Planning with Transaction Logic

Reza Basseda[1], Michael Kifer[1], and Anthony J. Bonner[2]

[1]Stony Brook University, USA
{rbasseda,kifer}@cs.stonybrook.edu
[2]University of Toronto, Canada
bonner@cs.toronto.edu

Abstract. Automated planning has been the subject of intensive research and is at the core of several areas of AI, including intelligent agents and robotics. In this paper, we argue that Transaction Logic is a natural specification language for planning algorithms, which enables one to see further afield and thus discover better and more general solutions than using one-of-a-kind formalisms. Specifically, we take the well-known *STRIPS* planning strategy and show that Transaction Logic lets one specify the *STRIPS* planning algorithm easily and concisely, and to prove its completeness. Moreover, extensions to allow indirect effects and to support action ramifications come almost for free. Finally, the compact and clear logical formulation of the algorithm made possible by this logic is conducive to fruitful experimentation. To illustrate this, we show that a rather simple modification of the *STRIPS* planning strategy is also complete and yields speedups of orders of magnitude.

1 Introduction

The classical problem of automated planning is at the core of several important areas, such as robotics, intelligent information systems, and multi-agent systems, and it has been preoccupying AI researchers for over forty years.

In this paper, we argue that a general logical theory, specifically *Transaction Logic* (or $\mathcal{TR}$) [10, 9, 8], provides multiple advantages for specifying, generalizing, and solving planning problems. Transaction Logic is an extension of classical logic with dedicated support for specifying and reasoning about actions, including sequential and parallel execution, atomicity of transactions, and more. To illustrate the point, we take the classical *STRIPS* planning problem [12, 19] and show that both the *STRIPS* framework and the associated planning algorithm easily and naturally lend themselves to compact representation in Transaction Logic.

We emphasize that this paper is *not* about *STRIPS* or about inventing new planning strategies (although we do—as a side effect). It is rather about the advantages of $\mathcal{TR}$ as a general tool for specifying a wide range of planning frameworks and strategies—*STRIPS* is just an illustration. One can likewise apply $\mathcal{TR}$ to HTN-like planning systems [20] and to various enhancements of *STRIPS*, like *RSTRIPS* and *ABSTRIPS* [21].

Our contention is that precisely because *STRIPS* is cast here as a purely logical problem in a suitable general logic, a number of otherwise non-trivial extensions become low-hanging fruits and we get them almost for free. In particular, *STRIPS* planning can be naturally extended with intensional rules, which endows the framework with support for ramification [14] (i.e., with indirect effects of actions), and we show that the resulting planning algorithm is complete. Then, after inspecting the logic rules that simulate the *STRIPS* algorithm, we observe that more restrictive rules can be derived, which intuitively should involve a smaller search space. The new rules lead to a different *STRIPS*-like algorithm, which we call *fast STRIPS*, or *fSTRIPS*. We show that *fSTRIPS* is also a complete planning algorithm, and our experiments indicate that it can be orders of magnitude faster than the original.

A number of deductive planning frameworks have been proposed over the years [2–4, 16, 11, 15, 26, 27], but only a few of these approaches support any kind of action ramification. Most importantly, our work differs in the following respects.

- Many of the above approaches invent one-of-a-kind theories that are suitable only for the particular problem at hand. We, on the other hand, use a general logic, which integrates well with most other approaches to knowledge representation.
- These works typically first demonstrate how they can capture *STRIPS*-like actions and then rely on a theorem prover of some sort to find plans. This type of planning is called *naive*, as it has to contend with extremely large search spaces. In contrast, we capture not merely *STRIPS* actions—they are part of the basic functionality in $\mathcal{TR}$—but also the actual optimized *planning strategies* (*STRIPS*, HTN, etc.), which utilize heuristics to reduce the search space. That is, we first compactly express these heuristics as $\mathcal{TR}$ rules and *then* use the $\mathcal{TR}$ theorem prover to find plans. The effect is that the theorem prover, in fact, executes those specialized and more efficient algorithms.
- The clear and compact form used to represent the planning heuristics is suggestive of various optimizations, which lead to new and more efficient algorithms. We illustrate this with the example of discovery of *fSTRIPS*.

We are also unaware of anything similar to our results in the literature on the situation calculus or other first-order logic based methodologies (cf. [19, 22]).

Finally, several aspects found in the planning literature, like parallelization of plans and loops [18, 17, 24], are orthogonal to the results presented here and provide a natural direction for further extensions of our work.

This paper is organized as follows. Section 2 introduces the *STRIPS* planning framework and its extension in support of action ramification. Section 3 provides the necessary background on Transaction Logic in order to make this paper self-contained. Section 4 casts the extended *STRIPS* as a problem of posing a transactional query in Transaction Logic and shows that executing this query using the $\mathcal{TR}$'s proof theory makes for a sound and complete *STRIPS* planning algorithm. Section 5 introduces the *fSTRIPS* algorithm, which is also cast as a transactional query in $\mathcal{TR}$, and shows that this is also a complete planning strategy. In Section 6, we present our experiments that show that *fSTRIPS* can be orders of magnitude better than *STRIPS* both in time and space. Section 7 concludes the paper.

2 Extended *STRIPS*-style Planning

In this section we first remind the reader a number of standard concepts in logic and then introduce the *STRIPS* planning problem.

We assume denumerable sets of variables $\mathcal{V}$, constants $\mathcal{C}$, and predicate symbols $\mathcal{P}$—all three sets being pairwise disjoint. The set of predicates, $\mathcal{P}$, is further partitioned into *extensional* ($\mathcal{P}_{ext}$) and *intensional* ($\mathcal{P}_{int}$) predicates. In *STRIPS*, actions update the state of a system by adding or deleting statements about predicates. In the original *STRIPS*, all predicates were extensional, and the addition of intentional predicates is a major enhancement, which allows us to deal with the so-called *ramification problem* [14], i.e., with indirect consequences of actions.

Atomic formulas (or just ***atoms***) have the form $p(t_1, ..., t_n)$, were $p \in \mathcal{P}$ and each t_i is either a constant or a variable. Extending the logical signature with function symbols is straightforward in our framework, but we avoid doing this here in order to save space.

An atom is ***extensional*** if $p \in \mathcal{P}_{ext}$ and ***intensional*** if $p \in \mathcal{P}_{int}$. A ***literal*** is either an atom or a negated extensional atom of the form $\neg p(t_1, ..., t_n)$. Negated intensional atoms are not allowed. (It is not too hard to extend our framework and the results to allow negated intensional atoms, but we refrain from doing so due to space limitations).

Extensional predicates represent database facts: they can be directly manipulated (inserted or deleted) by actions. Intensional predicate symbols are used for atomic statements defined by *rules*—they are *not* affected by actions directly. Instead, actions make extensional facts true or false and this indirectly affects the dependent intensional atoms. These indirect effects are known as action *ramifications* in the literature.

A ***fact*** is a ***ground*** (i.e., variable-free) extensional atom. A set $\mathbf{S}$ of literals is ***consistent*** if there is no atom, *atm*, such that both *atm* and $\neg atm$ are in $\mathbf{S}$.

A ***rule*** is a statement of the form $head \leftarrow body$ where *head* is an intensional atom and body is a conjunction of literals. A ***ground instance*** of a rule, R, is any rule obtained from R by a substitution of variables with constants from $\mathcal{C}$ such that different occurrences of the same variable are always substituted with the same constant. Given a set $\mathbf{S}$ of literals and a ground rule of the form $atm \leftarrow \ell_1 \wedge \cdots \wedge \ell_m$, the rule is *true* in $\mathbf{S}$ if either $atm \in \mathbf{S}$ or $\{\ell_1, \ldots, \ell_m\} \nsubseteq \mathbf{S}$. A (possibly non-ground) rule is *true* in $\mathbf{S}$ if all of its ground instances are true in $\mathbf{S}$.

Definition 1 (State). *Given a set $\mathbb{R}$of rules, a **state** is a consistent set $\mathbf{S} = \mathbf{S}_{ext} \cup \mathbf{S}_{int}$ of literals such that*

1. *For each fact atm, either $atm \in \mathbf{S}_{ext}$ or $\neg atm \in \mathbf{S}_{ext}$.*
2. *Every rule of $\mathbb{R}$ is true in $\mathbf{S}$.* □

Definition 2 (*STRIPS* action). *A STRIPS **action** is a triple of the form $\alpha = \langle p_\alpha(X_1, ..., X_n), Pre_\alpha, E_\alpha \rangle$, where*

- *$p_\alpha(X_1, ..., X_n)$ is an intensional atom in which $X_1, ..., X_n$ are variables and $p_\alpha \in \mathcal{P}_{int}$ is a predicate that is reserved to represent the action α and can be used for no other purpose;*
- *Pre_α, called the **precondition** of α, is a set of literals that may include extensional as well as intensional literals;*
- *E_α, called the **effect** of α, is a consistent set that may contain extensional literals only;*
- *The variables in Pre_α and E_α must occur in $\{X_1, ..., X_n\}$.*[1] □

[1] Requiring the variables of Pre_α to occur in $\{X_1, ..., X_n\}$ is not essential for us: we can easily extend our framework and consider the extra variables to be existentially quantified.

Note that the literals in Pre_α can be both extensional and intensional, while the literals in E_α can be extensional only.

Definition 3 (Execution of a *STRIPS* action). *A STRIPS action α is* ***executable*** *in a state $\mathbf{S}$ if there is a substitution $\theta : \mathcal{V} \longrightarrow \mathcal{C}$ such that $\theta(Pre_\alpha) \subseteq \mathbf{S}$. A* ***result of the execution*** *(with respect to θ) is the state $\mathbf{S}'$ such that $\mathbf{S}' = (\mathbf{S} \setminus \neg\theta(E_\alpha)) \cup \theta(E_\alpha)$, where $\neg E = \{\neg\ell \mid \ell \in E\}$. In other words, $\mathbf{S}'$ is $\mathbf{S}$ with all effects of $\theta(\alpha)$ applied.* $\square$

Note that $\mathbf{S}'$ is well-defined since E_α is consistent. Observe also that, if α has variables, the result of an execution, $\mathbf{S}'$, may depend on the chosen substitution θ.

The following simple example illustrates the above definition. We follow the standard logic programming convention whereby lowercase symbols represent constants and predicate symbols. The uppercase symbols denote variables that are implicitly universally quantified outside of the rules.

Example 1. Consider a world consisting of just two blocks and the action $pickup = \langle pickup(X,Y), \{clear(X)\}, \{\neg on(X,Y), clear(Y)\}\rangle$. Consider also the state $\mathbf{S} = \{clear(a), \neg clear(b), on(a,b), \neg on(b,a)\}$. Then the result of the execution of $pickup$ at state $\mathbf{S}$ with respect to the substitution $\{X \to a, Y \to b\}$ is $\mathbf{S}' = \{clear(a), clear(b), \neg on(a,b), \neg on(b,a)\}$. It is also easy to see that $pickup$ cannot be executed at $\mathbf{S}$ with respect to any substitution of the form $\{X \to b, Y \to ...\}$. $\square$

Definition 4 (Planning problem). *A* ***planning problem*** *$\langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ consists of a set of rules $\mathbb{R}$, a set of STRIPS actions $\mathbb{A}$, a set of literals G, called the* ***goal*** *of the planning problem, and an* ***initial state*** *$\mathbf{S}$. A sequence of actions $\sigma = \alpha_1, \ldots, \alpha_n$ is a* ***planning solution*** *(or simply a* ***plan****) for the planning problem if:*

- *$\alpha_1, \ldots, \alpha_n \in \mathbb{A}$; and*
- *there is a sequence of states $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_n$ such that*
  - *$\mathbf{S} = \mathbf{S}_0$ and $G \subseteq \mathbf{S}_n$ (i.e., G is satisfied in the final state);*
  - *for each $0 < i \leq n$, α_i is executable in state $\mathbf{S}_{i-1}$ and the result of that execution (for some substitution) is the state $\mathbf{S}_i$.*

In this case we will also say that $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_n$ is an execution of σ. $\square$

3 Overview of Transaction Logic

To make this paper self-contained, we provide a brief introduction to the parts of Transaction Logic that are needed for the understanding of this paper. For further details, the reader is referred to [7, 9, 10, 5, 8].

$\mathcal{TR}$ is a faithful extension of the first-order predicate calculus and so all of that syntax carries over. In this paper, we focus on rules, however, so we will be dealing exclusively with that subset of the syntax from now on. The most important new connective that Transaction Logic brings in is the ***serial conjunction***, denoted $\otimes$. It is a binary associative connective, like the classical conjunction, but it is not commutative. Informally, the formula $\phi \otimes \psi$ is understood as a composite action that denotes an *execution* of ϕ followed by an execution of ψ. The ***concurrent conjunction*** connective,

$\phi \| \psi$, is associative *and commutative*. Informally, it says that ϕ and ψ can execute in an *interleaved* fashion. For instance, $(\alpha_1 \otimes \alpha_2) \| (\beta_1 \otimes \beta_2)$ can execute as $\alpha_1, \beta_1, \alpha_2, \beta_2$, or as $\alpha_1, \beta_1, \beta_2, \alpha_2$, or as $\alpha_1, \alpha_2, \beta_1, \beta_2$, while $(\alpha_1 \otimes \alpha_2) \otimes (\beta_1 \otimes \beta_2)$ can execute only as $\alpha_1, \alpha_2, \beta_1, \beta_2$. When ϕ and ψ are regular first-order formulas, both $\phi \otimes \psi$ and $\phi \| \psi$ reduce to the usual first-order conjunction, $\phi \wedge \psi$. The logic also has other connectives but they are beyond the scope of this paper.

In addition, $\mathcal{TR}$ has a general, extensible mechanism of ***elementary updates*** or elementary ***actions*, ** which have the important effect of taking the infamous *frame problem* out of many considerations in this logic (see [9, 10, 7, 23, 6]). Here we will use only the following two types of elementary actions, which are specifically designed on complete *STRIPS* states (Definition 1): $+p(t_1, \ldots, t_n)$ and $-p(t_1, \ldots, t_n)$, where p denotes an *extensional* predicate symbol of appropriate arity and t_1, ..., t_n are terms.

Given a state $\mathbf{S}$ and a *ground* elementary action $\alpha = +p(a_1, \ldots, a_n)$, an execution of α at state $\mathbf{S}$ deletes the literal $\neg p(a_1, \ldots, a_n)$ and adds the literal $p(a_1, \ldots, a_n)$. Similarly, executing $-p(a_1, \ldots, a_n)$ results in a state that is exactly like $\mathbf{S}$, but $p(a_1, \ldots, a_n)$ is deleted and $\neg p(a_1, \ldots, a_n)$ added. In some cases (e.g., if $p(a_1, \ldots, a_n) \in \mathbf{S}$), the action $+p(a_1, \ldots, a_n)$ has no effect, and similarly for $-p(a_1, \ldots, a_n)$.

A ***serial rule*** is a statement of the form

$$h \leftarrow b_1 \otimes b_2 \otimes \ldots \otimes b_n. \tag{1}$$

where h is an atomic formula and b_1, ..., b_n are literals or elementary actions. The informal meaning of such a rule is that h is a complex action and one way to execute h is to execute b_1 then b_2, etc., and finally to execute b_n.

Thus, we now have regular first-order as well as serial-Horn rules. For simplicity (thought this is not required by $\mathcal{TR}$), we assume that the sets of intentional predicates that can appear in the heads of regular rules and those in the heads of serial rules are disjoint. Thus, we now have the following types of atomic statements:

- *Extensional atoms*.
- *Intentional atoms*: The atoms that appear in the heads of regular rules. These two categories of atoms populate database states and will be collectively called ***fluents***. We will now allow any kind of fluent to be negated in the body of a serial rule of the form (1).
- *Elementary actions*: $+p$, $-p$, where p is an extensional atom.
- *Complex actions*: These are the atoms that appear at the head of the serial rules. Complex and elementary actions will be collectively called ***actions***.

As remarked earlier, for fluents $f \otimes g$ is equivalent to $f \wedge g$ and we will often write $f \wedge g$ for fluents even if they occur in the bodies of serial rules. Note that a serial rule all of whose body literals are fluents is essentially a regular rule, since all the $\otimes$-connectives can be replaced with $\wedge$. Therefore, one can view the regular rules as a special case of serial rules.

The following example illustrates the above concepts.

$$\begin{array}{rl}
move(X,Y) & \leftarrow (on(X,Z) \wedge clear(X) \wedge clear(Y) \wedge \neg tooHeavy(X)) \\
 & \quad \otimes -on(X,Z) \otimes +on(X,Y) \otimes -clear(Y). \\
tooHeavy(X) & \leftarrow weight(X,W) \wedge limit(L) \wedge W < L. \\
?- & move(blk1, blk15) \otimes move(SomeBlk, blk1).
\end{array}$$

Here *on*, *clear*, *tooHeavy*, *weight*, and *limit* are fluents and the rest of atoms represent actions. The predicate *tooHeavy* is an intentional fluent, while *on*, *clear*, and *weight* are extensional fluents. The actions $+on(...)$, $-clear(...)$, and $-on(...)$ are elementary and the intentional predicate *move* represents a complex action. This example illustrates several features of Transaction Logic. The first rule is a serial rule defining of a complex action of moving a block from one place to another. The second rule defines the intensional fluent *tooHeavy*, which is used in the definition of *move* (under the scope of default negation). As the second rule does not include any action, it is a regular rule.

The last statement above is a *request to execute* a composite action, which is analogous to a query in logic programming. The request is to move block *blk1* from its current position to the top of *blk15* and then find some other block and move it on top of *blk1*. Traditional logic programming offers no logical semantics for updates, so if after placing *blk1* on top of *blk15* the second operation ($move(SomeBlk, blk1)$) fails (say, all available blocks are too heavy), the effects of the first operation will persist and the underlying database becomes corrupted. In contrast, Transaction Logic gives update operators the logical semantics of an *atomic database transaction*. This means that if any part of the transaction fails, the effect is as if nothing was done at all. For example, if the second action in our example fails, all actions are "backtracked over" and the underlying database state remains unchanged.

This semantics is given in purely model-theoretic terms and here we will only give an informal overview. The truth of any action in $\mathcal{TR}$ is determined over sequences of states—***execution paths***—which makes it possible to think of truth assignments in $\mathcal{TR}$'s models as executions. If an action, ϕ, defined by a set of serial rules, $\mathbb{P}$, evaluates to true over a sequence of states $\mathbf{D}_0, \ldots, \mathbf{D}_n$, we say that it can *execute* at state $\mathbf{D}_0$ by passing through the states $\mathbf{D}_1$, ..., $\mathbf{D}_{n-1}$, ending in the final state $\mathbf{D}_n$. This is captured by the notion of ***executional entailment***, which is written as follows:

$$\mathbb{P}, \mathbf{D}_0 \ldots \mathbf{D}_n \models \phi \tag{2}$$

The next example further illustrates $\mathcal{TR}$ by showing a definition of a recursive action.

Example 2 (Pyramid building). The following rules define a complex operation of stacking blocks to build a pyramid. It uses some of the already familiar fluents and actions from the previous example. In addition, it defines the actions *pickup*, *putdown*, and a recursive action $stack$.

$$\begin{array}{l}
stack(0, AnyBlock) \leftarrow . \\
stack(N, X) \leftarrow N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \otimes on(Y, X). \\
move(X, Y) \leftarrow X \neq Y \otimes pickup(X) \otimes putdown(X, Y). \\
pickup(X) \leftarrow clear(X) \otimes on(X, Y) \otimes -on(X, Y) \otimes +clear(Y). \\
pickup(X) \leftarrow clear(X) \otimes on(X, table) \otimes -on(X, table). \\
putdown(X, Y) \leftarrow clear(Y) \otimes \neg on(X, Z1) \otimes \neg on(Z2, X) \otimes \\
\qquad\qquad\qquad -clear(Y) \otimes +on(X, Y).
\end{array} \tag{3}$$

The first rule says that stacking zero blocks on top of X is a no-op. The second rule says that, for bigger pyramids, stacking N blocks on top of X involves moving some other block, Y, on X and then stacking $N-1$ blocks on Y. To make sure that the planner did not remove Y from X while building the pyramid on Y, we are verifying that $on(Y, X)$ continues to hold at the end. The remaining rules are self-explanatory. □

Several inference systems for serial-Horn $\mathcal{TR}$ are described in [7]—all analogous to the well-known SLD resolution proof strategy for Horn clauses plus some $\mathcal{TR}$-specific inference rules and axioms. The aim of these inference systems is to prove statements of the form $\mathbb{P}, \mathbf{D} \cdots \vdash \phi$, called ***sequents***. Here $\mathbb{P}$ is a set of serial rules and ϕ is a *serial goal*, i.e., a formula that has the form of a body of a serial rule, such as (1). A proof of a sequent of this form is interpreted as a proof that action ϕ defined by the rules in $\mathbb{P}$ can be successfully executed starting at state $\mathbf{D}$.

An inference succeeds if it finds an ***execution*** for the transaction ϕ, i.e., a sequence of database states $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathbb{P}, \mathbf{D}\,\mathbf{D}_1 \ldots \mathbf{D}_n \models \phi$. Here we will use the following inference system, which we present in a simplified form—only the version for ground facts and rules. The inference rules can be read either top-to-bottom (if *top* is proved then *bottom* is proved) or bottom-to-top (to prove *bottom* first prove *top*).

Definition 5 **($\mathcal{TR}$ inference System).** *Let $\mathbb{P}$ be a set of rules (serial or regular) and $\mathbf{D}$, $\mathbf{D}_1$, $\mathbf{D}_2$ denote states.*

- *Axiom: $\mathbb{P}, \mathbf{D} \cdots \vdash ()$, where $()$ is an empty clause (which is true at every state).*
- *Inference Rules*
  1. *Applying transaction definition: Suppose $t \leftarrow body$ is a rule in $\mathbb{P}$.*
$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash body \otimes rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \qquad (4)$$
  2. *Querying the database: If $\mathbf{D} \models t$ then*
$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \qquad (5)$$
  3. *Performing elementary updates: If the elementary update t changes the state $\mathbf{D}_1$ into the state $\mathbf{D}_2$ then*
$$\frac{\mathbb{P}, \mathbf{D}_2 \cdots \vdash rest}{\mathbb{P}, \mathbf{D}_1 \cdots \vdash t \otimes rest} \qquad (6)$$
  4. *Concurrency: If ϕ_i, $i = 1, ..., n$ are serial conjunctions then*
$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash \phi_1 \| ... \| \phi_j \| ... \| \phi_n}{\mathbb{P}, \mathbf{D}' \cdots \vdash \phi_1 \| ... \| \phi'_j \| ... \| \phi_n} \qquad (7)$$
     for any j, $1 \leq j \leq n$, where $\mathbf{D}'$ is obtained from $\mathbf{D}$ and ϕ'_j from ϕ_j as in either of the inference rules (4-6) above.

A ***proof*** of a sequent, seq_n, is a series of sequents, seq_1, seq_2, $\ldots$, seq_{n-1}, seq_n, where each seq_i is either an axiom-sequent or is derived from earlier sequents by one of the above inference rules. This inference system has been proven sound and complete with respect to the model theory of $\mathcal{TR}$ [7]. This means that if ϕ is a serial goal, the executional entailment $\mathbb{P}, \mathbf{D}\,\mathbf{D}_1 \ldots \mathbf{D}_n \models \phi$ holds if and only if there is a proof of $\mathbb{P}, \mathbf{D} \cdots \vdash \phi$ over the execution path $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$, i.e., $\mathbf{D}_1, \ldots, \mathbf{D}_n$ is the sequence of intermediate states that appear in the proof and $\mathbf{D}$ is the initial state. In this case, we will also say that such a proof proves the statement $\mathbb{P}, \mathbf{D}\,\mathbf{D}_1 \ldots \mathbf{D}_n \vdash \phi$.

4 The $\mathcal{TR}$-*STRIPS* Planner

The informal idea of using $\mathcal{TR}$ as a planning formalism and an encoding of *STRIPS* as a set of $\mathcal{TR}$ rules first appeared in an unpublished report [7]. The encoding was incomplete and it did not include ramification and intensional predicates. We extend the original method with intentional predicates, make it complete, and formulate and prove the completeness of the resulting planner.

Definition 6 (Enforcement operator). *Let G be a set of extensional literals. We define Enf$(G) = \{+p \mid p \in G\} \cup \{-p \mid \neg p \in G\}$. In other words, Enf$(G)$ is the set of elementary updates that makes G true.* □

Next we introduce a natural correspondence between *STRIPS* actions and $\mathcal{TR}$ rules.

Definition 7 (Actions as $\mathcal{TR}$ rules). *Let $\alpha = \langle p_\alpha(\overline{X}), Pre_\alpha, E_\alpha \rangle$ be a STRIPS action. We define its* ***corresponding $\mathcal{TR}$ rule****, $tr(\alpha)$, to be a rule of the form*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre_\alpha} \ell) \otimes (\otimes_{u \in Enf(E_\alpha)} u). \tag{8}$$

Note that in (8) the actual order of action execution in the last component, $\otimes_{u \in Enf(E_\alpha)} u$, is immaterial, since all such executions happen to lead to the same state.

We now define a set of $\mathcal{TR}$ clauses that simulate the well-known *STRIPS* planning algorithm and extend this algorithm to handle intentional predicates and rules. The reader familiar with the *STRIPS* planner should not fail to notice that, in essence, these rules are a natural (and much more concise and general) verbalization of the classical *STRIPS* algorithm [12]. However—importantly—unlike the original *STRIPS*, these rules constitute a *complete* planner when evaluated with the $\mathcal{TR}$ proof theory.

Definition 8 ($\mathcal{TR}$ planning rules). *Let $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S} \rangle$ be a STRIPS planning problem (see Definition 4). We define a set of $\mathcal{TR}$ rules, $\mathbb{P}(\Pi)$, which provides a sound and complete solution to the STRIPS planning problem. $\mathbb{P}(\Pi)$ has three disjoint parts, $\mathbb{P}_\mathbb{R}$, $\mathbb{P}_\mathbb{A}$, and $\mathbb{P}_G$, described below.*

- *The $\mathbb{P}_\mathbb{R}$ part: for each rule $p(\overline{X}) \leftarrow p_1(\overline{X}_1) \wedge \cdots \wedge p_k(\overline{X}_n)$ in $\mathbb{R}$, $\mathbb{P}_\mathbb{R}$ has a rule of the form*

$$achieve\_p(\overline{X}) \leftarrow \|_{i=1}^{n} achieve\_p_i(\overline{X}_i). \tag{9}$$

  Rule (9) is an extension to the classical STRIPS planning algorithm and is intended to capture intentional predicates and ramification of actions; it is the only major aspect of our $\mathcal{TR}$-based rendering of STRIPS that was not present in the original in one way or another.
- *The part $\mathbb{P}_\mathbb{A} = \mathbb{P}_{actions} \cup \mathbb{P}_{atoms} \cup \mathbb{P}_{achieves}$ is constructed out of the actions in $\mathbb{A}$ as follows:*
  - *$\mathbb{P}_{actions}$: for each $\alpha \in \mathbb{A}$, $\mathbb{P}_{actions}$ has a rule of the form*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre_\alpha} \ell) \otimes (\otimes_{u \in Enf(E_\alpha)} u). \tag{10}$$

    This is the $\mathcal{TR}$ rule that corresponds to the action α, introduced in Definition 7.
  - *$\mathbb{P}_{atoms} = \mathbb{P}_{achieved} \cup \mathbb{P}_{enforced}$ has two disjoint parts as follows:*

– $\mathbb{P}_{achieved}$*: for each extensional predicate* $p \in \mathcal{P}_{ext}$*,* $\mathbb{P}_{achieved}$ *has the rules*

$$\begin{array}{l} achieve\_p(\overline{X}) \leftarrow p(\overline{X}). \\ achieve\_not\_p(\overline{X}) \leftarrow \neg p(\overline{X}). \end{array} \tag{11}$$

These rules say that if an extensional literal is true in a state then that literal has already been achieved as a goal.

– $\mathbb{P}_{enforced}$*: for each action* $\alpha = \langle p_\alpha(\overline{X}), Pre_\alpha, E_\alpha \rangle$ *in* $\mathbb{A}$ *and each* $e(\overline{Y}) \in E_\alpha$*,* $\mathbb{P}_{enforced}$ *has the following rule:*

$$achieve\_e(\overline{Y}) \leftarrow execute\_p_\alpha(\overline{X}). \tag{12}$$

This rule says that one way to achieve a goal that occurs in the effects of an action is to execute that action.

- $\mathbb{P}_{achieves}$*: for each action* $\alpha = \langle p_\alpha(\overline{X}), Pre_\alpha, E_\alpha \rangle$ *in* $\mathbb{A}$*,* $\mathbb{P}_{achieves}$ *has the following rule:*

$$execute\_p_\alpha(\overline{X}) \leftarrow (\|_{\ell \in Pre_\alpha} achieve\_\ell) \otimes p_\alpha(\overline{X}). \tag{13}$$

This means that to execute an action, one must first achieve the precondition of the action and then perform the state changes prescribed by the action.

– $\mathbb{P}_G$*: Let* $G = \{g_1, ..., g_k\}$*. Then* $\mathbb{P}_G$ *has a rule of the form:*

$$achieve_G \leftarrow (\|_{g_i=1}^{k} achieve\_g_i) \otimes (\wedge_{i=1}^{k} g_i). \tag{14}$$

Given a set $\mathbb{R}$ of rules, a set $\mathbb{A}$ of *STRIPS* actions, an initial state $\mathbf{S}$, and a goal G, Definition 8 gives a set of $\mathcal{TR}$ rules that specify a planning strategy for that problem. To find a solution for that planning problem, one simply needs to place the request

$$? - achieve_G\,. \tag{15}$$

at a desired initial state and use the $\mathcal{TR}$'s inference system of Section 3 to find a proof. The inference system in question is sound and complete for *serial clauses*, and the rules in Definition 8 satisfy that requirement.

Example 3 (Planning rules for register exchange). Consider the classical problem of swapping two registers in a computer from [21]. The reason this problem is interesting is because it is the simplest problem where the original *STRIPS* is incomplete. Example 4 explains why and how our complete $\mathcal{TR}$-based planner handles the issue.

Consider two memory registers, x and y, with initial contents a and b, respectively. The goal is to find a plan to exchange the contents of these registers with the help of an auxiliary register, z. Let the extensional predicate $value(Reg, Val)$ represent the content of a register. Then the initial state of the system is $\{value(x, a), value(y, b)\}$. Suppose the only available action is $copy = \langle copy(Src, Dest, V), \{value(Src, V)\}, \{\neg value(Dest, V), value(Dest, V)\}\rangle$, which copies the value V of the source register, Src, to the destination register $Dest$. The old value of $Dest$ is erased and the value of Src is written over. The planning goal is $G = \{value(x, b), value(y, a)\}$. Per Definition 8, the planning rules for this problem are as follows.

Due to case (10):

$$\begin{aligned} copy(Src, Dest, V) \leftarrow\ & value(Src, V)\ \otimes \\ & -value(Dest, \_)\ \otimes +value(Dest, V). \end{aligned} \tag{16}$$

Due to (11), (12), and (13):

$$\begin{aligned} achieve\_value(R, V) &\leftarrow value(R, V). \\ achieve\_not\_value(R, V) &\leftarrow \neg value(R, V). \end{aligned} \tag{17}$$

$$achieve\_value(Dest, V) \leftarrow execute\_copy(Src, Dest, V). \tag{18}$$

$$\begin{aligned} execute\_copy(Src, Dest, V) \leftarrow\ & achieve\_value(Src, V) \otimes \\ & copy(Src, Dest, V). \end{aligned} \tag{19}$$

Due to (14):

$$\begin{aligned} achieve_G \leftarrow\ & (achieve\_value(x, b) \parallel achieve\_value(y, a)) \\ & \otimes (value(x, b)\ \wedge value(y, a)). \end{aligned} \tag{20}$$

Case (9) of Definition 8 does not contribute rules in this example because the planning problem does not involve intensional fluents. $\Box$

As mentioned before, a solution plan for a *STRIPS* planning problem is a sequence of actions leading to a state that satisfies the planning goal. Such a sequence can be extracted by picking out the atoms of the form p_α from a successful derivation branch generated by the $\mathcal{TR}$ inference system. Since each p_α uniquely corresponds to a *STRIPS* action, this provides us with the requisite sequence of actions that constitutes a plan.

Suppose $seq_0, \ldots, seq_m$ is a deduction by the $\mathcal{TR}$ inference system. Let $i_1, \ldots, i_n$ be exactly those indexes in that deduction where the inference rule (4) was applied to some sequent using a rule of the form $tr(\alpha_{i_r})$ introduced in Definition 7. We will call $\alpha_{i_1}, \ldots, \alpha_{i_n}$ the ***pivoting sequence of actions***. The corresponding **pivoting sequence of states** $\mathbf{D}_{i_1}, \ldots, \mathbf{D}_{i_n}$ is a sequence where each $\mathbf{D}_{i_r}$, $1 \leq r \leq n$, is the state at which α_{i_r} is applied. We will prove that the pivoting sequence of actions is a solution to the planning problem.

All theorems in this section assume that $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{D}_0 \rangle$ is a *STRIPS* planning problem and that $\mathbb{P}(\Pi)$ is the corresponding set of $\mathcal{TR}$ planning rules as in Definition 8.

Theorem 1 (Soundness of $\mathcal{TR}$ planning). *Any pivoting sequence of actions in the derivation of* $\mathbb{P}(\Pi), \mathbf{D}_0 \ldots \mathbf{D}_m \vdash achieve_G$ *is a solution plan.*[2]

Completeness of a planning strategy means that, for any *STRIPS* planning problem, if there is a solution, the planner will find at least one plan. Completeness of $\mathcal{TR}$ planning is established by induction on the length of the plans.

Theorem 2 (Completeness of $\mathcal{TR}$ planning). *If there is a plan that achieves the goal* G *from the initial state* $\mathbf{D}_0$ *then the* $\mathcal{TR}$*-based STRIPS planner will find a plan.*

[2] Sequents of the form $\mathbb{P}(\Pi), \mathbf{D}_0 \ldots \mathbf{D}_m \vdash \ldots$ were defined at the very end of Section 3.

Theorem 2 establishes the completeness of the planner that is comprised of the $\mathcal{TR}$ proof theory and the rules that express the original *STRIPS* strategy.

Recall that the *classical STRIPS* planner described in [12, 21] was incomplete. The next example illustrates the reason for this incompleteness and contrasts the situation to the $\mathcal{TR}$-based planner.

Example 4 (Register exchange, continued). Consider the register exchange problem of Example 3. The original *STRIPS* planner fails to find a plan if, in the initial state, the auxiliary register z has the value t distinct from a and b [21]. We will now illustrate how the $\mathcal{TR}$ based planner deals with this case. Let $\mathbb{P}$ be the set of $\mathcal{TR}$ rules (16-19) that constitute the planner for the $\mathcal{TR}$-based planner for this problem. Given the planning goal $G = \{value(x, b), value(y, a)\}$ and the initial state $\mathbf{D}_0$, where $\{value(x, a), value(y, b)\} \subseteq \mathbf{D}_0$, we will show how the $\mathcal{TR}$ inference system constructs a derivation (and thus a plan) for the sequent $\mathbb{P}, \mathbf{D}_0 \cdots \mathbf{D}_n \vdash \mathit{achieve}_G$ for some $\mathbf{D}_n$ such that $\{value(x, b), value(y, a)\} \subseteq \mathbf{D}_n$.

Consider the sequent $\mathbb{P}, \mathbf{D}_0 \cdots \vdash \mathit{achieve}_G$ that corresponds to the query (15). Applying the inference rule (4) to that sequent using the rule (20), we get:

$$\begin{aligned}\mathbb{P}, \mathbf{D}_0 \cdots \vdash\ &(achieve\_value(x, b) \| achieve\_value(y, a))\\ &\otimes (value(x, b) \land value(y, a))\end{aligned}$$

Applying the inference rule (4) twice to the resulting sequent using the rules (18–19) with appropriate substitutions result in:

$$\begin{aligned}\mathbb{P}, \mathbf{D}_0 \cdots \vdash\ &((achieve\_value(z, b) \otimes copy(z, x, b)) \| achieve\_value(y, a))\\ &\otimes (value(x, b) \land value(y, a))\end{aligned}$$

Applying the inference rule (4) once more and again using the rules (18–19) we get:

$$\begin{aligned}\mathbb{P}, \mathbf{D}_0 \cdots \vdash\ &((achieve\_value(y, b) \otimes copy(y, z, b) \otimes copy(z, x, b))\\ &\| \, achieve\_value(y, a)) \otimes (value(x, b) \land value(y, a))\end{aligned}$$

One more application of the inference rule (4) but this time in conjunction with (17) yields:

$$\begin{aligned}\mathbb{P}, \mathbf{D}_0 \cdots \vdash\ &((value(y, b) \otimes copy(y, z, b) \otimes copy(z, x, b))\\ &\| \, achieve\_value(y, a)) \otimes (value(x, b) \land value(y, a))\end{aligned}$$

Since $value(y, b) \in \mathbf{D}_0$, we can eliminate it by the inference rule (5). Then we can replace the first *copy* using its definition (16) due to the inference rule (4).

$$\begin{aligned}\mathbb{P}, \mathbf{D}_0 \cdots \vdash\ &((-value(z, \_) \otimes +value(z, b) \otimes copy(z, x, b))\\ &\| \, achieve\_value(y, a)) \otimes (value(x, b) \land value(y, a))\end{aligned}$$

Applying the inference rule (6) twice to the primitive updates at the front first yields

$$\begin{aligned}\mathbb{P}, \mathbf{D}_1 \cdots \vdash\ &((+value(z, b) \otimes copy(z, x, b))\\ &\| \, achieve\_value(y, a)) \otimes (value(x, b) \land value(y, a))\end{aligned}$$

and then

$$\mathbb{P}, \mathbf{D}_2 \cdots \vdash (\mathit{copy}(z,x,b) \| \mathit{achieve\_value}(y,a)) \otimes (\mathit{value}(x,b) \wedge \mathit{value}(y,a))$$

where $\mathbf{D}_1$ is $\mathbf{D}_0$ with $value(z,t)$ (where t denotes the old value of z) deleted and $\mathbf{D}_2$ is $\mathbf{D}_1$ with $value(z,b)$ added.

Now we can use the inference rule (7) to explore the subgoal $achieve\_value(y,a)$. Namely, we can expand this subgoal with the inference rule (4) twice, first using (18–19) and then using (17), obtaining

$$\begin{aligned}\mathbb{P}, \mathbf{D}_2 \cdots \vdash\ & (\mathit{copy}(z,x,b) \| (\mathit{value}(x,a) \otimes \mathit{copy}(x,y,a))) \\ & \otimes (\mathit{value}(x,b) \wedge \mathit{value}(y,a))\end{aligned}$$

Since $value(x,a)$ is true in $\mathbf{D}_2$, it can be removed. Finally, the two *copy*'s can be replaced by their definition (16) and then the remaining $+value(...)$ and $-value(...)$ can be executed using the inference rule (6). This will advance the database (via three intermediate states) to state $\mathbf{D}_6$ containing $\{value(x,b), value(y,a), value(z,b)\}$ in which both $value(x,b)$ and $value(y,a)$ are true. Therefore, the inference rule (5) can be used to derive the $\mathcal{TR}$ axiom $\mathbb{P}, \mathbf{D}_6 \cdots \vdash ()$, thus concluding the proof. The pivoting sequence of actions in this proof is $\langle \mathit{copy}(y,z,b), \mathit{copy}(x,y,a), \mathit{copy}(z,x,b)\rangle$, which constitutes the desired plan. □

5 The *fSTRIPS* Planner

In this section, we introduce *fSTRIPS* — a modification of the previously introduced *STRIPS* transform, which represents to a new planning strategy, which we call *fast STRIPS*. We show that although the new strategy explores a smaller search space, it is still sound and complete. Section 6 shows that *fSTRIPS* can be orders of magnitude faster than *STRIPS*.

Definition 9 **($\mathcal{TR}$ planning rules for *fSTRIPS*).** *Let $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ be a STRIPS planning problem as in Definition 4 and $\mathbb{P}(\Pi)$ is as in Definition 8. We define $\mathbb{P}_f(\Pi)$ to be exactly as $\mathbb{P}(\Pi)$ except for the $\mathbb{P}_{enforced}$ part. For $\mathbb{P}_f(\Pi)$, we redefine $\mathbb{P}^f_{enforced}$ (the replacement of $\mathbb{P}_{enforced}$) as follows:*

For each action $\alpha = \langle p_\alpha(\overline{X}), Pre_\alpha, E_\alpha\rangle$ in $\mathbb{A}$ and each $e(\overline{Y}) \in E_\alpha$, $\mathbb{P}^f_{enforced}$ has the following rule:

$$achieve\_e(\overline{Y}) \leftarrow \neg e(\overline{Y}) \otimes execute\_p_\alpha(\overline{X}). \qquad (21)$$

This rule says that an action, α, should be attempted only if it helps to achieve the currently pursued, unsatisfied goal. □

The other key aspect of *fSTRIPS* is that it uses a modified (general, unrelated to planning) proof theory for $\mathcal{TR}$, which relies on *tabling*, a technique analogous to [25]. This theory was introduced in [13] and was shown to be sound and complete. Here we use it for two reasons. First, it terminates if the number of base fluents is finite. Second, it has the property that it will not attempt to construct plans that have extraneous loops and thus will not attempt to large and unnecessary parts of the search space.

To construct a plan, as before, we can extract a pivoting sequence of actions with respect to *fSTRIPS* and show that the new pivoting sequence of actions is still a solution plan.

Similarly to Section 4, we assume till the end of this section that $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{D}_0 \rangle$ is a *STRIPS* planning problem, that $\mathbb{P}(\Pi)$ is the set of planning rules in Definition 8, and that $\mathbb{P}_f(\Pi)$ is the set of planning rules as specified in Definition 9.

Theorem 3 (Soundness of *fSTRIPS*). *Any pivoting sequence of actions in the derivation of* $\mathbb{P}_f(\Pi), \mathbf{D}_0 \ldots \mathbf{D}_m \vdash achieve_G$ *is a solution plan.*

Theorem 4 (Completeness of *fSTRIPS*). *If there is a plan to achieve the goal G from an initial state,* $\mathbf{D}_0$*, then* $\mathcal{TR}$ *will find a plan.*

Theorem 5 (*fSTRIPS* finds no more plans than *STRIPS*). *Any plan found by the fSTRIPS planner will also be found by the STRIPS planner.*

In other words, the *STRIPS* strategy may generate more plans than *fSTRIPS*. The plans that are not generated by *fSTRIPS* are those that contain actions whose effects were not immediately required at the time of the action selection. This has the effect of ignoring longer plans when shorter plans are already found. The upshot of all this is that *STRIPS* has a larger search space to explore, and this explains the inferior performance of *STRIPS* compared to *fSTRIPS*, as the experiments in the next section show.

6 Experiments

In this section we briefly report on our experiments that compare *STRIPS* and *fSTRIPS*, The test environment was a tabled $\mathcal{TR}$ interpreter [13] implemented in XSB and running on Intel®Xeon(R) CPU E5-1650 0 @ 3.20GHz 12 CPU and 64GB memory running on Mint Linux 14 64-bit.

The actual test cases are taken from [1] and represent so called *State Modifying Policies*. A typical use of such a policy is to determine if a particular access request (say, to play digital contents) should be granted. The first test case, a *Movie Store*, is shown in Example 5. The second test case, a *Health Care Authorization* example, is too large to be included here and can be found at *http://ewl.cewit.stonybrook.edu/planning/* along with the first test case and all the necessary items needed to reproduce the results.

Example 5 (State Modifying Policy for a Movie Store). The following represents a policy where users can buy movies online, try them, and sell them, if not satisfied.

$$\begin{aligned}
buy(X,M) &\leftarrow \neg bought(\_,M) \otimes +bought(X,M)\\
play1(X,M) &\leftarrow bought(X,M) \otimes \neg played1(X,M) \otimes +played1(X,M)\\
keep(X,M) &\leftarrow bought(X,M) \otimes \neg played1(X,M) \otimes +played1(X,M)\\
&\quad \otimes + happy(X,M)\\
play2(X,M) &\leftarrow played1(X,M) \otimes \neg played2(X,M) \otimes +played2(X,M)\\
play3(X,M) &\leftarrow played2(X,M) \otimes \neg played3(X,M) \otimes +played3(X,M)\\
sell(X,M) &\leftarrow played1(X,M) \otimes \neg played3(X,M) \otimes \neg happy(X,M)\\
&\quad \otimes + sold(X,M) \otimes -bought(X,M)
\end{aligned} \tag{22}$$

Size of goal	Movie Store				Size of goal	Health Care			
	STRIPS		*fSTRIPS*			*STRIPS*		*fSTRIPS*	
	CPU	Mem	CPU	Mem		CPU	Mem.	CPU	Mem.
6	0.0160	1095	0.0080	503	3	10.0240	246520	0.0400	2011
9	0.2760	14936	0.1360	6713	4	32.9540	774824	0.2040	8647
12	9.4120	409293	5.8840	184726	5	46.1380	1060321	0.3080	13622

Table 1. Results for different goal sizes (number of literals in the goals). The initial state is fixed and has 6 extensional atoms.

Size of initial state	Movie Store				Size of initial state	Health Care			
	STRIPS		*fSTRIPS*			*STRIPS*		*fSTRIPS*	
	CPU	Mem	CPU	Mem		CPU	Mem.	CPU	Mem.
20	9.2560	409293	5.8800	184726	3	0.148	5875	0.012	718
30	9.2600	409293	5.7440	184726	6	10.076	246519	0.04	2011
40	9.2520	409293	5.8000	184726	9	689.3750	9791808	0.124	5443
50	9.4120	409293	5.8840	184726	12	>1000	N/A	0.348	14832
60	9.3720	409293	5.8240	184726	18	>1000	N/A	0.94	38810

Table 2. Results for different sizes (number of facts) in initial states. The planning goal is fixed: 6 extensional literals in the "movie store" case and 3 extensional literals in the "health care" case.

The first rule describes an action of a user, X, buying a movie, M. The action is possible only if the movie has not already been purchased by somebody. The second rule says that, to play a movie for the first time, the user must buy it first and not have played it before. The third rule deals with the case when the user is happy and decides to keep the movie. The remaining rules are self-explanatory. $\Box$

A reachability query in a state modifying policy is a specification of a *target state* (usually an undesirable state), and the administrator typically wants to check if such a state is reachable by a sequence of actions. The target state specification consists of a set of literals, and the reachability query is naturally expressed as a planning problem. For instance, in Example 5, the second rule can be seen as a *STRIPS* action whose precondition is $\{bought(X, M) \otimes \neg played1(X, M)\}$ and the effect is $\{+played1(X, M)\}$. The initial and the target states in this example are sets of facts that describe the movies that have been bought, sold, and played by various customers.

The main difference between the two test cases is that the Health Care example has many actions and intensional rules, while the movie store case has only six actions and no intensional predicates. As seen from Tables 6 and 6, for the relatively simple Movie Store example, *fSTRIPS* is about twice more efficient both in time and space.[3] However, in the more complex Health Care example, *fSTRIPS* is at least two orders of magnitude better both time-wise and space-wise. While in the Movie Store example the statistics for the two strategies seem to grow at the same rate, in the Health Care case, the *fSTRIPS* time appears to grow linearly, while the time for *STRIPS* grows quadratically.

[3] Time is measured in seconds and memory in kilobytes.

7 Conclusion

This paper has demonstrated that the use of Transaction Logic accrues significant benefits in the area of planing. That is, the message is the benefits of $\mathcal{TR}$, not any particular planning heuristic. As an illustration, we have shown that sophisticated planning strategies, such as *STRIPS*, can be naturally represented in $\mathcal{TR}$ and that the use of this powerful logic opens up new possibilities for generalizations and devising new, more efficient algorithms. For instance, we have shown that once the *STRIPS* algorithm is cast as a set of rules in $\mathcal{TR}$, the framework can be extended, almost for free, to support such advanced aspects as action ramification, i.e., indirect effects of actions. Furthermore, by tweaking these rules just slightly, we obtained a new, much more efficient planner, which we dubbed *fSTRIPS* (fast *STRIPS*). These non-trivial insights were acquired merely due to the use of $\mathcal{TR}$ and not much else. The same technique can be used to cast even more advanced strategies such as *RSTRIPS*, *ABSTRIPS* [21], and HTN [20] as $\mathcal{TR}$ rules, and the *fSTRIPS* optimization straightforwardly applies to the first two.

There are several promising directions to continue this work. One is to investigate other planning strategies and, hopefully, accrue similar benefits. Other possible directions include non-linear plans and plans with loops [18, 17, 24]. For instance non-linear plans could be represented using Concurrent Transaction Logic [8], while loops are easily representable using recursive actions in $\mathcal{TR}$.

Acknowledgments. This work was supported, in part, by the NSF grant 0964196. We also thank the reviewers for valuable comments.

References

1. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. ACM Trans. Inf. Syst. Secur. 13(3), 20:1–20:28 (Jul 2010)
2. Bibel, W.: A deductive solution for plan generation. New Generation Computing 4(2), 115–132 (1986)
3. Bibel, W.: A deductive solution for plan generation. In: Schmidt, J.W., Thanos, C. (eds.) Foundations of Knowledge Base Management, pp. 453–473. Topics in Information Systems, Springer Berlin Heidelberg (1989)
4. Bibel, W., del Cerro, L.F., Fronhfer, B., Herzig, A.: Plan generation by linear proofs: On semantics. In: Metzing, D. (ed.) GWAI-89 13th German Workshop on Artificial Intelligence, Informatik-Fachberichte, vol. 216, pp. 49–62. Springer Berlin Heidelberg (1989)
5. Bonner, A., Kifer, M.: Transaction logic programming. In: Int'l Conference on Logic Programming. pp. 257–282. MIT Press, Budapest, Hungary (June 1993)
6. Bonner, A., Kifer, M.: Applications of transaction logic to knowledge representation. In: Proceedings of the International Conference on Temporal Logic. pp. 67–81. No. 827 in Lecture Notes in Artificial Inteligence, Springer-Verlag, Bonn, Germany (July 1994)
7. Bonner, A., Kifer, M.: Transaction logic programming (or a logic of declarative and procedural knowledge). Tech. Rep. CSRI-323, University of Toronto (November 1995), `http://www.cs.toronto.edu/~bonner/transaction-logic.html`
8. Bonner, A., Kifer, M.: Concurrency and communication in transaction logic. In: Joint Int'l Conference and Symposium on Logic Programming. pp. 142–156. MIT Press, Bonn, Germany (September 1996)

9. Bonner, A., Kifer, M.: A logic for programming database transactions. In: Chomicki, J., Saake, G. (eds.) Logics for Databases and Information Systems, chap. 5, pp. 117–166. Kluwer Academic Publishers (March 1998)
10. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science 133 (1994)
11. Cresswell, S., Smaill, A., Richardson, J.: Deductive synthesis of recursive plans in linear logic. In: Biundo, S., Fox, M. (eds.) Recent Advances in AI Planning, Lecture Notes in Computer Science, vol. 1809, pp. 252–264. Springer Berlin Heidelberg (2000)
12. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(34), 189 – 208 (1971)
13. Fodor, P., Kifer, M.: Tabling for transaction logic. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. pp. 199–208. PPDP '10, ACM, New York, NY, USA (2010)
14. Giunchiglia, E., Lifschitz, V.: Dependent fluents. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 1964–1969 (1995)
15. Guglielmi, A.: Concurrency and plan generation in a logic programming language with a sequential operator. In: Hentenryck, P.V. (ed.) ICLP. pp. 240–254. MIT Press (1994)
16. Hölldobler, S., Schneeberger, J.: A new deductive approach to planning. New Generation Computing 8(3), 225–244 (1990)
17. Kahramanogullari, O.: Towards planning as concurrency. In: Hamza, M.H. (ed.) Artificial Intelligence and Applications. pp. 387–393. IASTED/ACTA Press (2005)
18. Kahramanogullari, O.: On linear logic planning and concurrency. Information and Computation 207(11), 1229 – 1258 (2009), special Issue: 2nd International Conference on Language and Automata Theory and Applications (LATA 2008)
19. Lifschitz, V.: On the semantics of strips. In: Georgeff, M., Lansky, Amy (eds.) Reasoning about Actions and Plans, pp. 1–9. Morgan Kaufmann, San Mateo, CA (1987)
20. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
21. Nilsson, N.: Principles of Artificial Intelligence. Tioga Publ. Co., Paolo Alto, CA (1980)
22. Reiter, R.: Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. MIT Press, Cambridge, MA (2001)
23. Rezk, M., Kifer, M.: Transaction logic with partially defined actions. J. Data Semantics 1(2), 99–131 (2012)
24. Srivastava, S., Immerman, N., Zilberstein, S., Zhang, T.: Directed search for generalized plans using classical planners. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-2011). AAAI (June 2011)
25. Swift, T., Warren, D.: Xsb: Extending the power of prolog using tabling. Theory and Practice of Logic Programming (2011)
26. Thielscher, M.: Computing ramifications by postprocessing. In: IJCAI. pp. 1994–2000. Morgan Kaufmann (1995)
27. Thielscher, M.: Ramification and causality. Artificial Intelligence 89(12), 317 – 364 (1997)