Constraint based implementation of a PDDL-like language with static causal laws and time fluents

Agostino Dovier[1] and Jacopo Mauro[2]

[1] Dipartimento di Matematica e Informatica, Università di Udine
dovier@dimi.uniud.it
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna
jmauro@cs.unibo.it

Abstract. Planning Domain Definition Language (PDDL) is the most used language to encode and solve planning problems. In this paper we propose two PDDL-like languages that extend PDDL with new constructs such as static causal laws and time fluents with the aim of improving the expressivity of PDDL language. We study the complexity of the main computational problems related to the planning problem in the new languages. Finally, we implement a planning solver using constraint programming in GECODE that outperforms the existing solvers for similar languages.

1 Introduction

In the context of knowledge representation and reasoning, a very important application of logic programming within artificial intelligence is that of developing languages and tools for reasoning about actions and change and, more specifically, for the problem of planning [2]. The proposals on representing and reasoning about actions and change have relied on the use of concise and high-level languages, commonly referred to as *action description languages*. Some well-known examples include the languages $\mathcal{A}$ and $\mathcal{B}$ [11] and extensions like $\mathcal{K}$ [7]. Action languages allow one to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system. An *action description* is a specification of a planning problem using the action language.

Since 1998 a declarative language for planning has been defined either for establishing a common syntax or for allowing different research groups to participate to the planning competitions. This language is known as PDDL and its last release is 3.1 (see [16, 10, 12] for information on planning competitions and PDDL).

The goal of this work is to build two languages on the top of PDDL, called $APDDL$ and $BPDDL$, allowing new constructs and then explore the relevance of constraint solving for handling them. The main ideas of the constraint encoding comes from [6]. However, here we employed the C++ constraint solver platform GECODE [1] which is faster than the constraint solver of SICStus

Prolog used in [6], and we solve more precisely the frame problem. Moreover, we define a front-end from the PDDL-like languages to GECODE. We always outperform the running time of [6]; in some cases the improvements are really sensible.

The presentation is organized as follows. In Section 2 we introduce the language APDDL and in Section 3 we formally define its semantics. In Section 4 we define the language BPDDL. In Section 5 we report the complexities of some interesting problems related to planning in this language. The solver implementation and the tests are then discussed in Sections 6 and 7. Some proofs are reported in Appendix.

2 The language APDDL

APDDL is an extension of the well-known language PDDL and every APDDL program consists of two parts: the *domain definition* used to model the planning problem, and the *instance definition* used to define the instance of the problem to solve. We need to define a set $\mathcal{F}$ of fluent names. Each $f \in \mathcal{F}$ is assigned to a domain $\mathsf{dom}(f)$. We also need to define a set $\mathcal{A}$ of action names. Each action a is associated to a precondition $\mathsf{pre}(a)$ and an effect $\mathsf{eff}(a)$ all expressed as a Boolean combinations of arithmetic constraints on fluents (see Table 1 for a simplified syntax[3]).

`C`	::=	$0 \mid 1 \mid (\texttt{not}\ C) \mid (\texttt{and}\ C^+) \mid (\texttt{or}\ C^+) \mid (\texttt{AOP}\ \texttt{AC}\ \texttt{AC}) \mid$ $(\texttt{eqv}\ C\ C) \mid (\texttt{imp}\ C\ C) \mid (\texttt{xor}\ C\ C) \mid \texttt{TIME\_FLUENT}$
`AC`	::=	$n \mid \texttt{TIME\_FLUENT} \mid (\texttt{OP}\ \texttt{AC}\ \texttt{AC})$
`AOP`	::=	$> \mid \geq \mid < \mid \leq \mid = \mid \neq$
`OP`	::=	$+ \mid - \mid * \mid / \mid \texttt{mod} \mid \texttt{rem}$
`TIME_FLUENT`	::=	$f \mid (\texttt{at}\ n\ f)$

Table 1. Abstract syntax of constraints (C)—$n \in \mathbb{Z}, f \in \mathcal{F}$

The concrete syntax of the language is described by a EBNF grammar available at [15]. We just give here a taste of the syntax using a simple example: the famous Sam Lloyd's n-puzzle that consists of a frame of numbered square tiles in random order with one tile missing. The tiles should be arranged in increasing order, with the hole in the bottom right corner. Types can be used for differentiating objects. In this example we can define a type for the position of a tile, one for the direction of the move to do and one for the numbers on the tile. This can be done in the following way:

```
(:types positions directions tiles_numbers)
```

[3] where `eqv`, `imp`, `xor`, `mod`, `rem` are respectively the equivalence, implication, exclusive or, module and reminder operators

We can associate names (also known, with abuse of terminology, as constants) to constant and function symbols (with arity 0 and greater than 0, respectively). We can define the function symbol `near` with arity 2 used to determine what position should occupy a tile moved from a position `pos` according to a particular direction `dir`.

```
(near ?pos - positions ?dir - directions) - positions
```

In the example we can consider a missing tile as a tile numbered 0. We thus define the constant `empty_tile_number` for representing it.

```
empty_tile_number - tiles_numbers
```

We use these preliminary definitions to encode the relevant proprieties of the objects that we want to consider. These properties are the *fluents* and they are represented by a (multivalued) function. Boolean functions are called predicates. In our example we are interested in knowing the number in a particular position. This multi-valued fluent can be defined in the following way:

```
(has ?position - positions) - tiles_numbers
```

The last ingredient for the domain definitions are the definitions of the possible effects of the actions given their preconditions. In this example there is only an action: moving a tile. This action can be defined as follows[4]:

```
(:action move
        :parameters (?from ?to - positions)
        :precondition (and
                        (exists ?direction - directions
                                (== (near ?from ?direction) ?to)
                        )
                        (== (has ?to) empty_tile_number)   )
        :effect (and
                        (== (has ?from) empty_tile_number)
                        (== (has ?to) (at -1 (has ?from))))
)
```

Let us suppose that we want to solve this problem in a 3×3 board where the missing tile is at the bottom right corner. This can be encoded into the problem definition. First, all the objects involved are listed. In this case we have the following three types of objects[5]:

```
(:objects
        (set 1 9) - positions
        Left Right Up Down - directions
        (set 0 8) - tiles_numbers
)
```

[4] The term `(at -1 (has ?from)` is a time fluent and it will be described later

[5] `set` is an APDDL operator for defining set of integers in a concise way

Second, all the constants should be instantiated.

```
(:constants
        (== empty_tile_number 0)
        (== (near 1 Right) 2) (== (near 1 Down) 4)
        (== (near 2 Right) 3) ...
)
```

The problem definition includes also the definition of the values of the fluents in the initial state and the goal. In the problem we are considering this can be done in the following way:

```
(:init (== (has 1) 2) (== (has 2) 5) ... )
(:goal (== (has 1) 1) (== (has 2) 2) ... )
```

We impose that the fluents in the initial state are completely defined.

If the goal of the problem is to obtain an optimal plan it is possible to define a cost function. This function referred as *metric* has the following syntax:

```
    M ::=  (:metric MOP MC)
  MOP ::=  minimize | maximize
   MC ::=  AC | (is_violated C) | (OP MC MC)
```

where AC and C are defined as in Table 1. For example, suppose that a state has cost 0 if the number 1 is in the first row and 1 otherwise. If we want to minimize the cost, the metric can be defined in the following way:

```
(:metric minimize
       (is_violated (or (== (has 1) 1) (== (has 2) 1) (== (has 3) 1))))
```

Finally, at the end of the problem definition, it is necessary to specify the length of the plan we want to obtain. This can be done using the length primitive:

```
(:length 18)
```

The language APDDL has few more features like the possibility to introduce a metric function to maximize or minimize and additional constraints called *plan constraints*. For example in the n-puzzle problem it is possible to state that the tile with the number 1 should be in the last line at least once:

```
(:constraints (sometimes (or (== (has 7) 1)
                             (== (has 8) 1)
                             (== (has 9) 1))))
```

The main difference between the PDDL language and the APDDL is the possibility to use more operators in the action precondition and effects (division, remainder, exclusive or, ...) and the notion of time fluent.
A *time fluent* is an expression of the form $(\texttt{at}\ i\ f)$ where $i \in \mathbb{Z}$ is an integer and f is a fluent. This construct is used in actions for referring to the value of a fluent f in time instant i. If $i = 0$ then $(\texttt{at}\ 0\ f)$ (or, in short, f) is called present

fluent because it refers to the value of the fluent f in the current state. If $i < 0$ (resp. $i > 0$) the term ($\mathtt{at}\ i\ f$) is called instead past fluent (resp. future fluent). Let us observe that if the time fluent is used in an action precondition it refers to the state at which the action is executed. If it is used in the effect it refers to the state produced by the execution of the action.

An example of the use of time fluents is the following action that decreases the number of objects in a barrel in the next two states if during the last two state transitions at least one object is added into the barrel.

```
(:action empty
    :parameters (?barrel - barrel)
    :precondition (and
        (> (contains ?barrel) (at -1 (contains ?barrel)))
        (> (at -1 (contains ?barrel)) (at -2 (contains ?barrel))))
    :effect (and
        (== (contains ?barrel) (- (at -1 (contains ?barrel)) 1))
        (== (at 1 (contains ?barrel)) (- (contains ?barrel) 1)))
)
```

In goal constraints, plan constraints and metrics it is not possible to use past or present fluents while in init constraint it is not possible to use past fluents.

3 APDDL Semantics

Given an APDDL program P it is possible to obtain an equivalent ground instance $ground(P)$ by grounding all variables with all constants satisfying the types. In $ground(P)$ actions preconditions and effects, goal conditions and all the information on the initial state are (Boolean combinations of) Finite Domain constraints on time fluents.

A state is characterized by the values of all the fluents involved. We will use the term $\mathsf{val}(s, f)$ to denote the value of the fluent f in the state s. Any action a is characterized by its preconditions $\mathsf{pre}(a)$ and its effects $\mathsf{eff}(a)$. We allow parallel executions of different actions a_1 and a_2 provided their effects are independent. We impose a strong syntactic requirement: the sets of time fluents occurring in $\mathsf{eff}(a_1)$ and $\mathsf{eff}(a_2)$ must be disjoint.

Let us consider a sequence of states $s_0, s_1, \ldots, s_n$, and a constraint c. With $\mathsf{shift}_i(c)$ we denote the constraint obtained replacing each time fluent ($\mathtt{at}\ t\ f$) with the value $\mathsf{val}(s_{t+i}, f)$. If $t + i < 0$ or $t + i > n$ then the value is $\perp$ (undefined). Let us observe that $c' = \mathsf{shift}_i(c)$ is a Boolean combination of ground arithmetic constraints (or $\perp$). If $\perp$ occurs in it, then its value is false. Otherwise, its value is determined by the usual semantics of arithmetic and Boolean operators on ground formulas. If the value of c' is true, we say that $s_0, s_1, \ldots, s_n \models c'$, otherwise $s_0, s_1, \ldots, s_n \not\models c'$.

Let G be the set of *goal constraints* and I be the set of *initial constraints*. Then a plan of length n ($n \geq 0$) is a sequence $s_0, A_1, s_1, \ldots, A_n, s_n$ where

1. $s_0, \ldots, s_n$ are states

2. $A_1, \ldots, A_n$ are (possibly empty) sets of actions
3. $\forall i \in \{1, \ldots, n\} \forall a \in A_i \, . s_0 \ldots s_n \models \mathsf{shift}_{i-1}(\mathsf{pre}(a))$
4. $\forall i \in \{1, \ldots, n\} \forall a \in A_i \, . s_0 \ldots s_n \models \mathsf{shift}_{i}(\mathsf{eff}(a))$
5. $\forall c \in G \, . s_0 \ldots s_n \models \mathsf{shift}_{n}(c)$
6. $\forall c \in I \, . s_0 \ldots s_n \models \mathsf{shift}_{0}(c)$
7. $\forall a_1 \in A_i \forall a_2 \in A_i \, . a_1 \neq a_2$ $\mathsf{eff}(a_1)$ and $\mathsf{eff}(a_2)$ do not share future or present fluents.
8. if no action executed refers to a fluent f in s_i $(i > 0)$ then $\mathsf{val}(s_i, f) = \mathsf{val}(s_{i-1}, f)$ (inertia condition)

When further *plan constraints* are used, the plan definition must entail more constraints. For instance, if the constraint `(sometimes c)` is added, then there must be i such that $s_0 \ldots s_n \models \mathsf{shift}_i(c)$.

4 BPDDL

Starting from the language APDDL we add the possibility to use a construct like the static causal laws (briefly denoted here as *rules*) introduced in the language $\mathcal{B}$ [11], obtaining the new language BPDDL. A rule has a precondition and an effect similar to the action preconditions and effects, but without future fluents. Informally, the semantics of a rule is that at every state of the plan if the precondition is true then also the effect must be true.

Rules are more powerful than PDDL axioms [18]. As a matter of fact, differently from axioms, rules do not require to define predicates using only stratified programs (and this is a strong constraint for knowledge representation); moreover, they are allowed to change values of fluents that can be also used in action effects.

The possibility of using rules increases the expressiveness of the language. For instance it is possible to change a fluent value after a transition even if no action has been executed. A simple example is the implementation of a clock simply using a fluent and a rule

```
(:rule
    :parameters ( ?time - time)
    :effect (== ?time (+ (at -1 time) + 1))
)
```

Another interesting use of rules is the propagation of an action effect. Let us consider for example a colored directed graph where we want that all the nodes connected with edges in a set E_1 have the same color. This propriety can be encoded in the following way:

```
(:rule
    :parameters ( ?edge - edge)
    :precondition (is_edge_in_E_1 ?edge)
    :effect (== (node_colour (head ?edge))
                (node_colour (tail ?edge)))
)
```

When clear from the context, we will use the abstract notation $c_1 \rightarrow c_2$ for rules.

4.1 BPDDL Semantics and Inertia

Dealing with inertia in presence of rules is obviously more difficult than in a language that does not allow them. This is particularly true if the implementation of a language is based on the notion of constraint. Two rules stating the implications $p \rightarrow q$ and $q \rightarrow p$ are satisfied either by $p = q = 0$ or by $p = q = 1$. However, an arbitrary change of the values from 0 to 1 or vice versa cannot be simply justified by these rules.

A simple attempt of solution considered is that of choosing the states with a minimum change of fluents. Unfortunately, this definition cuts off a lot of solutions, as already pointed out in [3].

We instead used a solution based on the following principle: *"Given some action effects if something can be left unchanged then it must be unchanged"*

Let us define with $\mathsf{Act}(s_0, A_1, s_1, \ldots, A_n, s_n)$ the fluents of s_n that can be modified as a direct effect of an action in $\bigcup A_i$. Suppose that $\triangle F(s, s')$ is the set of fluents that have different values between the state s and s'. Now, in a plan $s_0, A_1, s_1, \ldots, A_n, s_n$ we say that there is a *critical situation* between s_{i-1} and s_i if there is a sequence $s_0, A_1, s_1, \ldots, s_{i-1}, A_i, s'$ where the state s':

- entails all the rules
- $\triangle F(s_{i-1}, s') \subset \triangle F(s_{i-1}, s_i)$
- $\forall f \in \mathsf{Act}(s_0, A_1, s_1, \ldots, A_i, s_i)\,.\,\mathsf{val}(s_i, f) = \mathsf{val}(s', f)$

Intuitively when there is a critical situations there is at least a fluent that can remain the same but instead has been changed by a rule. Therefore when there is a critical situation in a plan the over mentioned principle is violated.

Just a comment on the past references in rules. If a rule refers to a value of a fluent prior to the initial state s_0 of a plan the rule is trivially satisfied.

The semantics of the language BPDDL is similar to the semantics of APDDL with only two further requirements:

- the inertia condition is now the absence of critical conditions between two consecutive states in the plan
- the states must entail the applicable rules

5 Complexity

In this section we study the complexity of the main computational problems related to planning expressed within the APDDL and BPDDL languages. In particular, we focus our attention to ground APDDL/BPDDL programs. For APDDL programs some of the problems are equivalent, other are simpler or not meaningful. We assume moreover that no *plan constraints* is used in the program. Their inclusion would make the proofs more complicated but they would not affect the results. We studied the complexity of the following decision problems:

1. has_critical_situation (APDDL: not meaningful. BPDDL: NP-complete)
   input: a program, a sequence of states and actions $s_0, A_1, s_1, \ldots, A_n, s_n$ and two consecutive states s_i, s_{i+1} that entail all the conditions in the plan definition except the inertia
   output: 1 iff there is a critical situation between s_i, s_{i+1}, otherwise 0
2. validity (APDDL: P; BPDDL: co-NP-complete)
   input: a program and a sequence of states and actions $s_0, A_1, s_1, \ldots, A_n, s_n$
   output: 1 iff $s_0, A_1, s_1, \ldots, A_n, s_n$ is a plan, otherwise 0
3. k-plan (APDDL: NP-complete; BPDDL: Σ_2^P-complete)
   input: a program
   output: 1 iff there is a plan of length k that solves the problem encoded into the BPDDL program, otherwise 0
4. plan: (APDDL and BPDDL: PSPACE complete)[6]
   input: a program
   output: 1 iff there is a plan that solves the problem encoded into the BPDDL program, otherwise 0

We give here only the main ideas used. The complete proofs of the results are reported in Appendix.

The proof of NP-hardness of *has_critical_situation* is based on a reduction from a variant of SAT in which all false and all true assignments are forbidden. Let us consider the Boolean formula $\varphi = (X \vee Z) \wedge (\neg X \vee \neg Y \vee \neg Z)$ and consider the following program based on three rules with fluents f_X, f_Y, f_Z ($\oplus$ stands for exclusive or while f_W^{-1} for the past fluent ($\mathtt{at}\ -1\ f_W$)):

$$\mathsf{true} \rightarrow f_X \vee f_Z \vee (f_X \wedge f_Y \wedge f_Z) \vee (\neg f_X \wedge \neg f_Y \wedge \neg f_Z)$$
$$\mathsf{true} \rightarrow \neg f_X \vee \neg f_Y \vee \neg f_Z \vee (f_X \wedge f_Y \wedge f_Z) \vee (\neg f_X \wedge \neg f_Y \wedge \neg f_Z)$$
$$\mathsf{true} \rightarrow (f_X^{-1} \oplus f_X) \vee (f_Y^{-1} \oplus f_Y) \vee (f_Z^{-1} \oplus f_Z)$$

Let us consider now two states s_0 and s_1, where for every fluent f in $\{f_X, f_Y, f_Z\}$ it holds that $\mathsf{val}(s_0, f) = 0$ and $\mathsf{val}(s_1, f) = 1$. And, let us analyze the problem: is there a critical situation between s_0, s_1? The first two rules are satisfied if all the fluents are true or all are false or if there is an assignment that satisfies the formula φ. The last rule, instead, forces at least one fluent to change.

Let us observe that φ is not satisfiable by a trivial assignment [7]. One of the possible non trivial assignments that satisfy φ is instead $\{X/\mathsf{true}, Y/\mathsf{false}, Z/\mathsf{false}\}$. Using this assignment we can define a state s' such that $\mathsf{val}(s', f_X) = 1, \mathsf{val}(s', f_Y) = 0, \mathsf{val}(s', f_Z) = 0$ that satisfies all the rules and with fluent variations included w.r.t. those between s_0 and s_1. Therefore there is a critical situation.

[6] APDDL and BPDDL are PSPACE complete if the maximum temporal reference used is polynomially bounded on the length of the program encoding. See details in proofs.

[7] An assignment is trivial if all the variables are assigned to true or all the variables are assigned to false

As far as the validity problem is concerned, checking if all the plan conditions but the inertia holds can be done in polynomial time (and this is what is needed in APDDL). Verifying if there are no critical situations in BPDDL can be done in polynomial time using a co-NP oracle machine that solves the complement of has_critical_situation.

k-plan problem membership to Σ_2^P derives directly by the NP-completeness of *has_critical_situation*. To prove the Σ_2^P hardness of k-plan we reduce it to the problem of finding an answer set for the extended disjunctive logic program (EDLP programs) [4]. An EDLP program is a set of rules of the form

$$l_1 | \ldots | l_p \leftarrow l_{p+1}, \ldots, l_m, not\, l_{m+1}, \ldots, not\, l_n$$

where $n \geq m \geq p \geq 0$ and each l_i is a literal, i.e. an atom a or the classical negation $\neg a$ of an atom in a first-order language, and *not* is a negation-as-failure operator. The symbol | is used to distinguish disjunction in the head of a rule from disjunction $\vee$ used in classical logic.

The problem of deciding if a propositional (i.e. ground) EDLP has an answer set is Σ_2^P complete [8].

We introduce the reduction with an example. Consider the following propositional EDLP from [17] which states that everyone is pronounced not guilty unless proven otherwise:

$$\begin{aligned} \mathsf{innocent} | \mathsf{guilty} &\leftarrow \mathsf{charged} \\ \neg \mathsf{guilty} &\leftarrow \mathsf{not\ proven} \\ \mathsf{charged} &\leftarrow \end{aligned}$$

From this program we can generate a program based on the following rules with fluents $\mathtt{innocent}, \mathtt{guilty}, \mathtt{charged}, \mathtt{proven}, w$.

$$\begin{aligned} (w^{-1} = 2 \wedge \mathtt{charged} = 1) &\longrightarrow (\mathtt{innocent} = 1 \vee \mathtt{guilty} = 1) \\ (w^{-1} = 2 \wedge \neg(\mathtt{proven} = 1)) &\longrightarrow \mathtt{guilty} = 0 \\ w^{-1} = 2 &\longrightarrow \mathtt{charged} = 1 \end{aligned}$$

If the fluents $\mathtt{innocent}, \mathtt{guilty}, \mathtt{charged}, \mathtt{proven}, \mathtt{w}$ can have values in $\{0, 1, 2\}$ and at the initial state all fluents have value 2 then there is a plan of length 1 iff the EDLP program has a stable model. Intuitively the answer set has an atom a (resp. $\neg a$) if in the final state $a = 1$ (resp. $a = 0$). If $a = 2$ then it is not in the answer set. In the previous case the single answer set is $\{\neg\mathsf{guilty}, \mathsf{innocent}, \mathsf{charged}\}$ and the plan that solves the problem is

$$\begin{gathered} \{\mathsf{guilty}/2, \mathsf{innocent}/2, \mathsf{charged}/2, \mathsf{proven}/2, w/2\}, \\ \emptyset, \\ \{\mathsf{guilty}/0, \mathsf{innocent}/1, \mathsf{charged}/1, \mathsf{proven}/2, w/2\} \end{gathered}$$

PSPACE membership can be proven viewing the planning problem as a reachability problem on a graph where the nodes are states and the arcs are set of actions. Encoding a state and checking if there is an arc between two states is feasible in polynomial space and therefore reachability can be decided in PSPACE.

The plan problem is PSPACE complete because APDDL/BPDDL is more expressive than STRIPS [9]. A STRIPS program can be mapped into a APDDL or BPDDL program straightforwardly and thus since plan in STRIPS is PSPACE complete [5] the plan problem is PSPACE complete also in APDDL or BPDDL.

6 Solver

The positive results of the approach [6] encouraged us to write a constraint-based solver for the languages APDDL and BPDDL. The implementation of BPDLL subsumes that of APDDL. We decided to exploit the constraint solver GECODE, implemented in C++, that offers competitive performance w.r.t. both runtime and memory usage. Starting from the context-free grammar of BPDDL we have defined a lexical analyzer and a parser using the standard tools flex (Fast Lexical Analyser) and Bison. The developed solver solves the k-plan problem.

The overall structure of the solver is similar to that developed in [6] and it deals with the following variables:

1. for every fluent in every state one FD variable (a Boolean variable if the fluent is a predicate) represents the value of the fluent in that state
2. for every action in every transition one boolean variable represents if the action is executed

Constraints for checking action preconditions and imposing action effects are then added. In the case of the BPDDL-solver we also introduced a set of constraints to verify the closure of a state w.r.t. the rules and to solve the frame problem.

Verifying that there are no critical situations can lead to a definition of an exponential number of constraints. As pointed out in [13] and [14] for Answer Set Programming, it is not possible to solve the frame problem adding only a polynomial number of formulas of polynomial length unless $P = NP$.

Let now define a function $\mathsf{shiftRule}(F, r)$ that taking a set of fluents F and a rule r, decreases by one the time reference of all the time fluents `(at 0 f)` in r if $f \in F$.

Let $\mathsf{ruleModified}(f, s_0, A_1, s_1, \ldots, A_n, s_n)$ be the constraint that is true iff $f \notin \mathsf{Act}(s_0, A_1, s_1, \ldots, A_n, s_n)$ and $\mathsf{val}(s_n, f) \neq \mathsf{val}(s_{n-1}, f)$.

There is no critical situation between two states s_{i-1}, s_i in a plan $s_0, A_1, s_1, \ldots, s_n$ if for all non empty subset of fluents F

$$s_0, \ldots, s_n \models \mathsf{shift_i} \left(\bigwedge_{r \text{ rule}} \mathsf{shiftRule(F, r)} \rightarrow \neg \bigwedge_{f \in F} \mathsf{ruleModified(f, s_0, A_1, s_1, \ldots, A_i, s_i)} \right)$$

Intuitively, we check if the rules are fulfilled even if the fluents in F are left unchanged. When this happens we must assure that in this case there is at least a fluent in F that is not modified only by rules.

In the BPDDL solver we tried to minimize the number of constraints added to state the inertia. Suppose P is a partition such that for every `(at 0` f_1`),(at`

0 f_2) defined in a rule one of its elements contains f_1, f_2. To avoid critical situations it is possible to add one of the above-mentioned constraints for all the non empty subsets of the elements in P.

If a metric is used we employ the branch and bound algorithm for finding the optimum solution. Otherwise, we use the default algorithm provided by GECODE for exploring the search space (depth first search).

We also developed two optional heuristics for reducing the search space. The first one, called `no_state_repetition` avoids the possibility of returning to an already visited state (drawback: if a k-plan exists only with multiple visits of a node, we don't find it).

The second heuristics is called `confluent_actions` that imposes a partial order on actions. We say that $a_1 < a_2$ when for every plan of length 2 the execution of a_1 and a_2 has the same effect of the execution a_2 and a_1. We notice that $a_1 < a_2$ is always true if the set of fluents in a_1 effect is disjoint from the set of fluents in a_2 precondition and vice versa. When this happens we impose that in a plan the action a_1 should be executed before the action a_2. This heuristic can reduce the plan symmetries.

Since sometimes we are interested in finding a sequential plan we allow the programmer to choose at most or exactly one action per transition.

7 Tests

For the scope of this paper, we compared the performances of the GECODE based APDDL solver with the performances of the Solver for the language $\mathcal{B}^{MV}$ [6]. For the tests we used an AMD Opteron 2.2 GHz Linux Machine. The APDDL solver uses GECODE 2.1.1 and was compiled with the 4.1.2 version of g++. The $\mathcal{B}^{MV}$ solver instead is written and executed in SICStus Prolog 4.0.4. As benchmarks we chose some of the domains studied and presented in [6]. We are planning some other tests also with other systems (e.g., MIPS-XXL, SG-Plan5, SatPlan) when they are applicable (e.g. for domains without time fluents and rules). All the solver codes and the examples of the program used for the testing are available at [15].

As explained in [6], for implementation choice, treatment of inertia in $\mathcal{B}^{MV}$ can be incorrect for some programs where rules introduce loops. The BPDDL solver, instead, works correctly on those examples. Anyway, in our tests we choose to compare $\mathcal{B}^{MV}$ with APDDL on domains without rules. BPDDL has basically the same running time as APDDL (with a 5% of overhead) on the tested domains.

For every instance of the problems we considered both the time needed by the solver to post the constraints and the time needed to find the first solution (if any). Timings are expressed in ms and are given as a sum of the post time (first term) and the search time (last term in the addition). Even if the two languages used are different (PDDL like vs Prolog like) we encoded the domains basically in the same way (same actions, same preconditions, etc) and we have used both

the solvers with the default parameters (A/BPDDL solver chooses the variable with the smallest domain size and the smallest value during the search process).

Since the $\mathcal{B}^{MV}$ solver is studied for sequential plans we impose the same constraint for the A/BPDDL solver. Table 2 contains the execution times for the n-puzzle problem, the solitaire peg game[8] (a plan with 31 moves), the problem of finding a knight walk on a 4×4 chessboard, and the well-known three barrels problem with barrels of 20-11-9 liters.

Knight and peg has been launched with and without the no_state_repetition heuristic.

Table 2. Experimental results for the 3x3 puzzle, knight walk, peg solitaire and barrels (the * means that the APDDL solver was used without the no_state_repetition heuristics)

Prob.	Instance	Len.	Sol.	APDDL	$\mathcal{B}^{MV}$	$\frac{\mathcal{B}^{MV}}{\text{APDDL}}$
puzzle	I_1	19	No	10 + 3660	150 + 12580	3,5
puzzle	I_1	20	Yes	0 + 70	140 + 4210	62,1
puzzle	I_2	24	No	10 + 93970	150 + 270080	2,9
puzzle	I_2	25	Yes	10 + 38010	180 + 314930	8,3
puzzle	I_3	20	No	10 + 8910	90 + 31140	3,5
puzzle	I_3	25	No	10 + 129760	170 + 463500	3,6
knight		24	Yes	40 + 98610	670 + 2743660	27,8
knight *		24	Yes	30 + 68120	1550 + 2620060	38,5
peg		11	No	10 + 13790	620 + 841340	61,0
peg *		11	No	10 + 9510	640 + 849610	89,3
peg		31	Yes	50 + 41390	1850 + 47910	1,2
peg *		31	Yes	30 + 16690	1780 + 46360	2,88
barrels	20-11-9	18	No	10 + 350	60 + 560	1,7
barrels	20-11-9	19	Yes	10 + 150	60 + 240	1,9

In Table 3 we compare the times for two multivalued problems. The gas diffusion problem where Diabolik wishes to fill a room with sufficient amount of gas in order to generate an explosion below the central bank[9] and a community problem where, according to some rules, rich people wish to give money to poor people in order to reach an equilibrium.

The tests reveal that the APDDL solver is always the fastest one. In particular for the toughest instances the times can be decreased by an order of magnitude or more (see for example the results obtained in the gas diffusion and community problem).

[8] This problem is one of the benchmarks of the 2008 planning competition.
[9] This is a variant of the *pipesworld* domain of the 2006 planning competition.

Table 3. Experimental results for the gas diffusion problem and the community problem

Prob.	Instance	Len.	Sol.	APDDL	$\mathcal{B}^{MV}$	$\frac{\mathcal{B}^{MV}}{\text{APDDL}}$
gas	A_1	6	Yes	0 + 220	70 + 1348	6
gas	A_1	7	Yes	0 + 10	100 + 5350	545
gas	B_1	10	No	10 + 8500	170 + 3846200	452
gas	B_1	11	Yes	0 + 10	140 + 1802760	180290
gas	B_1	12	Yes	10 + 20	150 + 933350	31117
gas	B_1	13	Yes	10 + 70	160 + 302340	3781
gas	B_1	14	Yes	10 + 170	140 + 4600	26
community	A_5	5	No	0 + 9500	50 + 264760	28
community	A_5	6	Yes	0 + 100	30 + 200	2
community	A_5	7	Yes	0 + 12610	40 + 930080	74
community	B_5	5	No	0 + 5080	30 + 131630	26
community	B_5	6	Yes	10 + 0	30 + 110	14
community	B_5	7	Yes	0 + 10	40 + 170	21

8 Conclusion and Future Work

In this work we presented two extensions to PDDL-like languages. The first extension introduces new operators and the notion of time fluents that allow to express plan problems in a more concise way. For example suppose that we have a board of lights and when one of the lights is off or on also the status of the neighbor lights change. In BPDDL this situation can be easily modeled in the following way:

```
(:action press
   :parameters (?light - lights)
   :effect (and
      (xor (at -1 (is_on ?cell)) (is_on ?cell))
      (forall ?neighbor - lights
         (imp
            (== (neighbor ?cell) ?neighbor)
            (xor (at -1 (is_on ?neighbor)) (is_on ?neighbor))
         ))))
```

Then we introduced static causal laws and we provide a solver based on GECODE which has been proved to be effective and it also represents a solid starting point for future extensions. We then characterized the complexities of the major problems relating to planning within the proposed languages. With static causal laws the plan problem became harder than in the case of absence.

Some of the possible next steps are:

- extend the language BPDDL with some construct to query and constrain the occurrences of the action directly in the language
- extend the language to allow more expressive metrics (e.g. it is possible to define metrics giving for every action a certain cost)

– supporting in both languages other features like preferences or hierarchical types
– support multi-agent planning and forms of concurrent actions
– create a compiler from the language PDDL to APDDL
– port the code of the solver using the new GECODE 3.0 environment
– compare the solver with the state-of-the-art PDDL planning solvers

Acknowledgments

We thank Andrea Formisano and Enrico Pontelli for the several useful discussions and technical help. The research is partially supported by PRIN and FIRB RBNE03B8KK projects.

References

1. Gecode: Generic constraint development environment, 2008. Gecode is currently developed by Christian Schulte, Mikael Lagerkvist, Guido Tack and it is available from `http://www.gecode.org`.
2. C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.
3. C. Bell, A. Nerode, R. T. Ng, and V. S. Subrahmanian. Mixed integer programming methods for computing nonmonotonic deductive databases. *J. ACM*, 41(6):1178–1215, 1994.
4. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.
5. T. Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994.
6. A. Dovier, A. Formisano, and E. Pontelli. Multivalued Action Languages with Constraints in CLP(FD). In V. Dahl and I. Niemelä, editors, *ICLP 2007 Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2007. Extended version in http://www.dimi.uniud.it/dovier/PAPERS/rrUD_01-09.pdf.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log.*, 5(2):206–263, 2004.
8. T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *ILPS*, pages 266–278, 1993.
9. R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.
10. M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.
11. M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
12. A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.
13. V. Lifschitz and A. A. Razborov. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2):261–268, 2006.

14. F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
15. J. Mauro and A. Dovier. APDDL and BPDDL codes and examples. Available from `http://www.dimi.uniud.it/dovier/MISIGE/`.
16. D. V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.
17. T. C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Comput.*, 9(3/4):401–424, 1991.
18. S. Thiébaux, J. Hoffmann, and B. Nebel. In defense of pddl axioms. *Artif. Intell.*, 168(1-2):38–69, 2005.

A Complexity results and proofs

Let us start with some basic observations. Given a ground BPDDL program of length n, the rules, the actions, the goal constraints, the initial constraints have all length bounded by n. Similarly, since all the fluents in the initial state must be uniquely determined by the initial constraints, the number of fluents is bounded by n. Therefore, checking if a rule, an action precondition or an action effect is entailed by a state can be done in polynomial time on n. In a similar way checking if two actions can not be done simultaneously is feasible in polynomial time.

Let us define an assignment $\{X_1/v_1, \ldots, X_n/v_n\}$ non trivial if $\exists i, j \in \{1, \ldots, n\}$ s.t. $v_i = true$ and $v_j = false$. Let non-trivial-SAT be the problem of deciding if a formula is satisfied only by a non trivial assignment.

Lemma 1. *non-trivial-SAT is NP-complete*

Proof. NP membership derives from the fact that checking if an assignment satisfies a boolean formula and it is non trivial are two polynomial problems.

NP-hardness can be proved via reduction from SAT. Let φ be a SAT formula and X and Y two fresh boolean variables. Let $\phi = \varphi \wedge (X \vee Y) \wedge (\neg X \vee \neg Y)$. We have that φ is satisfiable iff there is a non trivial assignment that satisfies ϕ. □

Theorem 1. *has_critical_situation is NP-complete*

Proof. NP membership can be proved by noticing that a certificate of the problem is a state s' that entails the rules and $\triangle F(s_i, s_{i+1}) \supset \triangle F(s_i, s')$. Checking such a certificate can be done in polynomial time.

NP-hardness can be proved via reduction from non-trivial-SAT.

Let us consider a formula $\varphi = \psi_1 \wedge \cdots \psi_k$ where ψ_i are clauses. Let $\mathcal{F}$ be the set of variables in φ. We build a BPDDL program as follows (we use an abstract syntax for the sake of clarity):

- for all $i = 1..k$ $(\psi_i \vee \bigwedge_{f \in \mathcal{F}} f = 0 \vee \bigwedge_{f \in \mathcal{F}} f = 1)$
- $\bigvee_{f \in \mathcal{F}}$ *(at -1 f)* $\neq f$

Let us consider the plan $s_0, \emptyset, s_1$ where $\mathsf{val}(s_0, f) = 0$ and $\mathsf{val}(s_1, f) = 1$ for all $f \in \mathcal{F}$.

It is possible to prove that φ is satisfiable only by a non trivial assignment iff there is a critical situation between s_0 and s_1. □

Theorem 2. *validity is co-NP-complete*

Proof. Given a plan, checking if all the plan conditions except the inertia are respected can be done in polynomial time. Verifying if there are no critical situations can be done using a co-NP machine that solves the complement of has_critical_situation. Validity is therefore in co-NP.

Let be no_validity the complement of validity. In Theorem 1 we proved that has_critical_situation is still NP hard even with some restrictions (the plan

has 2 states, there are no actions, ...). With these restrictions no_validity is has_critical_situation and thus it is NP hard.

Since no_validity is NP hard than validity is co-NP hard. □

In APDDL instead the validity problem is only polynomial on the length of the program encoding. The difference between the two languages is made by the definition of inertia that in the case of the BPDDL leads to a potentially exponential search space of states on the number of the fluents.

Theorem 3. *k-plan is in* Σ_2^P

Proof. Let M be a non deterministic Turing machine that guesses a sequence of states and actions $s_0, A_1, s_1, \ldots, A_k, s_k$ and checks if all the plan conditions except inertia are satisfied. Then for checking the inertia M calls k times an oracle machine that solves has_critical_situation.

If all the checks are positive M returns 1, otherwise 0.

M solves k-plan in polytime. Therefore k-plan is in $NP^{NP} = \Sigma_2^P$ □

Given a EDLP program P we define as $T(P)$ the BPDDL program where:

- a multi-fluent a is defined for every atom a in P. These fluents can be 0, 1 or 2
- w is a new fluent
- for every rule $l_1 | \ldots | l_j \leftarrow l_{j+1}, \ldots, l_m, not\, l_{m+1}, \ldots, not\, l_n$ in P there is a rule

$$\left(((\text{at} - 1\ w) = 2) \wedge \bigwedge_{i=j+1}^{m} \sigma(l_i) \wedge \bigwedge_{i=m+1}^{n} \neg\sigma(l_i) \right) \rightarrow \bigvee_{i=1}^{j} \sigma(l_i)$$

where

$$\sigma(l_i) = \begin{cases} (a = 1) & \text{if } l_i = a \\ (a = 0) & \text{if } l_i = \neg a \end{cases}$$

- the initial constraint is $(w = 2) \wedge \bigwedge_{a\ \text{atom}} (a = 2)$
- there are no actions and goal constraints

Given a set S of P literals we use $T(S)$ for the sequence of state and actions s_0, A_1, s_1 where

- $\mathsf{val}(s_0, w) = \mathsf{val}(s_1, w) = 2$
- for every atom in P $\mathsf{val}(s_0, a) = 0$ and

$$\mathsf{val}(s_1, a) = \begin{cases} 1 & \text{if } a \in S \\ 0 & \text{if } \neg a \in S \\ 2 & \text{otherwise} \end{cases}$$

- $A_1 = \emptyset$

Just a simple consideration; given a program P and its transformation $T(P)$ all the initial states satisfy the rules since no rule is applicable to the first state.

We extend the notion of the translation T to rules of EDLP. We use the term $T(r)$ for the BPDDL rule obtained from the EDLP rule r.

With P^S we refer to the Gelfond-Lifschitz transformation [4].

Lemma 2. *if r is a rule of a disjunctive logic program then $S \models r \leftrightarrow T(S) \models T(r)$*

Proof. For definition of $T(S)$ for every literal l $S \models l \leftrightarrow T(S) \models \sigma(l)$ and $T(S) \models ((\text{at} - 1\ w) = 2)$. For definition of $\wedge, \vee, \leftarrow$ we have the thesis. □

Lemma 3. *$S \models P^S$ iff $T(S) \models T(P)$*

Proof. Given S the rules in P can be divided in the following tree disjoint sets.

1. rules that have a $(not\, l)$ term where $l \in S$
2. rules that do not have $(not\, l)$ terms
3. the remaining rules

The lemma can be proven by induction on the number of rules in the EDLP program.

If $P = \emptyset$ then $S \models P = P^S$ and $T(S) \models T(P)$

Suppose that $P = P_1 \cup \{r\}$.

- if r is in the first set $r \notin P^S$. Then $S \models P^S$ iff $S \models P_1^S$. If S is not a model of P_1^S then for inductive hypothesis $T(S) \not\models T(P_1)$ and thus $T(S) \not\models T(P) = T(P_1) \wedge T(r)$. Conversely if $T(S) \models T(P_1)$ then $T(S) \not\models (not\, \sigma(l))$ if $l \in S$. Since in $T(r)$ precondition there is at least one of these literals then $T(S) \models T(r)$ and thus $T(S) \models T(P_1) \wedge T(r) = T(P)$.
- if r is in the second set then for lemma 2 $S \models \{r\} \leftrightarrow T(S) \models T(r)$. For inductive hypothesis $S \models P_1^S \leftrightarrow T(S) \models T(P_1)$ and thus $S \models P^S = P_1^S \cup \{r\} \leftrightarrow S \models P_1^S \wedge S \models \{r\} \leftrightarrow T(S) \models T(P_1) \wedge T(S) \models T(r) \leftrightarrow T(S) \models T(P_1) \wedge T(r) = T(P)$
- if r is in the third set then for all the terms $(not\, l)$ in r $l \notin S$. Therefore $T(S) \models (not\, \sigma(l))$ and thus $T(S) \models r \leftrightarrow T(S) \models r'$ where r' is the rule r without the $(not\, l)$ terms. Now since $P^S = P_1^S \cup \{r'\}$ we can derive that $S \models P^S \leftrightarrow T(S) \models T(P)$

□

Theorem 4. *k-plan is Σ_2^P complete*

Proof. We reduce the existence of the answer set in propositional EDLP program to 1-plan.

Suppose that P has answer set S. Let us assume that $T(S) = s_0, \emptyset, s_1$ is not a plan. For lemma 3 $T(S) \models T(P)$. Since $T(S)$ is not a plan there is a critical situation between s_0, s_1 and therefore there exist s' s.t. $s_0, \emptyset, s' \models T(P)$ and $\triangle F(s_0, s') \subset \triangle F(s_0, s_1)$. If S' is a set of literals s.t. $T(S') = s_0, \emptyset, s'$ we have for

lemma 3 that S' is model of $P^{S'}$ but this is a contradiction since $S' \subset S$ and we supposed S answer set.

Suppose that there exist a plan $s_0, \emptyset, s_1$ for $T(P)$. Then there exist S such that $T(S) = s_0, \emptyset, s_1$. For lemma 3 we have that S is a model of P and therefore P has an answer set □

Theorem 5. *if t_{max}, t_{min} are the maximum and minimum time reference in fluents and $t_{max}, |t_{min}|$ are polynomially bounded on the length of the encoding then plan is PSPACE complete.*

Proof. Every fluent can assume $O(2^n)$ values and thus the number of possible states is $O(2^n)$. Given two states s, s' if $t_{max}, |t_{min}|$ are polynomially bounded it is possible to compute in polynomial space if there exist A s.t. s, A, s' is a subsequence of a plan. This can be done generating non deterministically a polynomial number of states and actions and then solving the validity problem without checking the entailment of goal and initial constraints.

The plan problem can therefore be seen as the reachability problem. Since it is possible to define a state and check if two states are connected in polynomial space using a non-deterministic Turing machine then the entire plan problem can be solved in polynomial space using a non-deterministic Turing machine. Since NPSPACE = PSPACE, plan is in PSPACE.

The plan problem is PSPACE complete because A/BPDDL is more expressive than STRIPS [9]. A STRIPS program can be mapped into a A/BPDDL program straightforwardly and thus since plan in STRIPS is PSPACE complete the plan problem is PSPACE complete also in A/BPDDL. □

Even if metric functions are used the optimization problem derived is in PSPACE. This is due to the fact that the metric function depends on the value of fluents in the final state and thus the metric value can be encoded in $O(n^2)$ space.