]. Likewise, the classic planning system STRIPS [6] is based on procedurally-defined actions, which STRIPS combines into plans that achieve larger goals. Section 6.1 shows that STRIPS is representable in $\mathcal{TR}$ and that its inference rules are sound. Even though STRIPS was given formal semantics in [16], this was not done within a *logical* formalism (and, unlike [16], STRIPS is just one of the many applications of $\mathcal{TR}$).

At first glance, there might seem to be many candidates for a logic of procedural knowledge, since many logics reason about updates or about the related phenomena of time and action. We have found, however, that none of them is suitable for representing and using procedural knowledge. First, most logics of time or action are *hypothetical*: Instead of executing procedures, they reason *about* them, or about what *would* happen *if* certain actions were to take place. For instance, some systems can infer that if action A precedes B, and B precedes C, then A must precede C. Others can infer that if a student took history 400, then he could graduate. Such systems were intended to be observers of action, not participants. They are therefore useful for reasoning about alternatives, or for analyzing programs and plans; but they are not very useful for defining procedures that actually *accomplish* state changes being reasoned about. In $\mathcal{TR}$, actions can be carried out hypothetically or they can be executed and have a permanent effect on the knowledge base, depending on one's desire. In this way procedural knowledge can be *used* as well as represented and reasoned about.

The second problem with many action logics is that it is awkward, if not impossible to assign names to composite actions. Such logics were designed for reasoning about sequences of actions, not for programming them. As such, they are inappropriate for defining actions since a naming facility is needed for representing even very basic procedural knowledge; *i.e.*, specifying actions without a naming facility is akin to programming without subroutines. This defeats the purpose of using logic in the first place, which is to free the user from the drudgery of low-level details.

Third, many logics make a clear distinction between queries and updates. However, this distinction is blurred in object-oriented systems, where both queries and up-

*Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A4, Canada; bonner@db.toronto.edu. Work supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada and by a Connaught Grant from the University of Toronto.

†Department of Computer Science, SUNY–Stony Brook, Stony Brook, NY 11794, U.S.A.; kifer@cs.sunysb.edu. Work supported in part by NSF grant CCR-9102159.

dates are special cases of a single idea: *method invocation*. In such systems, an update can be thought of as a query with side effects. We would like to model this behavior and thereby provide a logical foundation for object-oriented databases. $\mathcal{TR}$ achieves this by allowing every logical formula to have not only a truth value, but also a "side effect" on the database. In this way, one can account for the *behavior* of object-oriented databases—something that most formalisms do not do. In combination with F-logic [13], the structural aspects of object-oriented systems can be accounted for as well.

The system that comes closest in spirit to $\mathcal{TR}$ is Prolog. Unfortunately, updates in Prolog are non-logical and, as a result, state-changing procedures are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain. Semantically $\mathcal{TR}$ is closely related to Process Logic [11], but is different from it in several important ways detailed in [2].

Due to space limitation, many topics are merely sketched in this paper. Details appear in [2].

2 Syntax

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

Transaction formulas, which extend first-order formulas with a new connective, $\otimes$, called *serial conjunction*, are used to define transactions and formulate queries. Transaction formulas are defined as follows. An *atomic* transaction formula is an expression of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol, and $t_1, \ldots, t_n$ are terms (as in classical predicate calculus). If ϕ and ψ are transaction formulas, then so are $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \otimes \psi$, $\neg\phi$, $(\forall X)\phi$, and $(\exists X)\phi$, where X is a variable. Thus, the expression $a(X) \vee \neg[b(X) \otimes c(X,Y)]$ is a transaction formula. Intuitively, $\psi \otimes \phi$ means, "Do ψ, then do ϕ." A dual connective, *serial disjunction*, is also useful (Section 6.2): $\psi \oplus \phi$ is equivalent to $\neg(\neg\phi \otimes \neg\psi)$.

Serial conjunction provides a basic way to *sequence* transactions, where $\phi \otimes \psi$ means "do ϕ, then do ψ." In contrast, classical conjunction, "$\wedge$", constrains the non-determinism of a transaction. For instance, $\phi \wedge \psi$ means, "do ϕ in a way compatible with doing ψ." This use of "$\wedge$" is further discussed in Section 6.2. Apart from this, "$\wedge$" also has the traditional role of forming logic programs: in $\mathcal{TR}$, as in classical logic, any finite set of rules is equivalent to a conjunction of all the rules in the set. In $\mathcal{TR}$, such a set of transaction formulas is called a *transaction base*.

A transaction base defines complex transactions in terms of simpler ones. However, we also need a way to specify *elementary changes* to the underlying database. One way to define such transitions is to build them into the semantics as in, say, [17, 4, 19]. A problem with this approach is that adding new kinds of elementary transitions leads to a redefinition of the very notion of a model and thus to a revamping of the entire theory, including the need to reprove soundness and completeness results. This is a rather serious drawback since there appears to be no small, single set of elementary transitions that is best for all purposes [2]. Thus, rather than commit $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat the elementary transitions as a *parameter* of $\mathcal{TR}$. Each set of elementary transitions thus gives rise to a different version of the logic. To achieve this, elementary transitions are defined by logical axioms.

Elementary transitions are formulas of the form $\langle \phi, \psi \rangle u$, where ϕ, ψ are (sets of) closed first-order formulas and u is an atomic formula, called the *name* of the transition. Intuitively, this formula says that u is an update that transforms database ϕ into database ψ. For instance, if the atoms $ins{:}q(t)$ and $del{:}q(t)$ stand for the insertion and deletion of the atom $q(t)$, then they would be defined by an enumerable set of elementary transitions consisting of the following formulas:

$$\langle \mathbf{D}, \mathbf{D} + \{q(t)\} \rangle\ ins{:}q(t) \quad \langle \mathbf{D}, \mathbf{D} - \{q(t)\} \rangle\ del{:}q(t)$$

for every relational database, $\mathbf{D}$.[1] Enumerable sets of elementary transitions are called *transition bases*. In practice, these formulas would not be materialized all at once, but would be generated on demand by an algorithm. We refer to [2] for a more detailed discussion of transition bases.

As seen from the syntax, $\mathcal{TR}$ does not strictly distinguish between predicates that query the knowledge base and predicates that update it. As in classical logic, every predicate has a truth value; but in addition, every predicate may also have a side effect, by changing the state of the knowledge base. This uniformity of representation is important in modeling *methods* (interface functions encapsulated inside classes) in object-oriented databases, where one generally does not distinguish between information-retrieving and state-changing methods. Nevertheless, if desired, $\mathcal{TR}$ can make such a distinction by using different sorts of predicates, one for updates and one for queries.

For instance, it is good programming practice to reserve a special set of predicates for certain basic updates. This paper uses just such a convention: for each predicate symbol p, we use another predicate symbol, $ins{:}p$, to represent insertions of tuples into p. Likewise, we represent deletions from p by the predicate $del{:}p$. Thus the formula $ins{:}p(a) \otimes ins{:}p(b) \otimes ins{:}p(c)$ represents an updating transaction that inserts $p(a)$ into the database, then $p(b)$, and then $p(c)$.

3 Blocks-World Example

Before presenting the semantics, we illustrate the syntax through an example of a robot arm moving blocks around a table top [20]. This example is is used extensively throughout the paper. It illustrates the use of transaction subroutines, and it shows how $\mathcal{TR}$ improves upon Prolog's *assert* and *retract* operators.

Example 3.1 (Non-Deterministic Robot Actions) A state in our blocks world is defined in terms of three database predicates: $on(x,y)$, which says that block x is on top of block y; $clear(x)$, which says that nothing is on top of block x; and $color(x,c)$, which says that c is

[1] If $\mathbf{D}$ is a general first-order formula, then defining insertion and deletion is more involved [12].

the color of block x. The rules below define six actions that change the state of the world. Together, these rules form a transaction base. For each action, the premises are evaluated in the order given, and the action fails if any of its premises fails (in which case the database is left in its original state). All variables are assumed to be universally quantified outside the rules.

$$stackSameColor(Z) \leftarrow color(Z,C) \otimes stack2colors(C,C,Z)$$
$$stack2colors(C_1,C_2,Z) \leftarrow color(X,C_1) \otimes color(Y,C_2) \otimes stack2blocks(X,Y,Z)$$
$$stack2blocks(X,Y,Z) \leftarrow move(Y,Z) \otimes move(X,Y)$$
$$move(X,Y) \leftarrow pickup(X) \otimes putdown(X,Y)$$
$$pickup(X) \leftarrow clear(X) \otimes on(X,Y) \otimes del{:}on(X,Y) \otimes ins{:}clear(Y)$$
$$putdown(X,Y) \leftarrow wider(Y,X) \otimes clear(Y) \otimes ins{:}on(X,Y) \otimes del{:}clear(Y)$$

The basic actions are $pickup(X)$ and $putdown(X,Y)$, meaning, "pick up block X," and "put down block X on top of block Y," respectively. Both are defined in terms of elementary inserts and deletes to database relations. The remaining rules combine simple actions into more complex ones. For instance, $move(X,Y)$ means, "move block X to the top of block Y," and $stack2blocks(X,Y,Z)$ means, "stack blocks X and Y on top of block Z." These actions are deterministic: Each set of argument bindings specifies only one robot action.

In contrast, the two actions *stack2colors* and *stackSameColor* are *non-deterministic*. For instance, $stack2colors(C1,C2,Z)$ means, "stack two blocks, of colors C_1 and C_2, on top of block Z." The action does not say which two blocks to use, only their colors. To perform the action, the inference system searches the database for blocks of the appropriate color that can be stacked. If several such blocks are available, the system chooses any two arbitrarily. The action $stackSameColor(Z)$ means, "stack two blocks on top of Z that are of the same color as Z." Again, the inference system searches the database for appropriate blocks. In this way, by defining non-deterministic actions, a user can specify what to do (declarative knowledge) and how to do it (procedural knowledge). □

Note that the rules in Example 3.1 involve queries as well as updates. In the last rule, for instance, the atom $clear(Y)$ (which itself may be defined by other deductive rules) is a Boolean test that must return *true* in order for the transaction to succeed. In the first rule, the atom $color(Z,C)$ is a query that retrieves the color C of the block Z. The second rule is, perhaps, the most interesting. Here, the atoms $color(X,C_1)$ and $color(Y,C_2)$ are non-deterministic queries. They retrieve two blocks X and Y, of colors C_1 and C_2, respectively. The particular blocks retrieved by these queries then determine the *future* course of action taken in the rest of the transaction.

Example 3.1 easily extends to recursively defined actions. For instance, $stack(N,X)$ can be defined as an action that recursively stacks N blocks on top of block X [2].

Finally, observe that the rules in Example 3.1 can easily be rewritten in Prolog form, by replacing "$\otimes$" with "," and by replacing the elementary state transitions with *assert* and *retract*. However, the resulting, apparently innocuous, Prolog program does not execute correctly! The problem is that Prolog updates are not undone during backtracking. For instance, suppose that during a *move* action, the robot picked up $blkA$, the widest block on the table. The *move* action would then fail, since the robot cannot put $blkA$ down on the stack, since $blkA$ is too wide. In $\mathcal{TR}$, the inference system simply backtracks and then tries to find another block to pick up. Prolog, too, will backtrack, but it will leave the database in an incorrect state, since it will not undo the *pickup* action. Thus, if $blkA$ was previously on top of $blkB$, then $on(blkA,blkB)$ would remain deleted and $clear(blkB)$ would stay in the database.

4 Semantics

Just as the syntax of $\mathcal{TR}$ is based on two basic ideas—serial conjunction and elementary transitions—the semantics is also based on a few fundamental principles:

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

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

To model transactions, we start with a modal-like semantics, where each state represents a database, and each elementary update causes a transition from one state to another, thereby changing the database. At this point, however, modal logic and Transaction Logic begin to part company. The first major difference is that truth in $\mathcal{TR}$ structures does not hinge on a set of arcs between states. Instead, we focus on *paths*, that is, on sequences of states. (This focus on paths is related to the version of Process Logic in [11], but the two logics are fundamentally different [2].) Because of the emphasis on paths, we refer to semantic structures in $\mathcal{TR}$ as *path structures*. Second, truth in path structures is defined on paths, not states. For example, we would say that the path $\mathbf{D}$, $\mathbf{D}+\{a\}$, $\mathbf{D}+\{a,b\}$ satisfies the formula $ins{:}a \otimes ins{:}b$, since it represents an insertion of a followed by an insertion of b. A path of length 1 corresponds to a single database state. In this way, one model-theoretic device, paths, accounts for databases, updates, queries and more general transactions.

Database States: Another difference between modal logic and Transaction Logic is in the nature of states.

In modal logic, a state is basically a first-order semantic structure, since each state specifies the truth of a set of ground atomic formulas. Such structures are adequate for representing relational databases, but not for representing more general theories, like indefinite databases or general logic programs. We therefore take a more general approach. Since a database is a first-order formula, which has a *set* of first-order models, we define a state to be a *set* of first-order semantic structures. Each state, s, thus corresponds to a particular database—the database having precisely the models in s.

This approach to states provides a lot of flexibility when defining elementary updates. Such flexibility is needed since, for general databases, the semantics of elementary updates is not clear, not even for relatively simple updates like insert and delete. For example, what does it mean to insert an atom b into a database that entails $\neg b$, especially if $\neg b$ itself is not explicitly present in the database? There is no consensus on the answer to this question, and many solutions have been proposed (see [12] for a comprehensive discussion). For these reasons, we take a general approach to elementary updates. For us, an elementary update is a mapping that takes each database $\mathbf{D}_1$ to some other database $\mathbf{D}_2$, where a database is any first-order formula. More generally, an elementary update may be non-deterministic, so it is not just a mapping, but a *binary relation* on databases.

4.1 Path Structures

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

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

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

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

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

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

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

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

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

1. $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ iff $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, where "$\models^c_\nu$" denotes classical satisfaction in first-order logic, and $p(t_1, \ldots, t_n)$ is an atomic formula.
2. $\mathbf{M}, \pi \models_\nu \neg\phi$ iff $\mathbf{M}, \pi \not\models_\nu \phi$.
3. $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ iff $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$. As usual, "$\wedge$" is dual to "$\vee$".
4. $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ iff $\mathbf{M}, \gamma \models_\nu \phi$ and $\mathbf{M}, \delta \models_\nu \psi$ for *some* split $\gamma \circ \delta$ of path π. *Serial disjunction*, $\oplus$, is the dual of $\otimes$: $\phi \oplus \psi \equiv \neg(\neg\phi \otimes \neg\psi)$.
5. $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ iff $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X. The meaning of $(\exists X)\phi$ is dual to that of $(\forall X)\phi$.

As usual, variable assignments can be omitted for closed formulas. From now on, we shall deal only with such formulas, unless stated otherwise. □

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

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

$$\begin{array}{lcl} (\phi \vee \psi) \otimes \eta & \leftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\ (\phi \wedge \psi) \oplus \eta & \leftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\ (\phi \wedge \psi) \otimes \eta & \longrightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta) \\ (\phi \vee \psi) \oplus \eta & \longleftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta) \end{array} \qquad (1)$$

Definition 4.3 tells us what it means for a path structure to be a model of a transaction formula ϕ. Such formulas are used to define complex transactions in terms

of simpler ones. In addition, we must define what it means to be a model of an elementary state transition, $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$. Intuitively, this formula means that u is an update that changes database $\mathbf{D}_1$ into database $\mathbf{D}_2$. The next two definitions make this idea precise.

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

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

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

4.2 Execution as Entailment

We now define *executional entailment*, a concept that connects the model theory with transaction execution. Intuitively, execution of a transaction formula corresponds to truth on a path. In $\mathcal{TR}$, a program consists of three distinct parts: a transaction base $\mathbf{P}$, a database $\mathbf{D}$, and a transition base $\mathcal{B}$. Each of these parts plays a distinct role in defining executional entailment. Only the database is updatable. The other two parts specify transactions that update the database and/or answer queries. The transition base defines elementary updates, and the transaction base contains logical rules that define complex queries and transactions. It will normally be composed of formulas containing the serial connectives $\otimes$ or $\oplus$, though classical first-order formulas are also allowed. In contrast, the database consists entirely of classical first-order formulas.

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

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \qquad (2)$$

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

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \qquad (3)$$

which is true iff there is a sequence of databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$ that makes Statement (2) true. □

Intuitively, Statement (2) means that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1$ ... to $\mathbf{D}_n$. Formally, it means that every model of $\mathcal{B}$ and $\mathbf{P}$ has a path corresponding to $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ that satisfies formula ϕ.

5 Proof Theory

$\mathcal{TR}$ has a sound and complete proof theory. Furthermore, there is a Horn-like subset of $\mathcal{TR}$, called *serial-Horn*, that includes classical logic programs and that has an efficient SLD-style proof procedure, *i.e.*, a Prolog-style procedure based on unification. All rules in Example 3.1 are serial-Horn. The proof theory for serial Horn rules consists of several inference rules whose goal is to establish statements of the form (3). The main feature of this theory is that constructing certain kinds of proofs is tantamount to *executing* transactions and extracting their execution paths.

The proof theory, which is developed in detail in [2], cannot be given adequate coverage here. Instead, we present the following properties of executional entailment, which can form the basis of a "naive" proof theory. Note that the first three properties correspond to inference rules, and the last two correspond to axioms.[2]

- $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ iff $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.
- If $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.
- If $\alpha \leftarrow \beta$ is in $\mathbf{P}$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.
- If $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$ is in $\mathcal{B}$, then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.
- If $\mathbf{D}_0 \models^c \psi$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \psi$, where ψ is a 1st-order formula, and $\models^c$ is classical entailment.

6 Applications

A wide variety of interesting and useful formulas can be constructed in $\mathcal{TR}$, formulas that capture many of the novel and important features of database and knowledge-base systems. These features include action definition and execution, rule-based inference, consistency maintenance, hierarchical and script-based planning, static and dynamic constraints, non-deterministic updates, random sampling, view updates, and more. This section describes some of these applications, focussing on those that are most relevant to AI. Additional applications and further details of the applications presented herein can be found in [2]. We shall also see (in the next section) that the semantics of $\mathcal{TR}$ allows the easy introduction of a modal necessity operator, □, which captures a whole new range of applications. These applications include hypothetical reasoning, subjunctive queries, counterfactuals, imperative programming constructs, active databases, software verification, and more. $\mathcal{TR}$, thus, provides a wide range of features whose amalgamation in a single declarative formalism has proved elusive in the past. Furthermore, these features all follow naturally from $\mathcal{TR}$'s path-based semantics.

6.1 Planning

Planning of robot actions is carried out by representing various planning regimes as formulas in $\mathcal{TR}$. This is possible because $\mathcal{TR}$-transaction bases can be used to

[2] Here, $\mathcal{B}$, $\mathbf{P}$, and $\mathbf{D}_0, \ldots, \mathbf{D}_n$ are as in Definition 4.6, and α and β are transaction formulas.

represent two types of knowledge about actions: action-definitions, which describe how to execute actions; and planning strategies, which describe how and when to execute actions. Unlike many planning systems, $\mathcal{TR}$ does not need special mechanisms or special syntax to deal with these two types of knowledge. In $\mathcal{TR}$, both types of knowledge can be represented as serial Horn rules. Planning itself is carried out by executing plan-generating transactions using the proof theory of [2].

In [2], we consider three planning regimes. In the first regime, called *naive* planning, the inference system searches blindly for any sequence of actions that achieves the planning goal. Naive planning corresponds roughly to planning using a forward production system as described in [20]. The naive planner is simple to formulate, but is very inefficient; its utility is limited to a mere demonstration that planning is possible in $\mathcal{TR}$.

The other two regimes improve upon the naive system by incorporating knowledge about how goals are to be achieved. In the second regime, planning is described in terms of a hierarchy of "scripts" or "skeletal plans." Scripts suggest ways of achieving goals, and often they summarize known methods in a particular problem domain. This kind of planning, which is exemplified by systems such as NOAH [24] and MOLGEN [7, 25], is natural for $\mathcal{TR}$, since each script corresponds to a high-level, non-deterministic action. For example, a script for making coffee might be as follows: grind coffee, boil water, put coffee in filter, pour water into filter [5, Article XV.D1]. In $\mathcal{TR}$, this script could be represented by a rule like the following:

$$\begin{array}{lcl} makeCoffee & \leftarrow & grindCoffee \otimes boilWater \otimes \\ & & fillFilter \otimes pourWater \end{array} \qquad (4)$$

Scripts and skeletal plans often have the form of a procedure (or recipe), and they are often specified in a procedural language. However, as (4) above illustrates, they can also be specified *in a logic*, $\mathcal{TR}$, that integrates procedural and declarative knowledge in a single framework.

The third planning regime is exemplified by the well-known STRIPS system [6, 16], which plans movements for a robot arm. STRIPS is naturally representable in $\mathcal{TR}$ and its planning strategy is sound, albeit incomplete, in our semantics. This incompleteness is responsible for certain failures of STRIPS, *e.g.*, its inability to exchange the contents of two registers [20]. Since $\mathcal{TR}$ has a *complete* proof theory [2], we thus obtain a more powerful planning system. Other problems with STRIPS-like planning (which extend to many other planning systems) are pointed out and solved in [2].

All examples in this paper are based on the insertion and deletion of tuples from a database. They thus bear a conceptual resemblance to STRIPS-actions. Several differences are worth noting:

- Unlike STRIPS-actions, $\mathcal{TR}$-rules are formulas in a rigorous *logical* formalism. They are therefore declarative as well as executable.
- Rules in $\mathcal{TR}$ are *hierarchical* and can be defined at many levels of abstraction. The six rules in Example 3.1, for instance, represent six different levels of abstraction. In contrast, STRIPS only allows actions to be defined at one level, directly in terms of database inserts and deletes; *i.e.*, it does not support intermediate-level actions (subroutines).
- Unlike STRIPS, actions in $\mathcal{TR}$ can be *non-deterministic*. As Example 3.1 shows, non-determinism can make actions simpler and easier to formulate, reducing the amount of detail that a user must specify.
- STRIPS-actions are relatively simple; they consist of a pre-condition (which is a series of tests), followed by a set of deletes, followed by a set of inserts, in that order. In contrast, in $\mathcal{TR}$, inserts, deletes, and tests can be sequenced in any order. Thus, apart from pre-conditions and post-conditions, any number of intermediate tests can also be specified. In fact, even more general actions are possible in $\mathcal{TR}$, since in addition to sequential ordering, formulas may be combined via classical conjunction, disjunction and negation.
- Finally, lest this is forgotten, $\mathcal{TR}$ is a general-purpose logic for which STRIPS, planning, and other issues considered here and in [2] are only some of the many applications.

6.2 Constraints on Transaction Execution

Because transactions are defined on paths, it is possible to express a large variety of constraints on the way they execute. For instance, we can place conditions on the state of the database during transaction execution, or we may forbid certain sequences of states. We refer to such conditions as *path constraints*, or *dynamic constraints*. Such constraints are particularly well suited to areas such as planning and design, where it is common to place constraints on the way things are done. This section illustrates a variety of dynamic constraints expressible in $\mathcal{TR}$. These include temporal constraints in the style of James Allen [1], such as, "immediately after," "some time after," "before," etc.

There are several important problems related to constraints. One such problem is *constraint satisfaction*. That is, given a transaction and a constraint, we want to execute the transaction in such a way that it satisfies the constraint. For example, we might ask a robot to carry out a task while not entering restricted areas and not executing certain undesirable or dangerous sequences of action. In general, starting from the current database, we want to find *some* way of executing a transaction while satisfying constraints.

Constraint satisfaction problems are particularly easy to express in $\mathcal{TR}$ because they correspond to classical conjunction. That is, if ψ and ϕ are transaction formulas, then $\psi \wedge \phi$ means, "Do transaction ψ so that ϕ will be satisfied on the execution path." Intuitively, if ψ is a non-deterministic transaction, then ϕ acts as a filter, removing unwanted execution paths, and reducing the non-determinism of the transaction. If ψ is deterministic, then ϕ acts as a guard, forbidding execution unless the constraints are satisfied. Note that in either case, it is execution *paths* that are constrained.

Two types of path constraint naturally arise in $\mathcal{TR}$: those based on serial conjunction, and those based on serial implication. The former specify that something must be true *somewhere* on a path, and the latter specify that something must be true *everywhere* on a path. These two types of path constraint correspond roughly to two types of database integrity constraint: those based on existential quantification, and those based on universal quantification, respectively. We give examples of both.

The examples center around a planning system for robot navigation. The system is composed of rules defining an action, $goto(Y)$, that instructs the robot to go to location Y. This action is highly non-deterministic since there may be many routes that the robot can take. Dynamic constraints can force the planner to reject certain routes or to focus its attention on others.

6.2.1 Constraints Based on "$\otimes$"

A simple constraint might require the robot to do something while en route to some location, such as passing certain check points. There are two natural cases to this problem. In the first case, the constraint imposes an order on the way the robot does things. The constrained transaction can then be expressed as a sequence of goals. For instance, suppose we request the robot to go to room A in a building and to pass through room B along the way. This request can be expressed as the serial goal $goto(roomB) \otimes goto(roomA)$, *i.e.*, "go to room B and then go to room A."

In the second, and more interesting case, the constraint does not imply an order on the way things are done. For instance, suppose we request the robot to go to room A, passing through rooms A_1 and A_2 on the way. This does not commit the robot to visiting the rooms in a particular order, and so it cannot be expressed as a single, serial goal. Instead, it is properly expressed as a *conjunction* of serial goals, where each conjunct constrains the robot to pass through a particular room.

To express such constraints, we define a new proposition, **path**, which is true on every path, *i.e.*, $\mathbf{path} \equiv \phi \vee \neg\phi$, for any formula ϕ. The formula $\mathbf{path} \otimes at(X)$, thus, specifies a path in which the robot ends up at location X. Likewise, the formula $at(X) \otimes \mathbf{path}$ specifies a path in which the robot starts off at location X, while $\mathbf{path} \otimes at(X) \otimes \mathbf{path}$ specifies a path in which the robot passes through location X. For convenience, we abbreviate this latter formula as $go\_thru(X)$, by adding the following rule to the transaction base:

$$go\_thru(X) \leftarrow \mathbf{path} \otimes at(X) \otimes \mathbf{path}$$

It is now easy to specify paths in which the robot must pass through any number of locations, without specifying an order. For instance, the following formula specifies that the robot must go to room A, passing through rooms A_1 and A_2 along the way:

$$goto(roomA) \wedge go\_thru(roomA_1) \wedge go\_thru(roomA_2)$$

The use of classical conjunction ensures that this formula is true only on paths where all four conjuncts are true. In this way, the formulas $go\_thru(X)$ constrain the way in which the transaction $goto(roomA)$ may execute.

We can build up more complex constraints by combining classical and serial conjunction. For instance, the following formula requests that the robot go to room A and that along the way it pass first through rooms B_1 and B_2, in any order, and then through rooms C_1 and C_2, in any order:

$$goto(roomA) \wedge [go\_thru(roomB_1) \wedge go\_thru(roomB_2)] \otimes [go\_thru(roomC_1) \wedge go\_thru(roomC_2)]$$

6.2.2 Constraints Based on "$\Rightarrow$"

Constraints based on serial implications, "$\Leftarrow$" and "$\Rightarrow$" introduced in Section 4.1, can constrain a transaction during every moment of its execution. For instance, we can request a robot to remain inside a particular region while executing a task. We can also put constraints on specific actions that the robot might take. For instance, we might request a robot to perform a series of actions subject to the following constraints:

(i) Open doors before passing through them.

(ii) Shut doors after passing through them.

(iii) Before leaving a room, turn off all the lights.

(iv) After entering a room, turn on all the lights.

(v) Unlock the rifle before firing it.

(vi) Lock and reload the rifle after firing it.

Due to space limitation, serial-conjunction-based constraints can be only sketched in this paper. Here we only remark that $\mathcal{TR}$ can express many other temporal relations in the style of James Allen's theory of time intervals [1]. These relations include "some time before," "some time after," "immediately before," etc. For instance, $\beta \Leftarrow \alpha$ says that β occurs *immediately* before α, while $\alpha \Rightarrow (\mathbf{path} \otimes \beta)$ expresses the fact that β occurs *some* time after α. Details appear in [2].

7 Hypothetical Reasoning

Hypothetical queries play an important role in reasoning about knowledge. Because of such queries, it is often necessary to perform hypothetical updates as well as actual ones. For instance, a game playing program may reason as follows: After a given series of actions, α, does the opponent's situation improve? Observe that the actions mentioned in this query are purely hypothetical and are *not* committed. If the answer to the query is "no," then the program would perform action α, at which point the action *is* committed. Otherwise, the program would do further depth analysis and perform the most favourable move that it finds. By distinguishing between real and hypothetical actions, this program combines reasoning about action (planning, exploration of alternatives, etc) with actual execution of actions (committing itself to a particular course of action). $\mathcal{TR}$ is the only logic we are aware of that can do *both* these things. This section first describes the semantics of hypotheticals in $\mathcal{TR}$, and then focuses on applications of hypotheticals to knowledge-based systems. Other applications of hypotheticals as well as a sound-and-complete proof theory for them are developed in [2].

To represent hypothetical actions, we extend the syntax of $\mathcal{TR}$. Formally, a *hypothetical formula* is an expression of the form $\Diamond\phi$ or $\Box\phi$, where ϕ is a transaction formula or a hypothetical transaction formula. Hypothetical operators can thus be nested. In modal terms, $\Diamond\phi$ means that the execution of ϕ is *possible* starting at the present state, and $\Box\phi$ means that the execution of ϕ is *necessary* at the present state. Necessity means that ϕ is executable along *every* path leaving the current state, $\mathbf{D}$, including $\langle\mathbf{D}\rangle$, the unique path of length 1 that leaves (and enters) $\mathbf{D}$; possibility means that ϕ is executable along *some* path leaving the current state.

To define the meaning of hypotheticals formally, let $\mathbf{M} = \langle U, \mathcal{F}, N, I_{path}\rangle$ be a path structure, s be a state in N, and let ν be a variable assignment. Then

- $\mathbf{M}, \langle s\rangle \models_\nu \Diamond\phi$ if and only if there is a path, π, starting at state s such that $\mathbf{M}, \pi \models_\nu \phi$ holds.
- $\mathbf{M}, \langle s\rangle \models_\nu \Box\phi$ if and only if for every path, π, starting at state s, it is the case that $\mathbf{M}, \pi \models_\nu \phi$.

Note that hypotheticals hold over paths of length 1, and so they do not cause any real state transitions. We remark that "$\Diamond$" and "$\Box$" are not exactly dual to each other. Dual versions of these modal operators can also be defined [2], but we do not discuss them here. Instead, we explore two non-trivial applications of hypothetical operators: subjunctive statements and imperative programming constructs.

7.1 Subjunctive Queries and Counterfactuals

Subjunctive queries [9] are statements of the form "if ϕ were true, ψ would have been true/possible as well." When ϕ is actually *false* in the present state, the subjunctive query is called a *counterfactual* [15].

The meaning of a subjunctive query is as follows [9]: Update the current knowledge-base with ϕ; if the resulting database satisfies ψ, then the subjunctive query is true; otherwise, it is false. Following Katsuno and Mendelzon [12], it is widely accepted that "updating" a knowledge-base, $\mathbf{D}$, means finding a knowledge-base that is closest to $\mathbf{D}$ according to some metric.[3]

The following are the two major classes of subjunctive queries together with their formulation in $\mathcal{TR}$. To be precise in our formulations, we use the notation of executional entailment.

- *At state* $\mathbf{D}$, *if* ϕ *were true, then* ψ *would be true too*: $\mathcal{B}, P, \mathbf{D} \models \Box(ins{:}\phi \Rightarrow \psi)$
- *At state* $\mathbf{D}$, *if* ϕ *were true, then* ψ *would be possible*: $\mathcal{B}, P, \mathbf{D} \models \Box(ins{:}\phi \Rightarrow \Diamond\psi)$

Note the role of the necessity operator "$\Box$" in the above queries. This operator ensures that the consequent formula (ψ or $\Diamond\psi$) is true in *every* state obtained from $\mathbf{D}$ by inserting ϕ.[4] Also note that we used serial implication, "$\Rightarrow$", instead of classical implication, "$\rightarrow$". This requires the consequents to hold *after* the insertion of ϕ. (Classical implication would require them to hold *during* the insertion.)

7.2 Imperative Programming Constructs

Perhaps, one of the most interesting bonuses provided by of the hypothetical operators in $\mathcal{TR}$ is the ability to express standard imperative constructs, such as **if-then-else** and **while-do** in a simple declarative way. For instance, the following rules express an **if-then-else** statement:

$$\begin{array}{l} p \longleftarrow (\Diamond a) \otimes b \\ p \longleftarrow (\Box\neg a) \otimes c \end{array} \tag{5}$$

The query $?-p$ here means, "if it is possible to do a, then do b, else do c." Because a can be an action that changes the state of the knowledge-base, the use of the hypotheticals is crucial to the proper formulation of this imperative statement. Furthermore, the negation in "$\Box\neg a$" is of the negation-by-failure variety. In [2], we present a semantics for this kind of negation.

The following rules express a **while-do** statement:

$$\begin{array}{l} q \longleftarrow (\Diamond a) \otimes b \otimes q \\ q \longleftarrow (\Box\neg a) \end{array} \tag{6}$$

Intuitively, this statement means, "while it is possible to do a, do b." Here, q is a new proposition whose definition is *recursive*, which is what achieves the iterative effect. Notice, again, the role of "$\Box\neg a$" in the second clause of (6). Here it says that if a cannot be executed, then terminate the loop. More discussion of imperative constructs can be found in [2].

8 Comparison with Other Works

There has been extensive work on languages for performing updates to first-order knowledge bases. In [2] we survey Process Logic [11], Action Logic [21], the Situation Calculus [18], the recent works of Reiter [23, 22], and the works based on Allen's temporal logic of time intervals [1, 10, 14] and point out their deficiencies and differences with $\mathcal{TR}$.

Acknowledgments: We wish to thank Mariano Consens, Alberto Mendelzon, and Ray Reiter for many insightful discussions. Comments by Greg Meredith, Gösta Grahne, Peter Revesz, Fangzhen Lin, Javier Pinto, Dimitris Lagouvardos, Jan Van den Bussche, and Roel Wieringa are also gratefully acknowledged.

References

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

[2] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-270, University of Toronto, April 1992. Revised: May 1993. Available in *csri-technical-reports/270/report.ps* by anonymous ftp to *csri.toronto.edu*.

[3] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming (ICLP)*, Budapest, Hungary, June 1993.

[3] One metric suitable for updating arbitrary classical theories was proposed by Winslett [27]. Others have also been proposed.

[4] Recall that elementary updates are not limited to the insertion and deletion of single tuples. Insertion of a first-order formula, such as $ins{:}\phi$ above, is a perfectly valid elementary update in $\mathcal{TR}$ [2].

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

[5] P.R. Cohen and E.A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume III. William Kaufmann, Inc., Los Altos, CA, 1982.

[6] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[7] P.E. Friedland. *Knowledge-Based Experiment Design in Molecular Genetics.* PhD thesis, Computer Science Department, Stanford University, 1979. Report Number 79-771.

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

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

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

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

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

[13] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 93/06, Department of Computer Science, SUNY at Stony Brook, April 1993. To appear in J. of ACM.

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

[15] D.K. Lewis. *Counterfactuals.* Blackwell, Oxford, 1973.

[16] V. Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Timberline, OR, 1987. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, A. Tate (eds.), Morgan-Kaufmann, 1990, 523–530.

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

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

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

[20] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publ. Co., Paolo Alto, CA, 1980.

[21] 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, 1990.

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

[23] R. Reiter. The projection problem in the situation calculus: A soundness and completeness result, with an application to database updates. Technical report, Univerity of Toronto, Department of Computer Science, January 1992.

[24] E.D. Sacerdoti. The non-linear nature of plans. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 206–214, 1975. Also appears in *Readings in Planning*, pp. 162–170. Morgan Kaufmann, San Mateo, CA, 1990.

[25] M.J. Stefik. *Planning with Constraints.* PhD thesis, Computer Science Department, Stanford University, 1980. Report Number 80-784.

[26] T. Winograd. A procedural model of language understanding. In R. Schank and K. Colby, editors, *Computer Models of Thought and Language.* Freeman Press, San Francisco, CA, 1973.

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