Applications of Transaction Logic to Knowledge Representation

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

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

Abstract

We present applications of the recently proposed *Transaction Logic*—an extension of classical logic that accounts in a clean and declarative fashion for the phenomenon of state changes in knowledge bases, including logic programs and databases. Transaction Logic has a natural model theory and a sound-and-complete proof theory, but unlike many other logics, it allows users to *program* transactions. Its semantics also leads naturally to features whose amalgamation in a single logic has proved elusive in the past. These features include both hypothetical *and* committed updates, static and dynamic constraints, nondeterministic actions, nested transactions, bulk updates, view updates, active databases, subjunctive queries, and more. Finally, Transaction Logic holds promise as a logical model of hitherto non-logical phenomena, including *procedural knowledge* in AI, and the *behavioral* aspect of object-oriented databases, especially methods with side effects. This paper outlines the model theory of Transaction Logic, and then focuses on some of its applications to AI, including action definition and execution, planning, and dynamic constraints.

This paper appears in *Proceedings of the International Conference on Temporal Logic (ICTL)*, pages 67–81. Springer Verlag, Lecture Notes in Artificial Intelligence, volume 827. Conference held in Bonn, Germany, July 11-14 1994.

This paper is available at the following URL:
`ftp://db.toronto.edu/pub/bonner/papers/transaction.logic/ictl94.ps.gz`

---

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

†Supported in part by NSF grant CCR-9102159. Part of this work is done during sabbatical year at the University of Toronto. Support of Computer Systems Research Institute of University of Toronto is gratefully acknowledged.

1 Introduction

Transaction Logic (abbreviated $\mathcal{TR}$) accounts in a clean, declarative fashion for the phenomenon of updating arbitrary logical theories, most notably, databases and logic programs. Unlike most logics of action, $\mathcal{TR}$ is a declarative formalism for specifying and executing procedures that update and permanently change a database, a logic program or, more generally, a logical theory. As a special case, transactions can be defined as logic programs. This is possible because, like classical logic, $\mathcal{TR}$ has a "Horn" version that has *both* a procedural and a declarative semantics, as well as an efficient SLD-style proof procedure. Since the formal aspects of $\mathcal{TR}$ can be found in [6, 5], this paper focuses on applications of $\mathcal{TR}$ to AI. Numerous other applications can be found in [5, 8]

$\mathcal{TR}$ was designed with several applications in mind, especially in databases, logic programming, and AI. It was therefore developed as a general logic, so that it could solve a wide range of update-related problems. Individual applications can be carved out of different fragments of the logic. These applications, both practical and theoretical, are discussed in great detail in [5]. For instance, in logic programming, $\mathcal{TR}$ leads to a clean, logical treatment of the *assert* and *retract* operators in Prolog, which effectively extends the theory of logic programming to include updates as well as queries. In object-oriented databases, $\mathcal{TR}$ can be combined with object-oriented logics, such as F-logic [15], to provide a logical account of *methods*, *i.e.*, procedures hidden inside objects that manipulate these objects' internal states. Thus, while F-logic covers the structural aspect of object-oriented databases, its combination with $\mathcal{TR}$ would account for the behavioral aspect as well. In AI, $\mathcal{TR}$ suggests a logical account of planning. STRIPS-like actions [11], for instance, and many aspects of hierarchical and non-linear planning are easily expressed in $\mathcal{TR}$. In spite of the previous efforts to give these phenomena declarative semantics, until now there has been no unifying *logical* framework to account for all of them.

On the surface, there would seem to be many other candidates for a logic of transactions, since many logics reason about time or action. However, despite a plethora of logics, researchers continue to complain that there is no clear declarative semantics for updates, whether in databases or in logic programming [4, 3, 20]. In fact—in stark contrast to classical logic—no logic of change has ever become a core of databases or logic-programming, in theory or in practice. There appear to be a few simple reasons for this unsuitability of existing logics of change. These reasons are discussed at length in [5], and we discuss some of them briefly here.

First, most logics of time or action are *hypothetical*. For instance, some systems can infer that if action A precedes B, and B precedes C, then A must precede C. Others can infer, say, that if a student took History400, 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* the 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 database, depending on the intent. Furthermore, the proof theory of $\mathcal{TR}$ is not only a verifier of truth, but also an *executor* of transactions. That is, given a transaction and an initial database state, $\mathcal{TR}$ will materialize the final database state.

Second, many logics insist on strict separation between queries and updates. However, this distinction is blurred in object-oriented systems, where both queries and updates 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. By integrating $\mathcal{TR}$ with F-logic [15], the structural aspect of object-oriented systems can be accounted for as well.

In [5], we provide a detailed comparison of $\mathcal{TR}$ with many other works. The system that comes closest in *spirit* to $\mathcal{TR}$ is Prolog. Unfortunately, updates in Prolog are non-logical operations and, as a result, state-changing procedures are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain. $\mathcal{TR}$ provides a general solution to the aforementioned limitations, both of Prolog and of action and temporal logics.

2 Overview of Transaction Logic

$\mathcal{TR}$ is an extension of first-order logic, both syntactically and semantically. It has a natural model theory and a sound-and-complete proof theory. This section gives an overview of the syntax and the model theory. A complete development of $\mathcal{TR}$, including proof theory, can be found in [5].

Like classical logic, $\mathcal{TR}$ has a "Horn" version that is of particular interest for logic programming and deductive databases. In Horn $\mathcal{TR}$, a transaction is defined by Horn-style rules, where the premise specifies a *sequence* of queries and updates. $\mathcal{TR}$ is, thus, a logical language for programming database transactions, just as classical logic is a logical language for programming queries. Furthermore, $\mathcal{TR}$ has an inference system and two natural proof procedures: an efficient SLD-style procedure, and a dual, bottom-up procedure [6, 5]. These proof procedures both answer queries, execute transactions, *and* update the database. Because of its importance, much of this paper focuses on applications of Horn $\mathcal{TR}$. First, though, we describe full $\mathcal{TR}$, without the Horn restriction.

2.1 Syntax

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

Transaction formulas are used to define transactions (or complex actions) and to formulate queries. Transaction formulas extend first-order formulas with a new connective, $\otimes$, called *serial conjunction*. Thus, if ϕ and ψ are transaction formulas, then so are $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \otimes \psi$, $\neg\phi$, $(\forall X)\phi$, and $(\exists X)\phi$, where X is a variable. The expression $a(X) \vee \neg[b(X) \otimes c(X,Y)]$ is a transaction formula. Informally, $\psi \otimes \phi$ says, "Do ψ and then do ϕ." A dual connective, *serial disjunction*, is also useful: $\psi \oplus \phi$ 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 3.2. 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 (called a *transaction base*) is equivalent to a conjunction of all the rules in the set.

A transaction base defines complex formulas in terms of simpler ones. However, we also need a way to specify *elementary changes* to a database. One way to define such transitions is to build them into the semantics as in [17, 18, 9, 1]. The main problem with this approach is that adding new kinds of elementary transitions requires redefining the very notion of a model and, hence, entails an overhaul of the entire proof theory. This is a serious drawback since there appears to be no small, single set of elementary transitions that is best for all purposes [5]. Indeed, in [5, 8], we introduce two new kinds of elementary update. Thus, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat the elementary transitions as a *parameter* of $\mathcal{TR}$. 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 $q.ins(t)$ and $q.del(t)$ stand for the insertion and deletion of atom $q(t)$, then they would be defined by an enumerable set of elementary transitions consisting of the following formulas:

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

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. The reader is referred to [5] for a more detailed discussion of transition bases.

As seen from the syntax, there is no strict distinction in $\mathcal{TR}$ between predicates that query the database and predicates that update it. As in classical logic, every predicate has a truth value, but in addition, every predicate may have a side effect by changing the state of the database. This uniformity of representation is important

[1] For relational databases, the operators $+$ and $-$ can be thought of as union and set-difference. However, if $\mathbf{D}$ is a general first-order formula, then defining insertion and deletion is more involved [14].

for modeling *methods* 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. In fact, it may be a 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, $p.ins$, to represent insertions of tuples into p. Likewise, we represent deletions from p by the predicate $p.del$. Thus the formula $p.ins(a) \otimes p.ins(b) \otimes p.ins(c)$ represents an updating transaction that inserts $p(a)$ into the database, then $p(b)$, and then $p(c)$.

2.2 Example

Before presenting the semantics, we illustrate $\mathcal{TR}$ through an example of a robot arm moving blocks around a table top [19]. Planning of robot actions is discussed in Section 3.1, and in more detail in [5]. The example shows how updates are combined with queries to define complex transactions. In the example, the body of each rule is a sequence of atomic formulas, some of which are queries and some of which are updates. We say that such rules are *serial-Horn*, or simply *Horn*. The example also illustrates the use of transaction subroutines and shows how $\mathcal{TR}$ improves upon Prolog's *assert* and *retract* operators.

Example 2.1 (Non-Deterministic Robot Actions) A state in our blocks world is defined via three database predicates: $on(x, y)$, which says that block x is on top of block y; $isclear(x)$, which says that nothing is on top of block x; and $color(x, c)$, which says that c is 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 universally quantified outside the rules.

$$
\begin{array}{l}
stackSameColor(Z) \ \leftarrow \ color(Z, C) \otimes stack2colors(C, C, Z) \\
stack2colors(C_1, C_2, Z) \ \leftarrow \ color(X, C_1) \otimes color(Y, C_2) \otimes stack2blocks(X, Y, Z) \\
stack2blocks(X, Y, Z) \ \leftarrow \ move(Y, Z) \otimes move(X, Y) \\
move(X, Y) \ \leftarrow \ pickup(X) \otimes putdown(X, Y) \\
pickup(X) \ \leftarrow \ isclear(X) \otimes on(X, Y) \otimes on.del(X, Y) \otimes isclear.ins(Y) \\
putdown(X, Y) \ \leftarrow \ wider(Y, X) \otimes isclear(Y) \otimes on.ins(X, Y) \otimes isclear.del(Y)
\end{array}
$$

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 2.1 involve queries as well as updates. In the last rule, for instance, the atom $isclear(Y)$ (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 2.1 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 [5].

Finally, observe that the rules in Example 2.1 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.[2] Prolog, too, will backtrack, but it will leave the database in an incorrect state, since it will not undo the *pickup* action. Thus, if $blkA$ was previously on top of $blkB$, then $on(blkA, blkB)$ would remain deleted and $isclear(blkB)$ would stay in the database.

2.3 Model Theory

This section gives an overview of the model theory of $\mathcal{TR}$. The reader is referred to [5, 7, 6] for a full treatment, including the proof theory. Just as the syntax of $\mathcal{TR}$ is based on two basic ideas—serial conjunction and elementary transitions—semantics is also based on a few fundamental principles, described below.

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

[2] The important point here is that this backtracking behavior of the proof theory is grounded in the model theory, which is unlike the various ad hoc ideas about "fixing" Prolog's *assert* and *retract*.

may pass through any number of intermediate states. For example, execution of the transaction $a.ins \otimes b.ins \otimes c.ins$ 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 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. Note that 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. This focus on paths is related to the version of Process Logic in [13], but the two logics are fundamentally different [5]. Because of the emphasis on paths, we refer to semantic structures in $\mathcal{TR}$ as *path structures*.

Truth on Paths: In $\mathcal{TR}$, truth is defined on paths, where a path is a finite sequence of states. For example, we would say that the formula $a.ins \otimes b.ins$ is true on the path $\mathbf{D}$, $\mathbf{D} + \{a\}$, $\mathbf{D} + \{a, b\}$, since the formula represents an insertion of the proposition a followed by an insertion of b. Note that this formula is *false* on the path $\mathbf{D}$, $\mathbf{D}+\{a, b\}$, on the path $\mathbf{D}$, $\mathbf{D}+\{b\}$, $\mathbf{D}+\{a, b\}$, and on $\mathbf{D}$, $\mathbf{D}+\{a, c\}$, $\mathbf{D}+\{a, b, c\}$. All logical connectives are interpreted on paths, *i.e.*, in terms of action. For instance, $\phi \vee \psi$ is true on a path iff ϕ is true or ψ is true. This gives rise to non-deterministic actions, since intuitively, $\phi \vee \psi$ means, "Do ϕ or do ϕ." This idea is illustrated in [5, 8]. Likewise, $\phi \wedge \psi$ is true on a path iff ϕ and ψ are both true. This provides a way of constraining non-deterministic actions. For instance, $\phi \wedge \neg\psi$ intuitively means, "Do ϕ but without doing ψ in the process." Section 3.2 illustrates this idea. Finally, note that $\otimes$ and $\wedge$ are identical on paths of length 1, and both coincide with classical conjunction. Thus, on states, $\mathcal{TR}$ reduces to classical logic.

Database States: In $\mathcal{TR}$, a database state is a class of equivalent first-order classical formulas. The need to consider equivalence classes rather than just formulas arises because syntactically distinct formulas may represent the same database state. For instance, we do not normally think of $a \wedge b$ and $b \wedge a$ as being two different databases. For convenience, we will let a single member of a class stand for the entire class. Treating states as formulas provides a lot of flexibility when defining elementary updates. Such flexibility is needed since, for general databases, the semantics of elementary updates is not obvious, 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 simple answer to this question, and many solutions have been proposed (see [14] 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 one database, $\mathbf{D}_1$, to some other database, $\mathbf{D}_2$, where a database is any (equivalence class of) first-order formulas. More generally, an elementary update may be non-deterministic, so it is not just a mapping, but a *binary relation* on databases.

The rest of this section presents the Herbrand semantics of $\mathcal{TR}$. A detailed discussion and development of this semantics is given in [7]. A general, non-Herbrand semantics and a sound-and-complete proof theory are given in [5, 6]. For convenience, we define a *split* of a path $\pi = \langle s_1, ..., s_n\rangle$ to be any pair of subpaths, $\pi_1 = \langle s_1, ..., s_i\rangle$ and $\pi_2 = \langle s_i, ..., s_n\rangle$. In this case, we write $\pi = \pi_1 \circ \pi_2$. We also define $\top$ to be a special classical Herbrand structure that satisfies every first-order formula.

Definition 2.1 **[Structures]** *A* Herbrand path structure *is a mapping,* $\mathbf{M}$*, from paths to classical Herbrand structures. For every state, s, the structure* $\mathbf{M}(\langle s \rangle)$ *must satisfy all the formulas in* s*.*

A *variable assignment* is a mapping that takes a variable as input, and returns a Hebrand term as output. We extend the mapping from variables to function terms in the usual way. As usual in logic, we omit the variable assignment for *sentences*, *i.e.*, for formulas with no free variables.

Definition 2.2 **[Satisfaction]** *Let* $\mathbf{M}$ *be a Herbrand path structure,* π *be a a path, and* ν *be a a variable assignment. Then,*

- $\mathbf{M}, \pi \models_\nu p$ *iff* $\mathbf{M}(\pi) \models^c_\nu p$, *for any atomic formula p, where "*$\models^c_\nu$*" denotes classical satisfaction.*
- $\mathbf{M}, \pi \models_\nu \neg\phi$ *iff it is not the case that* $\mathbf{M}, \pi \models_\nu \phi$*.*
- $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ *iff* $\mathbf{M}, \pi \models_\nu \phi$ *and* $\mathbf{M}, \pi \models_\nu \psi$*.* $\vee$ *is the dual of* $\wedge$*.*
- $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ *iff* $\mathbf{M}, \pi_1 \models_\nu \phi$ *and* $\mathbf{M}, \pi_2 \models_\nu \psi$ *for* some *split* $\pi_1 \circ \pi_2$ *of path* π*.*
- $\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ *iff* $\mathbf{M}, \pi_1 \models_\nu \phi$ *or* $\mathbf{M}, \pi_2 \models_\nu \psi$ *for* every *split* $\pi_1 \circ \pi_2$ *of path* π*.*
- $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ *iff* $\mathbf{M}, \pi \models_\mu \phi$ *for* every *variable assignment* μ *that agrees with* ν *everywhere except on* X*.* $\exists X$ *is the dual of* $\forall X$*.*

Definition 2.3 **[Models – Transaction Formulas]** *A path structure* $\mathbf{M}$ satisfies *a transaction formula* ϕ *iff* $\mathbf{M}, \pi \models \phi$ *for every path* π*.* $\mathbf{M}$ *is a* model *of a set of formulas iff it satisfies every formula in the set.*

Definition 2.4 **[Models – Elementary Transitions]** *A path structure* $\mathbf{M}$ satisfies *an elementary transition* $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ *if and only if* $\mathbf{M}, \langle \mathbf{D}_1, \mathbf{D}_2 \rangle \models u$*.* $\mathbf{M}$ *is a* model *of a set of transitions iff it satisfies every transition in the set.*

Definition 2.5 **[Executional Entailment]** *Let* $\mathcal{B}$ *be a set of elementary transitions,* $\mathbf{P}$ *be a set of transaction formulas,* ϕ *be a transaction formula, and* $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ *be a sequence of first-order formulas. Then,*

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

if and only if $\mathbf{M}, \langle \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \rangle \models \phi$ *for every model,* $\mathbf{M}$*, of* $\mathcal{B}$ *and* $\mathbf{P}$*.*

3 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, planning, consistency maintenance, static and dynamic constraints, non-deterministic actions, random sampling, nested transactions, view updates, heterogeneous databases, scientific databases, simulation of systems with state, rule-based inference, and more. This section describes some of these applications. Additional applications and further details of the applications presented herein can be found in [5] and [8]. In addition, $\mathcal{TR}$ can be easily extended with a modal necessity operator, $\Box$, 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 [5, 8]. $\mathcal{TR}$ 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.

3.1 Planning

Planning of robot actions is carried out by representing various planning regimes as formulas in $\mathcal{TR}$. This is possible because rules in $\mathcal{TR}$ can be used to 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 rules. Planning itself is carried out by executing plan-generating transactions using the general proof theory of [5]—no need to invent specialized search algorithms for each new planning strategy. Moreover, in $\mathcal{TR}$, the infamous *frame* problem is *not an issue* when it comes to planning and execution of actions, just as it is not an issue in STRIPS planning systems or in Pascal programming. $\mathcal{TR}$ captures this attractive property within a formal logic. The frame problem in the context of $\mathcal{TR}$ is extensively discussed in [5].

In [5], we consider three planning regimes. In the first regime, which we call *naive* planning, the inference system searches blindly for any sequence of actions that achieves the goal. Naive planning corresponds roughly to planning using a forward production system as described in [19]. The naive planner is simple to formulate, but is very inefficient, and so its utility is limited to a mere demonstration that planning is possible in $\mathcal{TR}$. However, due to space limitation, this paper gives a detailed description of naive planning only.

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 [21] and MOLGEN [12, 22], 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 [10, Article XV.D1]. In $\mathcal{TR}$, this script could be represented by a rule like the following:

$$makeCoffee \leftarrow grindCoffee \otimes boilWater \otimes fillFilter \otimes pourWater \qquad (1)$$

As this example illustrates, scripts and skeletal plans often have the form of a procedure (or recipe). For this reason, they are often specified in a procedural language. As the rule above illustrates, however, 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 [11, 16], which plans movements for a robot arm. STRIPS is naturally represented in $\mathcal{TR}$ by a script consisting of just four deductive rules and rule-schemas. It can be shown that STRIPS's planning strategy is sound, albeit incomplete, in $\mathcal{TR}$'s semantics. This incompleteness is responsible for certain failures of STRIPS, *e.g.*, its inability to exchange the contents of two registers [19]. This problem was fixed in RSTRIPS, a successor of STRIPS, by using a better search strategy. The point, however, is that, since $\mathcal{TR}$ has a *complete* proof theory [5], it provides a means of programming powerful planning systems that are free from the problems created by the use of rigid, ad hoc search algorithms. Other problems with STRIPS-like planning (which extend to many other planning systems) are pointed out and solved in [5].

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 2.1, 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 2.1 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 [5] are only some of the many applications.

Naive Planning. Naive planning is based on primitive planning actions (or operators). These primitive *actions* differ from elementary *updates* in that they correspond to real-world activities, such as robot actions. In Example 2.1, for instance, there are two primitive robot actions, $pickup(X)$ and $putdown(X, Y)$, in terms of which the other robot actions are defined. These primitive actions are in turn defined in terms of low-level database inserts and deletes, which are elementary updates that do not correspond to robot actions.

To see how naive planning is done in $\mathcal{TR}$, consider a transaction base with n primitive actions, $a_1, \ldots, a_n$. We define a *plan* to be any sequence of these actions. Such sequences are defined by the following $\mathcal{TR}$ rules:

$$\begin{array}{lcl} plan \leftarrow act \otimes plan & \qquad\qquad & act \leftarrow a_1 \\ plan \leftarrow act & & act \leftarrow a_2 \\ & & \quad \cdots \\ & & act \leftarrow a_n \end{array}$$

These rules state that *act* can be any primitive action, and *plan* is any sequence of such actions. Now, suppose we want to achieve a particular goal, such as $isclear(b)$, which states that block b must be clear. The transaction $?-\ plan \otimes isclear(b)$ would then achieve this goal, and a sequence of primitive actions that results in $isclear(b)$ being true would be recorded in the proof. More generally, we will want to achieve complex goals. For instance, suppose we want to stack block b on top of c on top of d, *i.e.*, the planning goal is $on(b, c) \wedge on(c, d)$. The transaction $?-\ plan \otimes [on(b, c) \wedge on(c, d)]$ achieves this goal. Since the predicate *on* is a query, and thus does not change the database, we can replace the classical conjunction by a serial one and write $?-\ plan \otimes on(b, c) \otimes on(c, d)$.

These ideas are easily generalized. If ψ is an arbitrary planning goal, then $?-\ plan \otimes \psi$ is a transaction that achieves the goal. This transaction, therefore, specifies a sequence of primitive actions such that ψ is true in the final database state. The most common situation is when ψ is a conjunction of queries, *i.e.*, ψ has the form $g_1 \wedge \ldots \wedge g_m$, where each g_i is a query. In this case, the transaction $?-\ plan \otimes g_1 \otimes \ldots \otimes g_m$ achieves the planning goal. The plan itself can be extracted from the proof generated by the inference system. Indeed, it suffices to retain only those steps in which the primitive actions, $a_1, \ldots, a_n$, were used in the proof of $plan \otimes g_1 \otimes \ldots \otimes g_m$, disregarding lower-level actions used to define the a_l.

Note that this formulation of planning relies on several specific features of $\mathcal{TR}$ actions:

- *Serial conjunction*, which allows primitive actions to be combined sequentially.
- *Post-conditions* on actions, which allow planning goals to be stated.

- *Recursion*, which allows the inference system to generate sequences of actions of arbitrary length, by applying the same rule(s) over and over again.

- *Non-determinism*, which allows actions to be carried out in several different ways. Together with recursion, non-determinism allows us to define a plan to be *any* sequence of primitive actions.

The approach to planning which we have just illustrated is conceptually simple, but is too unfocussed in its blind search for plans. The inference systems of $\mathcal{TR}$ [5, 6], for instance, will simply string actions together at random until a sequence is found that achieves the goal. This behavior is often called forward (or bottom-up) planning. If the plan does not achieve the goal, then the inference system simply backtracks and generates another plan. This process continues until a plan that achieves the goal is finally found.

It is due to this unfocussed behavior that we call the above planning strategy "naive." In contrast, script-based planning strategies described in [5] are highly focussed, as they augment action definitions with "scripts" or "skeletal plans" that are known to be useful in achieving certain kinds of goals. In $\mathcal{TR}$, one can easily represent the planning strategies of STRIPS and NOAH via skeletal plans consisting of a small number of rules. The actual plan is then obtained via the general proof procedure of $\mathcal{TR}$, which *executes* these script. (See [5] for details.)

3.2 Constraints on Execution of Complex Actions

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 [2], 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 $\psi \wedge \phi$ can be interpreted as, "Do transaction ψ so that ϕ will be satisfied on the execution path."[3] Intuitively, if ψ is a non-deterministic transaction, then ϕ acts as a

[3] Note that "$\wedge$" is symmetric, so other interpretations of this conjunction are also possible.

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.

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.

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 goal $goto(roomB) \otimes goto(roomA)$, *i.e.*, "go to room B, 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.*, $\texttt{path} \equiv \phi \vee \neg\phi$, for any formula ϕ. The formula $\texttt{path} \otimes at(X)$, thus, specifies a path in which the robot ends up at location X. Likewise, the formula $at(X) \otimes \texttt{path}$ specifies a path in which the robot starts off at location X, while $\texttt{path} \otimes at(X) \otimes \texttt{path}$ 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:

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

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 three 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:

$$goto(roomA) \;\wedge\; \left[\begin{array}{ll} & [go\_thru(roomB_1) \wedge go\_thru(roomB_2)] \\ \otimes & \\ & [go\_thru(roomC_1) \wedge go\_thru(roomC_2)] \end{array}\right]$$

Many more examples of constraints can be found in [5].

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

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

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

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

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

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

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

[8] 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 (DBPL)*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[9] 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.

[10] P.R. Cohen and E.A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume III. Addison-Wesley Publishing Co., Reading, MA, 1986.

[11] 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.

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

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

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

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

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

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

[20] 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.

[21] 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.

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