State Space Planning Using Transaction Logic

Reza Basseda(✉) and Michael Kifer(✉)

Stony Brook University, Stony Brook, New York, NY 11794, USA
{rbasseda,kifer}@cs.stonybrook.edu

Abstract. State space planning algorithms have been considered as one of the main classical planning techniques to solve classical planning problems since 1960. In this paper, we show that Transaction Logic is an appropriate language and framework to study and compare these planning algorithms, which enables one to have more efficient planners in logic programming frameworks. Specifically, we take *STRIPS* planning and forward state space planning algorithms, and show that the specification of these algorithms in Transaction Logic lets one implement complicated planning algorithms in declarative programming languages (e.g. Prolog). We first provide a formal representation of these planning algorithms in Transaction Logic, which can be used to automatically translate *STRIPS* planning problems in PDDL to Transaction Logic rules. Then, we use the resulting Transaction Logic rules to solve planning problems and compare the performance of those algorithms in our simple interpreter implemented in XSB Prolog. We use several case studies to show how the linear *STRIPS* planning algorithm is faster than forward state space search. Our experiments highlight the fact that a planner implemented by logic programming framework can become faster if an appropriate planning algorithm is applied.

Keywords: Declarative planning algorithms · Planning in logic programming · State space planning

1 Introduction

The classical automated planning has been used in a wide range of applications such as robotics, multi-agent systems, and more. This wide range of applications has made automated planning one of the most important research areas in Artificial Intelligence (AI). The history of using logical deduction to solve classical planning problems in AI dates back to the late 1960s when situation calculus was applied in the planning domain [30]. There are several planners that encode planning problems into satisfiability problems [35][29][17] or constraint satisfaction problems (CSP) [47][18][2] and use logical deduction to solve the planning problems. Beside planning as satisfiability and CSP, a number of deductive planning frameworks have been proposed over the years. Linear connection proof method [6][7][8], equational horn logic [32], and linear logic [34][15]

This work was supported, in part, by the NSF grant 0964196.

are well-known examples of logic-based deduction methods applied for solving classical planning problem. Answer set programming is another, more recent logic based technique to solve planning problems [36][26][24][44].

There are several reasons that make logical deduction suitable to be used by a classical planner [45][16]: (1) Logical deduction used in planning can be cast as a formal framework that eases proving different planning properties such as completeness and termination. (2) Logic-based systems naturally provide a declarative language that simplifies the specification of planning problems. (3) Logical deduction is usually an essential component of intelligent and knowledge representation systems. Therefore, applying logical deduction in classical planning makes the integration of planners with such systems simpler. In-depth discussion of these reasons is beyond the scope of our paper. Despite the benefits of using logical deduction in planning, many of the above mentioned deductive planning techniques are not getting as much attention as algorithms specifically devised for planning problems. There are several reasons for this state of affairs:

– Many of the above approaches invent one-of-a-kind techniques that are suitable only for the particular problem at hand. For instance, the effects or preconditions of actions are sometimes encoded indirectly in answer set programming planners. This makes the encoding of planning problem difficult, and thus reduces the generality of this technique.
– These works generally show how they can represent and encode classical planning actions and rely on a theorem prover of some sort to find plans. Therefore, the planning techniques embedded in such planners are typically some of the simplest state space planning strategies (e.g. forward state space search) and they have extremely large search space. Consequently, they cannot exploit heuristics and techniques invented by different classical and neo-classical planning technique to reduce the search space.

In this paper, we show that a general logical theory, called *Transaction Logic* (or $\mathcal{TR}$) [12–14], addresses the above mentioned issues and also 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. To illustrate this point, [5] has shown how state space planning techniques, such as *STRIPS* (also known as goal-stack state space planning) and forward state space planning algorithms, can be naturally represented and improved upon using Transaction Logic. Since planning techniques are cast here as purely logical problems in a suitable general logic, a number of otherwise non-trivial further developments became low-hanging fruits and were gotten almost for free. In the present paper, based on the aforesaid representations of planning algorithms in Transaction Logic, we develop a simple translator that maps *STRIPS* planning problems (specified in PDDL [1]) and planning algorithms to Prolog programs. This technique makes many already existing Prolog based planners [4][49][3][48] more efficient. We emphasize that this paper, unlike [5], does not propose new planning algorithms in Transaction Logic. Instead, we use the sequential subset of Transaction Logic (i.e., without concurrent transactions) to represent the linear *STRIPS* planning algorithm [19], as suggested in [11]. A more computationally complex strategy was proposed in [5]. It deals with non-linear *STRIPS* and *Concurrent* Transaction Logic, and it was shown to be complete. The present paper, in contrast, just shows how planning algorithms written in Transaction Logic can be simply mapped into Prolog rules.

The next section briefly characterizes a planning problem and overviews Transaction Logic. Section 3 explains how we formally encode planning techniques in $\mathcal{TR}$. Section 4 also provides the results of our simple experiments to illustrate the practical applications of this method. Section 5 describes the relation of this work to PDDL and other research on planning with logic. The last section concludes our paper.

2 Characterization of a Planning Problem

In this section, we briefly remind the reader the basic concepts of logic and formally define an extended STRIPS planning problem. Then we briefly overview Transaction Logic ($\mathcal{TR}$) [11].

2.1 *STRIPS* Planning Problem

In a *STRIPS* planning problem, actions update the state of a system (e.g. Knowledge-Base): Facts may be inserted into or removed from the state as a result of execution of an action. We assume denumerable sets of variables $\mathcal{X}$, constants $\mathcal{C}$, and disjoint sets of predicate symbols, extensional ($\mathcal{P}_{ext}$) and intensional ($\mathcal{P}_{int}$) ones. A *term* is a variable or constant. Extensional (resp. intensional) ***Atoms*** have the form $p(t_1, ..., t_n)$, where t_i is a term and $p \in \mathcal{P}_{ext}$ (resp. $p \in \mathcal{P}_{int}$). A ***ground*** atom is a variable free atom. A ***literal*** is either an atom or a negated extensional atom, $\neg p(t_1, ..., t_n)$. Note that negative intensional atoms are not literals. A substitution θ is a set of expressions of the form $X \longleftarrow c$, where $X \in \mathcal{X}$ and $c \in \mathcal{C}$. Given a substitution θ, an atom $a\theta$ is obtained from atom a by replacing its variables with constants according to θ.

Intensional predicate symbols are defined by *rules*. A rule r, shown as $head(r) \leftarrow b_1 \wedge \cdots \wedge b_n$, consists of an intensional atom $head(r)$ in the head and a conditional body, a (possibly empty) conjunction of literals $b_1, \ldots, b_n$, where $b_i \in body(r)$. A ***ground instance*** of a rule, $r\theta$, is any rule obtained from r by a substitution of $head(r)$ and $body(r)$ with ground atoms $head(r)\theta$ and $body(r)\theta$ respectively. Given a set of literals $\mathbf{S}$ and a ground rule $r\theta$, the rule is *true* in $\mathbf{S}$ if either $head(r)\theta \in \mathbf{S}$ or $body(r)\theta \nsubseteq \mathbf{S}$. A (possibly non-ground) rule is *true* in $\mathbf{S}$ if all of its ground instances are true in $\mathbf{S}$. A ***fact*** is a ground extensional atom that can be inserted or deleted by *STRIPS* actions. A set $\mathbf{S}$ of literals is ***consistent*** if there is no atom, a, such that $\{a, \neg a\} \subseteq \mathbf{S}$.

Definition 1 (State). *Given a set of rules* $\mathbb{R}$*, a consistent set* $\mathbf{S}$ *of literals is called a* **state** *if and only if*

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

Definition 2 (*STRIPS* action). *A STRIPS* **action** *$\alpha = \langle p_\alpha(X_1, ..., X_n), Pre(\alpha), E(\alpha)\rangle$ consists of an intensional atom $p_\alpha(X_1, ..., X_n)$ in which $p_\alpha \in \mathcal{P}_{int}$ is a predicate that is reserved to represent the action α and can be used for no other purpose, a set of literals $Pre(\alpha)$, called the **precondition** of α, and a consistent set of extensional literals $E(\alpha)$, called the **effect** of α. The variables in $Pre(\alpha)$ and $E(\alpha)$ must occur in $\{X_1, ..., X_n\}$.*

Note that the literals in $Pre(\alpha)$ can be both extensional and intensional, while the literals in $E(\alpha)$ 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 θ 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 $\neg E = \{\neg\ell | \ell \in E\}$.*

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

Definition 4 (Planning problem). *Given a set of rules $\mathbb{R}$, a set of STRIPS actions $\mathbb{A}$, a set of literals G, called the **goal**, and an **initial state** $\mathbf{S}$, a* **planning solution** *(or simply a **plan**) for the planning $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ is a sequence of ground actions $\sigma = \alpha_1, \ldots, \alpha_n$ such that for each $1 \leq i \leq n$;*

- *there is a substitution θ_i and a STRIPS action $\alpha'_i \in \mathbb{A}$ such that $\alpha'_i\theta = \alpha_i$; 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);*
  - *α_i is executable in state $\mathbf{S}_{i-1}$ and the result of that execution is the state $\mathbf{S}_i$.*

2.2 Overview of Transaction Logic

To make this paper self-contained, we provide a brief introduction to the *subset* of Transaction Logic ($\mathcal{TR}$) [9,11–14] that are needed for the understanding of this paper.

As an extension of first-order predicate calculus, $\mathcal{TR}$ is sharing most of its syntax with the first-order predicate calculus' syntax. One of the new connectives that $\mathcal{TR}$ adds to the first-order predicate calculus is the ***serial conjunction***, denoted $\otimes$. It is a binary associative, non-commutative connective. The formula $\phi \otimes \psi$, showing a composite action, denotes an *execution* of ϕ followed by an execution of ψ. When ϕ and ψ are regular first-order formulas, $\phi \otimes \psi$ reduces to the usual first-order conjunction, $\phi \wedge \psi$. The logic also introduces other connectives to support hypothetical reasoning, concurrent execution, etc., but these are beyond the scope of this paper.

To take the *frame problem* out of many considerations in $\mathcal{TR}$, it has an extensible mechanism of ***elementary updates*** (see [10,11,13,14,42]). Due to the definition of *STRIPS* actions, we just need the following two types of elementary updates (actions): $+p(t_1, \ldots, t_n)$ and $-p(t_1, \ldots, t_n)$, where $p(t_1, \ldots, t_n)$ denotes an *extensional* atom. Given a state $\mathbf{S}$ and a *ground* elementary action $+p(a_1, \ldots, a_n)$, an execution of $+p(a_1, \ldots, a_n)$ 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. Apparently, if $p(a_1, \ldots, a_n) \in \mathbf{S}$, the action $+p(a_1, \ldots, a_n)$ has no effect, and similarly for $-p(a_1, \ldots, a_n)$.

We can define a ***complex action*** using ***serial rule*** that 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 denoting the complex action and b_1, ..., b_n are literals or elementary actions. That means 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. Note that we have regular first-order as well as serial-Horn rules. For simplicity, we assume that the sets of intensional predicates that can appear in the heads of regular rules and those in the heads of serial rules are disjoint. *Extensional atoms* and *Intensional atoms* compose state (see Definition 1) and will be collectively called ***fluents***. 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. All our examples use the standard logic programming convention whereby lowercase symbols represent constants and predicate symbols, while the uppercase symbols stand for variables that are universally quantified outside of the rules. It is common practice to omit such quantifiers.

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

Here *on*, *clear*, *tooHeavy*, and *weight* are fluents and the rest of atoms represent actions. The predicate *tooHeavy* is an intensional fluent, while *on*, *clear*, and *weight* are extensional fluents. The actions $+on(...)$, $-clear(...)$, and $-on(...)$ are elementary and the intensional 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.

$\mathcal{TR}$'s 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, ψ, 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:

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

Various inference systems for serial-Horn $\mathcal{TR}$ [11] are similar to the well-known SLD resolution proof strategy for Horn clauses plus some $\mathcal{TR}$-specific inference rules and axioms. Given a set of serial rules, $\mathbb{P}$, and a *serial goal*, ψ (i.e., a formula that has the form of a body of a serial rule such as (1), these inference systems prove statements of the form $\mathbb{P}, \mathbf{D} \cdots \vdash \psi$, called ***sequents***. A proof of a sequent of this form is interpreted as a proof that action ψ defined by the rules in $\mathbb{P}$ can be successfully executed starting at state $\mathbf{D}$.

An inference succeeds iff it finds an execution for the transaction ψ. The execution is a sequence of database states $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathbb{P}, \mathbf{D}\,\mathbf{D}_1 \ldots \mathbf{D}_n \models \psi$. We will use the following inference system in our planning application. For simplicity, we present 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* one needs to 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 $\mathbb{P}$.*

$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash body \otimes rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \tag{3}$$

2. *Querying the database: If* $\mathbf{D} \models t$ *then*

$$\frac{\mathbb{P}, \mathbf{D} \cdots \vdash rest}{\mathbb{P}, \mathbf{D} \cdots \vdash t \otimes rest} \tag{4}$$

3. *Performing elementary updates: If the elementary update t changes the state $\mathbf{D}_1$ into the state $\mathbf{D}_2$ then*

$$\frac{\mathbb{P}, \mathbf{D}_2 \cdots \vdash rest}{\mathbb{P}, \mathbf{D}_1 \cdots \vdash t \otimes rest} \tag{5}$$

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 to be sound and complete with respect to the model theory of $\mathcal{TR}$ [11]. 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 $\mathbb{P}, \mathbf{D}\, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash \phi$.

3 $\mathcal{TR}$ Planners

The informal encoding of *STRIPS* and forward state space planning as sets of $\mathcal{TR}$ rules first appeared in an unpublished report [11]. To use $\mathcal{TR}$ as a planning formalism, we formally show how a planning problem specification can be transformed into a set of $\mathcal{TR}$ rules that represent *STRIPS* and *Forward State Space* planning techniques. From now on, we call Forward State Space planning technique *naive planning*, as it is one of the simplest possible state space planning techniques. We also show that $\mathcal{TR}$ inference system uses those sets of $\mathcal{TR}$ rules to construct a plan. To highlight the correspondence between these sets of $\mathcal{TR}$ rules and original *STRIPS* and naive planning techniques, we first briefly review these planning techniques in terms of imperative pseudo codes.

The original *STRIPS* planning algorithm, proposed by [19], maintains a *stack* of goals and tries to achieve the goals from the top of the stack until the stack gets empty. We can simply implement this technique using recursive functions as depicted in Figure 1. Naive planning algorithm is based on *depth first search.* As illustrated in Figure 2, it starts from initial state, iteratively chooses actions, and moves to a new state until eventually finds a goal state.

The following definitions encode the aforesaid planning techniques as a set of $\mathcal{TR}$ rules.

Definition 6 (Enforcement operator). *Let G be a set of extensional literals. We define $Enf(G) = \{+p | p \in G\} \cup \{-p | \neg p \in G\}$. 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.

```
function STRIPS($\mathbb{R}, \mathbb{A}, \mathbf{S}, G$)
    $\sigma \leftarrow []$
    loop
        if $G \subseteq \mathbf{S}$ then
            return $\sigma$
        else
            $A \leftarrow \{\alpha\theta | \alpha \in \mathbb{A}, \theta(E(\alpha)) \subseteq G\}$
            if $A = \emptyset$ then
                reutrn failure
            else
                Choose non-deterministically $\alpha \in A$
                $\sigma' \leftarrow \mathit{STRIPS}(\mathbb{R}, \mathbb{A}, \mathbf{S}, \mathit{Pre}(\alpha))$
                if $\sigma' =$ failure then
                    reutrn failure
                else
                    $\mathbf{S} \leftarrow \mathit{exec}(\mathbf{S}, \sigma')$
                    $\mathbf{S} \leftarrow (\mathbf{S} \setminus \neg\theta(E(\alpha))) \cup \theta(E(\alpha))$
                    $\sigma \leftarrow [\sigma | \sigma' | \alpha\theta]$
                end if
            end if
        end if
    end loop
    return $\sigma$
end function
```

Fig. 1. *STRIPS* Planning

```
function NAIVE($\mathbb{R}, \mathbb{A}, \mathbf{S}_0, G$)
    $\mathbf{S} \leftarrow \mathbf{S}_0$
    $\sigma \leftarrow []$
    loop
        if $G \subseteq \mathbf{S}$ then
            return $\sigma$
        else
            $A \leftarrow \{\alpha\theta | \alpha \in \mathbb{A}, \theta(\mathit{Pre}(\alpha)) \subseteq \mathbf{S}\}$
            if $A = \emptyset$ then
                reutrn failure
            else
                Choose non-deterministically $\alpha \in A$
                $\mathbf{S} \leftarrow (\mathbf{S} \setminus \neg\theta(E(\alpha))) \cup \theta(E(\alpha))$
                $\sigma \leftarrow [\sigma | \alpha\theta]$
            end if
        end if
    end loop
    return $\sigma$
end function
```

Fig. 2. Naive Planning

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*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \ \otimes \ (\otimes_{u \in Enf(E(\alpha))} u). \tag{6}$$

Note that in (6) 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 give a set of $\mathcal{TR}$ clauses that simulates naive planning for *STRIPS* planning problems [19]. For convenience, we use $a \widehat{\otimes} b$ as a shorthand for $a \otimes b \vee b \otimes a$. This connective is called the *shuffle* operator in [11]. We define it to be commutative and associative and thus extend it to arbitrary number of operands.

Definition 8 (Naïve planning rules). *Given a STRIPS planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$ (see Definition 4), we define a set of $\mathcal{TR}$ rules, $\mathbb{P}(\Pi)$, which simulate naive planning technique to provide a planning solution to the planning problem. $\mathbb{P}(\Pi)$ has two parts, $\mathbb{P}_{general}$, $\mathbb{P}_{\mathbb{A}}$, described below.*

- *The $\mathbb{P}_{general}$ part: contains a couple of rules as follows;*

$$\begin{array}{l} plan \leftarrow . \\ plan \leftarrow execute\_action \otimes plan. \end{array} \tag{7}$$

  These rules construct a sequence of actions and bind them to the plan.
- *The $\mathbb{P}_{actions}$ part: for each $\alpha \in \mathbb{A}$, $\mathbb{P}_{actions}$ has a couple of rules as follows;*

$$\begin{array}{l} p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \ \otimes \ (\otimes_{u \in Enf(E(\alpha))} u). \\ execute\_action \leftarrow p_\alpha(\overline{X}). \end{array} \tag{8}$$

  This is the $\mathcal{TR}$ rule that corresponds to the action α, introduced in Definition 7 and generally links an action to a plan.

Given a *STRIPS* planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$, Definition 8 gives a set of $\mathcal{TR}$ rules that specify a naive planning strategy for that problem. To find a solution for that planning problem, one simply needs to place the request

$$?- plan \otimes (\wedge_{g_i \in G} g_i). \tag{9}$$

and use the $\mathcal{TR}$'s inference system to find a proof. 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

(3) 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 $\mathbf{D}_{i_r}$, $1 \le r \le n$, is the state at which α_{i_r} is applied. One can show that the pivoting sequence of actions generated from a deduction of (9) is a solution to the planning problem. *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. Based on the completeness of $\mathcal{TR}$'s inference system, one can show that the planner in Definition 8 is complete.

Definition 9 (*STRIPS* 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 simulate STRIPS planning technique to provide a planning solution to the 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*

$$achieve\_p(\overline{X}) \leftarrow \widehat{\otimes}_{i=1}^{n} achieve\_p_i(\overline{X}_i). \tag{10}$$

Rule (10) is an extension to the classical STRIPS planning algorithm. It captures intensional predicates and ramification of actions, and 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*

$$p_\alpha(\overline{X}) \leftarrow (\wedge_{\ell \in Pre(\alpha)} \ell) \ \otimes \ (\otimes_{u \in Enf(E(\alpha))} u). \tag{11}$$

  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*

$$\begin{aligned} achieve\_p(\overline{X}) &\leftarrow p(\overline{X}). \\ achieve\_not\_p(\overline{X}) &\leftarrow \neg p(\overline{X}). \end{aligned} \tag{12}$$

    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:*

$$achieve\_e(\overline{Y}) \leftarrow \neg e(\overline{Y}) \otimes execute\_p_\alpha(\overline{X}). \tag{13}$$

    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:*

$$execute\_p_\alpha(\overline{X}) \leftarrow (\widehat{\otimes}_{\ell \in Pre(\alpha)} achieve\_\ell) \otimes p_\alpha(\overline{X}). \tag{14}$$

  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:*

$$achieve_G \leftarrow (\widehat{\otimes}_{g_i=1}^{k} achieve\_g_i) \otimes (\wedge_{i=1}^{k} g_i). \tag{15}$$

Similar to naive planning, given a *STRIPS* planning problem $\Pi = \langle \mathbb{R}, \mathbb{A}, G, \mathbf{S}\rangle$, Definition 9 gives a set of $\mathcal{TR}$ rules. $\mathcal{TR}$'s inference system can use those rules to simulate *STRIPS* planning strategy for that problem. If one places the request

$$? - achieve_G\,. \tag{16}$$

the $\mathcal{TR}$'s inference system will find a proof. It can be shown that the pivoting sequence of actions generated from this proof is a solution to the planning problem. One can also show that the linear planner provided by Definition 9 is complete under the set of goal-serializable planning problems. We do not further discuss these issues in this paper due to space limitations. One can also show that with the help of existing tabling methods for $\mathcal{TR}$'s inference systems, *STRIPS* planner always terminates.

Definitions 8 and 9 are transforms that convert the specification of planning problems to $\mathcal{TR}$ rules. The similarity of $\mathcal{TR}$'s inference system in Definition 5 and well-known SLD resolution algorithm shows that one can use a similar approach to encode planning algorithms in logic programming frameworks. Based on Definitions 8 and 9, we can build a simple translator that constructs $\mathcal{TR}$ rules out of planning problem specifications in PDDL.

4 Experiments

In this section we briefly report on our experiments that compare naive and *STRIPS* planning. The test environment was a tabled $\mathcal{TR}$ interpreter [20] implemented in XSB and running on Intel®Xeon(R) CPU E5-1650 0 @ 3.20GHz 12 CPU, 64GB memory running on Mint Linux 14 64-bit. We use our translator to build $\mathcal{TR}$ rules out of PDDL files. We use the generated $\mathcal{TR}$ rules and our interpreter to solve our planning problems, which are test cases taken from [1]. We do not explain our test cases as they are well explained at *http://ipc.icaps-conference.org/*. Our test cases also can be found at *http://ewl.cewit.stonybrook.edu/planning/* along with our PDDL2TR translator, $\mathcal{TR}$ interpreter, and all the necessary items needed to reproduce the results. The tests highlight how the performance of the two planning techniques varies depending on the domain of application.

Table 1. Results for the Elevator test case (4 actions)

Test Case	Naive		*STRIPS*	
	CPU	Mem	CPU	Mem
s1-0	0	19	0	17
s2-0	0.004	277	0	96
s3-0	0.092	3636	0.02	853
s4-0	1.352	54628	0.1	4152
s5-0	24.213	867806	0.348	14148
s6-0	463.908	13440627	1.032	44681
s7-0	1000>	N/A	3.14	144060
s8-0	1000>	N/A	9.564	430435
s9-0	1000>	N/A	27.425	1212350
s10-0	1000>	N/A	74.24	3115811
s11-0	1000>	N/A	217.545	9074006
s12-0	1000>	N/A	546.606	21151356

Table 2. Results for the Travelling and Purchase Problem test case (3 actions)

Test Case	Naive		*STRIPS*	
	CPU	Mem	CPU	Mem
p01	0	22	0.004	121
p02	0.004	125	0.004	215
p03	0.024	664	0.016	878
p04	0.124	4040	0.056	2067
p05	41.43	2350601	0.592	19371

The main difference between the two test cases is that the Healthcare test case has many more actions and intensional rules than the movie store case. As seen from Tables 1 and 2, for both of these test cases, *STRIPS* planning gets to about two orders of magnitude more efficient both in time and space.[1] However, *STRIPS* is not able to solve problems that are not goal serializable.

We do not compare and analyse the performances of studied planning techniques in this paper as this study would be beyond the scope of this paper. The aim of our experiments is to provide show the differences between planning techniques in different application domains and to illustrate the ability of $\mathcal{TR}$ to not only provide a theoretical framework for analysis of planning techniques, but also to implement such techniques in a declarative way. Moreover, our experiments also show that $\mathcal{TR}$ simplifies and eases the implementation of complicated planning techniques, such as *STRIPS*.

5 Compatibility with PDDL and Related Work

The representation of planning algorithms in $\mathcal{TR}$ enables us to develop a simple translator that constructs $\mathcal{TR}$ rules out of planning problems and algorithms.

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

PDDL is a standard language intended to express planning problems [28] in AIPS planning competitions [1]. Planning problems from AIPS planning competitions are usually considered as standard benchmarks for planners. Therefore, providing an automated method of translating PDDL planning domains shows the generality of our approach.

A planning problem consists of domain predicates, possible actions, the structure of compound actions, and the effects of actions. To express a planning problem, PDDL supports several syntactic features such as basic *STRIPS*-style actions, conditional effects, universal quantifications in the effects, ADL features [41], domain axioms, safety constraints, hierarchical actions, and more. The formalism in Section 2 also supports the basic *STRIPS*-style actions and domain axioms. Clearly, it is simple to extend this formalism to include other features. For instance, it is easy to show that $\mathcal{TR}$ is can represent most of the features provided by different extensions of PDDL. The following list briefly shows how $\mathcal{TR}$ can express some of these main features.

- ADL features [28,37]: in PDDL, actions can have a first order formula in their precondition. The effect of an action can also include universal quantifications over fluents. $\mathcal{TR}$ can use Lloyd-Topor transformation to support first order formula (including universal and existential quantifiers and disjunction) in the precondition of actions. It also can simulate universal quantifications over fluents in the effects of actions.
- Numerical extensions [22]: in PDDL, one can associate actions, objects, and plans with numeric costs and use these costs in numerical expressions to compute different planning metrics. This syntactical feature also needs PDDL to include numerical operators. Since $\mathcal{TR}$ can express and encode numerical operators and expressions as a part of its model theory, it can easily handle this feature.
- Temporal extensions and durative actions [22,25,38,43]: PDDL is able to express discretised and continuous actions [22]. $\mathcal{TR}$ is also able to represent discretised and continuous actions because the notion of *time* can be encoded in $\mathcal{TR}$'s transactions.
- Plan and solution preferences and constraints [27,31]: In a planning problem, it is possible that only a subset of goals can be achieved because of the conflict between goals. In this situation, the ability to assign importance and preferences to different goals is essential. PDDL provides such ability to express such preferences among goals and planning solutions. $\mathcal{TR}$ augmented with defeasible reasoning [21] also can easily provide this feature.

Answer set programming is one of the leading logic-based planning techniques [36][24][44]. However, encoding a planning problem in answer set programming requires the addition of inertia axioms to solve frame problem [23]. Clearly, $\mathcal{TR}$-based planning does not face this problem. Our PDDL2TR translator also shows that $\mathcal{TR}$-based planning is general enough to automatically encode planning problems. Since $\mathcal{TR}$'s model theory also avoids the frame problem and no inertia axioms are required in $\mathcal{TR}$'s planning rules. Picat is another logic programming framework that relies on tabling. It has been shown to be an efficient logic-based system for solving planning problems [4][49][3][48]. However, the techniques employed by Picat are orthogonal to the results presented here and, we believe, this work provides a natural direction for incorporation of complex planning algorithms, like *STRIPS*, into Picat.

6 Conclusion

This paper has demonstrated that Transaction Logic can bridge the gap between AI planning and logic programming. Specially, we claim that $\mathcal{TR}$ is a general framework for analysis and implementation in the area of planning, which does not depend on any particular planning strategy.

As an illustration, we have shown that different planning strategies, such as *STRIPS*, not only can be naturally represented in $\mathcal{TR}$, but that also such representations can be used to automatically translate planning problems and algorithms into declarative programming languages (e.g. Prolog). We have also shown that the use of this powerful logic opens up new possibilities for improvement of existing planning methods in logic programming. For instance, we have shown that the sophisticated *STRIPS* algorithm can be cast as a set of rules in $\mathcal{TR}$, which shows the ability of rule based systems to represent such planning techniques.

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 *ABSTRIPS* [40], and HTN [39] as $\mathcal{TR}$ rules, and those techniques can straightforwardly be used to solve planning problems in logic programming frameworks.

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 heuristics and plans with loops [33,34,46]. For instance loops are easily representable using recursive actions in $\mathcal{TR}$.

References

1. Bacchus, F.: AIPS-00 planning competition, May 2000. http://www.cs.toronto.edu/aips2000
2. Barták, R., Toropila, D.: Solving sequential planning problems via constraint satisfaction. Fundam. Inform. **99**(2), 125–145 (2010). http://dx.doi.org/10.3233/FI-2010-242
3. Barták, R., Zhou, N.-F.: On modeling planning problems: experience from the petrobras challenge. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013, Part II. LNCS, vol. 8266, pp. 466–477. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-45111-9_40
4. Barták, R., Zhou, N.: Using tabled logic programming to solve the petrobras planning problem. TPLP **14**(4–5), 697–710 (2014). http://dx.doi.org/10.1017/S1471068414000295

5. Basseda, R., Kifer, M., Bonner, A.J.: Planning with transaction logic. In: Kontchakov, R., Mugnier, M.-L. (eds.) RR 2014. LNCS, vol. 8741, pp. 29–44. Springer, Heidelberg (2014)
6. Bibel, W.: A deductive solution for plan generation. New Generation Computing **4**(2), 115–132 (1986)
7. Bibel, W.: A deductive solution for plan generation. In: Schmidt, J.W., Thanos, C. (eds.) Foundations of Knowledge Base Management. Information Systems, pp. 453–473. Springer, Heidelberg (1989)
8. 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, Heidelberg (1989)
9. Bonner, A., Kifer, M.: Transaction logic programming. In: Int'l Conference on Logic Programming, pp. 257–282. MIT Press, Budapest, June 1993
10. Bonner, A.J., Kifer, M.: Applications of transaction logic to knowledge representation. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL1994. LNCS, vol. 827, pp. 67–81. Springer, Heidelberg (1994)
11. 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
12. 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, September 1996
13. Bonner, A., Kifer, M.: A logic for programming database transactions. In: Chomicki, J., Saake, G. (eds.) Logics for Databases and Information Systems, ch. 5, pp. 117–166. Kluwer Academic Publishers, March 1998
14. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science **133** (1994)
15. 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, Heidelberg (2000)
16. Dovier, A., Formisano, A., Pontelli, E.: Perspectives on logic-based approaches for reasoning about actions and change. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS, vol. 6565, pp. 259–279. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-20832-4_17
17. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic sat-compilation of planning problems. In: Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence, IJCAI 1997, vol. 2, pp. 1169–1176. Morgan Kaufmann Publishers Inc., San Francisco (1997). http://dl.acm.org/citation.cfm?id=1622270.1622325
18. Erol, K., Hendler, J.A., Nau, D.S.: UMCP: a sound and complete procedure for hierarchical task-network planning. In: Hammond, K.J. (ed.) AAAI 1994, pp. 249–254. University of Chicago, Chicago (1994). http://www.aaai.org/Library/AIPS/1994/aips94-042.php
19. 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)
20. Fodor, P., Kifer, M.: Tabling for transaction logic. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP 2010, pp. 199–208. ACM, New York (2010)

21. Fodor, P., Kifer, M.: Transaction logic with defaults and argumentation theories. In: Gallagher, J.P., Gelfond, M. (eds.) Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011, July 6–10, 2011, Lexington, Kentucky, USA. LIPIcs, vol. 11, pp. 162–174. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). http://dx.doi.org/10.4230/LIPIcs.ICLP.2011.162
22. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. J. Artif. Int. Res. **20**(1), 61–124 (2003). http://dl.acm.org/citation.cfm?id=1622452.1622454
23. Gebser, M., Kaminski, R., Knecht, M., Schaub, T.: plasp: A prototype for PDDL-based planning in ASP. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 358–363. Springer, Heidelberg (2011)
24. Gebser, M., Kaufmann, R., Schaub, T.: Gearing up for effective ASP planning. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning. LNCS, vol. 7265, pp. 296–310. Springer, Heidelberg (2012). http://dl.acm.org/citation.cfm?id=2363344.2363364
25. Geffner, H.: Pddl 2.1: Representation vs. computation. J. Artif. Int. Res. **20**(1), 139–144 (2003). http://dl.acm.org/citation.cfm?id=1622452.1622457
26. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. **2**, 193–210 (1998). http://www.ep.liu.se/ej/etai/1998/007/
27. Gerevini, A., Haslum, P., Long, D., Saetti, A., Dimopoulos, Y.: Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. Artif. Intell. **173**(5–6), 619–668 (2009). http://dx.doi.org/10.1016/j.artint.2008.10.012
28. Ghallab, M., Isi, C.K., Penberthy, S., Smith, D.E., Sun, Y., Weld, D.: PDDL - The Planning Domain Definition Language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212
29. Giunchiglia, E., Massarotto, A., Sebastiani, R.: Act, and the rest will follow: exploiting determinism in planning as satisfiability. In: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI 1998/IAAI 1998, pp. 948–953. American Association for Artificial Intelligence, Menlo Park (1998). http://dl.acm.org/citation.cfm?id=295240.295931
30. Green, C.: Application of theorem proving to problem solving. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence, IJCAI 1969, pp. 219–239. Morgan Kaufmann Publishers Inc., San Francisco (1969). http://dl.acm.org/citation.cfm?id=1624562.1624585
31. Gregory, P., Long, D., Fox, M.: Constraint based planning with composable substate graphs. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) Proceedings of ECAI 2010–19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16–20, 2010. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 453–458. IOS Press (2010). http://dx.doi.org/10.3233/978-1-60750-606-5-453
32. Hölldobler, S., Schneeberger, J.: A new deductive approach to planning. New Generation Computing **8**(3), 225–244 (1990)
33. Kahramanogullari, O.: Towards planning as concurrency. In: Hamza, M.H. (ed.) Artificial Intelligence and Applications, pp. 387–393. IASTED/ACTA Press (2005)
34. 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)

35. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence, ECAI 1992, pp. 359–363. John Wiley & Sons Inc., New York (1992). http://dl.acm.org/citation.cfm?id=145448.146725
36. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(12), 39–54 (2002). http://www.sciencedirect.com/science/article/pii/S0004370202001868. knowledge Representation and Logic Programming
37. McDermott, D.V.: The 1998 AI planning systems competition. AI Magazine **21**(2), 35–55 (2000). http://www.aaai.org/ojs/index.php/aimagazine/article/view/1506
38. McDermott, D.V.: PDDL2.1 - the art of the possible? commentary on fox and long. J. Artif. Intell. Res. (JAIR) **20**, 145–148 (2003). http://dx.doi.org/10.1613/jair.1996
39. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
40. Nilsson, N.: Principles of Artificial Intelligence. Tioga Publ. Co., Paolo Alto (1980)
41. Pednault, E.P.D.: Adl: Exploring the middle ground between strips and the situation calculus. In: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, pp. 324–332. Morgan Kaufmann Publishers Inc., San Francisco (1989). http://dl.acm.org/citation.cfm?id=112922.112954
42. Rezk, M., Kifer, M.: Transaction logic with partially defined actions. J. Data Semantics **1**(2), 99–131 (2012)
43. Smith, D.E.: The case for durative actions: A commentary on PDDL2.1. J. Artif. Intell. Res. (JAIR) **20**, 149–154 (2003). http://dx.doi.org/10.1613/jair.1997
44. Son, T.C., Baral, C., Tran, N., Mcilraith, S.: Domain-dependent knowledge in answer set planning. ACM Trans. Comput. Logic **7**(4), 613–657 (2006). http://doi.acm.org/10.1145/1183278.1183279
45. Son, T.C., Pontelli, E., Sakama, C.: Logic programming for multiagent planning with negotiation. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 99–114. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-02846-5_13
46. 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
47. Stefik, M.J.: Planning with Constraints. Ph.D. thesis, Stanford, CA, USA (1980). aAI8016868
48. Zhou, N., Dovier, A.: A tabled prolog program for solving sokoban. In: IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7–9, 2011, pp. 896–897. IEEE Computer Society (2011). http://dx.doi.org/10.1109/ICTAI.2011.145
49. Zhou, N., Dovier, A.: A tabled prolog program for solving sokoban. Fundam. Inform. **124**(4), 561–575 (2013). http://dx.doi.org/10.3233/FI-2013-849