Compiling and Executing PDDL in Picat

Marco De Bortoli[1], Roman Barták[2], Agostino Dovier[1], and Neng-Fa Zhou[3]

[1] Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine, Italy
[2] Charles University in Prague, Faculty of Mathematics and Physics,
Prague, Czech Republic
[3] City University of New York, Brooklyn College,
New York, U.S.A.

Abstract. The declarative language Picat has recently entered the scene of constraint logic programming, in particular thanks to the efficiency of its planning library that exploits a clever implementation of tabling, inherithed in part from B-Prolog. Planning benchmarks, used in competitions, are defined in the language PDDL and this implied that Picat users were forced to reimplement those models within the language. In this paper we present an automatic compiler from PDDL to Picat planning models. The compiler is written in Picat and tested on standard PDDL benchmarks.

Keywords: Action Languages, Planning, Tabling

1 Introduction

Solving planning problems is a central task of AI since the birth of this discipline. Ideally, an autonomous agent should be able to take his own decisions, in order to achieve a given goal, affecting a portion of the world. In automated planning, this is modeled using a set of variables to describe the state of the world, and a set of possible actions, each of them applicable if some preconditions hold, and affecting the values of the variables if applied. A suitable sequence of actions leading to the goal is a (successful) plan. Languages for reasoning on this problem have been proposed (action languages, see, e.g., [7]) and solvers for these languages based on encoding to SAT, Answer Set Programming, or Constraint Programming have been proposed (see, e.g. [4, 3, 2]). The increasing popularity of planning gave birth to the IPC (International Planning Competition) [15], an event where the participants test their planners on a set of benchmarks. To this aim the Planning Domain Definition Language (PDDL [9]) has been introduced as a standard encoding (action) language that all solvers willing to participate to the competition should understand.

Picat [13] is a recently designed declarative programming language, offering a large number of different features inherited from several programming paradigms, from the classic imperative, to scripting, and logic and constraint programming. One of the most successful features of Picat up to now is the

planner module that performs well on IPC benchmarks (see, e.g., [1, 11]). One of the key ideas of the planner module is to exploit a clever implementation of tabling, partially inherithed from the B-Prolog implementation and already proved to behave well in hard planning benchmarks (such as the sokoban problem [12]).

However, in the above papers, PDDL encodings have been ported in Picat by hand, one domain at a time, while one might be interested in having an automatic Picat model from PDDL benchmarks. The subject of this paper is the development of a compiler from PDDL to Picat, in order to enable the latter to solve automatically competition instances. The compiler is written directly in Picat. After a brief survey on PDDL evolution, we explain the main ideas of the translator and we compare the running time of the translated code in Picat with a state-of-the-art PDDL solver and with Picat with ad-hoc encodings of the same problems. Since Picat is a Turing complete programming language (not simply a modeling language) extra features exploiting the problem structure can be programmed to speed-up the solution. This paper can be seen as a starting point for future work where some of these features are automatically used to improve the running time of the compiled domains.

2 Planning and PDDL

In classical planning, a domain model is a description of the world using variables for representing the attributes of objects of interest in states, and of actions and of how they might affect the state if their preconditions are satisfied. A planner is an algorithm capable of finding a suitable sequence of actions that leads from a given initial state to a goal state. This activity can be seen as the search of a path in a directed graph, where every node represents a state of the world, and every edge corresponds to an action. If $s_{i+1} = \gamma(s_i, a_i)$ is the state obtained by applying action a_i to the state s_i, and if for each $i \in \{0, \ldots, n-1\}$ the action a_i is applicable to state s_i, a *plan* is the sequence of actions $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ that applied to the initial state s_0 yelds the final state s_n.

Modern planners have their roots in Shakey the robot [10], the first robot that was able to reason about its own actions. It used the STRIPS (STanford Research Institute Planner) automated planner [5], that became the formal specification language for planning and it is the base of modern action description languages. Among them PDDL, proposed by Drew McDermott in 1998 [9], is *the* language of the International Planning Competition (IPC [15]). PDDL was improved many times, by adding new constructs, from the first version (1.1). PDDL uses "physics only" principles to describe a domain, so it has no clue how to reach the goal or resolve the problem, and this contributes to make PDDL planners totally domain-independent, with separate files for describing the domain and the specific problem, the latter consisting of the description of the initial and final state.

In PDDL modeling, domain and problem files are separated and both of them should be invoked during execution. The problem file is structured as follows.

```
(define (problem <problem name>)
    (:domain <domain name>)
    (:objects [object declaration])
    (:init [logical fact])
    (:goal [precondition]))
```

The init and goal sections should contain the definitions of the properies that should hold (i.e., predicates that must be true) in the initial (resp., final) state. If the domain uses typing (see below), in the objects declaration the types of the objects are defined. A generic PDDL domain file is composed by the definition of predicates (logical facts that can be typed), constants, and of the description of a certain number of actions, every one characterized by: *parameters*, i.e variables that may be instantiated with objects, *preconditions*, namely the conditions that must be true to activate the action, and *effects* (post-conditions), i.e., the predicates that will become true (or false) after the action execution.

2.1 Brief History of PDDL Evolution

PDDL 1.2 was used for the 1998 and 2000 IPC. It had basic numeric-value support, that allowed numeric quantities to be assigned and updated, but in those competitions numeric operations were avoided and therefore these features were not exploited. It had also the possibility to define a object-type hierarchy and constant objects.

Below we report the action drive of a simple depot domain that does not use type hierarchy. Its aim is to move the truck from a place to another for loading a cargo

```
(:action drive
     :parameters ( ?x ?y ?z)
     :precondition
          (and (truck ?x)
               (place ?y)
               (place ?z)
               (at ?x ?y))
     :effect
          (and (at ?x ?z)
               (not (at ?x ?y))))
```

The planner, for applying the action, tries to instantiate the three variables with three objects from the input (problem) file: if it succeeds in finding three objects satisfying the *four* preconditions, it can execute the action, reaching a new state where the truck x drives to z (and it is therefore no longer in y).

The use of a *type hierarchy* allows the programmer to decrease the number of predicates used in preconditions (e.g., predicates like *place* and *truck* in the above example). This is possible with the following versions of PDDL such as version 2.1, where numeric fluents, basic math functions like sum and quantifiers both for pre-conditions and post-conditions are also allowed. Numeric fluents are

functions $Object^n \longrightarrow \mathbb{R}$, that can be used for action costs management. This release was the specification for IPC 2002 [6].

The action drive taken from the transport domain shows how numeric fluents can be inserted in an action concerning the movement of a truck.

```
(:action drive
    :parameters (?v - vehicle ?loc1 ?loc2 - location)
    :precondition
        (and (at ?v ?loc1)
             (road ?loc1 ?loc2))
    :effect
        (and (not (at ?v ?loc1))
             (at ?v ?loc2)
             (increase (total-cost) (road-length ?loc1 ?loc2))))
```

There is a numeric function total-cost, without arguments, that is used to keep memory of the costs of the actions, that depends on the distance of the two locations: this information is returned by a function, road-length. Using these functions the programmer can exploit optimization features (e.g., minimizing the total distance of a path). This is achieved by the statement:

```
(:metric minimize (total-cost))
```

As anticipated, in this example there isn't a predicate like truck to check if v is a vehicle, since a type hierarchy is used to define the variables.

The PDDL language supports many different levels of expressiveness. The smallest possible subset is STRIPS, from the language that gave birth to Automated Planning, but, since PDDL 2.1, a larger subset, namely the ADL set (Action Description Language), is introduced to exploit the new features, such as disjunction and negation in preconditions and quantifiers both in preconditions and effects.

Version 3.1 is currently the latest release of PDDL.[4] It doesn't include all features offered by 2.1 version, but it offers some interesting new features such as the possibility of having functions $Object^n \longrightarrow Object$ that therefore can have return types different from $\mathbb{R}$.

Until now, we have seen only examples with a basic type hierarchy, where all types are at the same level. PDDL permits to create relationships between types, creating a Type Hierarchy that reminds a class-hierarchy in OOP. Let us consider this example from the nomystery domain:

```
(:types movable location  - object)
(:predicates
    (at ?n - movable ?l - location)
    (inPred ?c - movable ?t - movable)
    (goal ?c - movable ?l - location)
```

[4] Checked June 1st 2016.

```
    (truck ?t - movable))
(:action move
    :parameters (?t  - movable  ?loc1 ?loc2 - location )
    :precondition  (and  (truck ?t) (at ?t ?loc1))
    :effect (and (not (at ?t ?loc1)) (at ?t ?loc2)))
```

In the next example we will define two types of movable objects: truck and cargo:

```
(:types movable loc  - object
    truck cargo - movable)
(:predicates
    (at ?n - movable ?l - location)
    (inPred ?c - cargo ?t - truck)
    (goal ?c - cargo ?l - location))
(:action move
   :parameters (?t  - truck  ?loc1 ?loc2 - location )
   :precondition  (at ?t ?loc1)
   :effect
         (and  (not (at ?t ?loc1))
               (at ?t ?loc2)))
```

With this type hierarchy we can discriminate immediately between two movable objects like a truck and a cargo. In this way, specifying in the parameters that ?t must be a truck, we can omit the truck predicate and still use the at predicate for all movable objects, without the need to create another predicate. The structure of the type system must be kept when translating to Picat.

3 The Picat Language

Picat is a general-purpose programming language, developed in 2012 by Neng-Fa Zhou, which aims to collect the features of various types of programming languages [13]. It provides control flow features from imperative programming together with more advanced features from logic programming, functional programming and scripting, with the purpose of obtaining compact, intuitive and easy to read programs. The Picat implementation is based on the B-Prolog engine but, despite this, it is far more expressive and scalable, thanks to arrays, loops, lists and array comprehension, allowing the solution of a problem with fewer lines of code.

Picat is a dynamically-typed language, so type checking occurs at runtime. Variables in Picat are identifiers that begin with a capital letter (or the underscore), and they are uninstantiated until they are bound to a value. A value in Picat can be *primitive*, such as Integer and Real, or *compound* such as lists $[\mathsf{t}_1, \ldots, \mathsf{t}_n]$ or structures $\$\mathsf{identifier}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ (where identifier is the name of the structure, and the t_i are terms). The use of "$" is necessary to discriminate structures from function calls. Other compound types are arrays, of the form $\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$ (they are in fact a special kind of structure without name) and maps, a hash-table of pairs of values and keys represented with a structure.

Picat provides list/array comprehension for defining lists/arrays in a compact way.

Predicates and functions in Picat are defined with pattern-matching rules. They take the form $Head, Cond \Rightarrow Body$ in the deterministic case, and $Head, Cond\ ? \Rightarrow Body$ if we want to make the rule backtrackable, i.e. able to return multiple answers (the default in Prolog).

In the following we report the definitions of a deterministic function and of a non-deterministic bacOktrackable predicate:

```
power(X,1) = Result => Result = X.
power(X,N) = Result => Result = X * power (X, N-1).

member(X,[Y|_])   ?=> X = Y.
member(X,[_|List]) => member (X,List).
```

One of the most interesting features of Picat is *tabling*. Tabling guarantees recursive programs with bounded-size terms to terminate, preventing infinite loops and redundancy; morever, tabling can be exploited for implementing dynamic programming when a parameter is declared to be minimized or maximized.

3.1 Planning in Picat

Picat includes a `planner` module that, combined with the expressiveness of the language, allows the user to write domain models that are more sophisticated and shorter than those in PDDL. The programmer is more aware and has more control over the execution of the search process, thanks to features such as the choice between determinism and non-determinism for an action or the fact that a *state* can be represented by any kind of term provided by Picat. But the main peculiarity of the Picat planner is the use of tabling to improve plan creation, without user intervention. To achieve this goal, tabling brings some interesting properties to the planner: for example, Picat is able to recognize if it is on an already visited state, because it has previously memorized it, preventing loops; this operation increases memory requirements, but significantly speeds up the search process as much.

A predicate for checking whether a state is final must be defined:

```
final(State)   =>  <goal_condition>.
```

The goal_condition can be defined by other predicates in Picat (analyzing the content of the variable State). Let us observe that it must be deterministic. Actions are instead defined as:

```
action(+State,-NextState,-Action,-Cost),
   precondition,
   [control_knowledge]
?=>
```

```
  description_of_the_next_state,
  action_cost_calculation,
  [heuristic_and_deadend_verification].
```

Taking the input State as the first parameter, if the preconditions are true, Picat might apply the action, changing the current state according to description_of_the_next_state. We must specify a cost for the action (computed with some algorithm). In the optional parts control_knowledge and heuristic_and_deadend_verification we can include extra domain-dependent knowledge or heuristics that allow Picat to cut the search space when the resources used plus the action cost exceeds a heuristic bound associated to the new state.

The basic search of the best plan is implemented by the predicate:

```
table (+,-,min)
path(S,Plan,Cost),final(S) =>
      Plan=[],Cost=0.
path(S,Plan,Cost) ?=>
      action(S,NextS,Action,ACost),
      path(NextS,Plan1,Cost1),
      Plan = [Action|Plan1],
      Cost = Cost1+ACost.
```

Tabling attributes have the following meaning: $+/-$ denotes an "input/output" argument, while min means "output argument that must be minimized".

This basic schema is extended to allow resource-bounded search, possibly looking for one generic plan or for the best one. Several variants are implemented. Tabling and branch-and-bound are successfully merged. Besides the basic resource-bounded search, Picat provides variants such as *iterative-deepening*, and *branch-and-bound*.

3.2 A remark on State Representation

A state can be represented in several different ways that might affect efficiency. The so-called *Factored Representation* is the typical representation allowed by PDDL: a state is defined by a set of (propositional) atoms. An atom can be *rigid* if it represents a property that never changes or *fluent*, otherwise. In the nomystery domain, for instance, a state can be represented (in Picat) by a list:

```
$[ at(c1,loc1),at(c2,loc2),at(t,loc3),
   connected(loc1,loc2),connected(loc2,loc3),
   truck(t) ]
```

This representation allows some variants. Various levels of speed-ups can be obtained by keeping the list ordered, or using one list for each predicate (at, connected, and truck in example).

An alternative representation, typical of Picat is the so-called *Structured representation*. The structured representation is more compact and reduces symmetries: these features fit well with tabling. For example, consider two identical

trucks moving between locations: Picat during the search will consider the two states: (truck_1 in loc_1 and truck_2 in loc_2) and (truck_1 in loc_2 and truck_2 in loc_1) as different states. Let us consider a representation abstracting from the specific name of equivalent objects. We can represent the state as s(TruckLoc, TruckLoad, Cargo), where TruckLoc is the position of the truck, TruckLoad is a list of the destinations of the cargo loaded on the truck, and Cargo is a list of pairs (From, To) representing cargo items to load.

4 The compiler

In this section we present the main contribution of the paper, namely the definition and implementation of a tool for the automatic conversion of PDDL domains and instances into Picat, written in Picat. It provides support for many PDDL features, like object typing (not naturally supported by Picat planner), functions (either numerical or not), action costs, quantifiers, and other features, some of which unsupported by some PDDL planners (it includes also the pi2pddl parser written by Neng-Fa Zhou for problem instance conversion). The output of the compiler is a Picat program, exploiting the planner library, which can be used as a starting point for subsequent optimizations using Picat features. In the description below, we will use $\Rightarrow$ for representing the transformation of portions of codes from PDDL to Picat.

The *Factored State Representation* is the one used in PDDL and thus the one considered by our translator. A state is represented by a term $s(v_1, \ldots, v_n)$ storing the tuple of values of the fluent variables $p_1, \ldots, p_n$ defining it. To avoid redundancy and to save memory for tabling *Picat rigid facts* are used for static info. Since in PDDL there is no explicit distinction, rigid predicates are inferred by a static analysis of the effect of all the actions.

Domains and problems are separated, so the parser must read the domain file to collect all rigid facts. To complete this task we use list iterator:

```
Facts = [Fact : Fact in IFacts,
                ( Fact = $rigid_1(_); ... ;Fact = $rigid_n(_) )],
cl_facts(Facts,[]).
```

where IFacts is the list containing the input predicates and cl_facts/2 is the Picat built-in predicate for storing rigid facts. PDDL functions are instead represented with maps, using the input attributes as the key and the return value as the map value.

4.1 A simple translated action

Let us start by showing the translation of a simple action, without typing, quantifiers and action-costs. Translation of conjunctions and of function calls is the same for both preconditions and effects.

– **Conjunctions** in Picat, as in Prolog, are written with *commas*, thus $\mathsf{and}(\cdots)(\cdots)\cdots(\cdots)) \Rightarrow (\cdots),(\cdots),\cdots,(\cdots)$
– **Function calls**:
$(\mathsf{function}(\mathsf{x}_1,\ldots,\mathsf{x}_n)) \Rightarrow \mathsf{get}(\mathsf{FUNCTION},(\mathsf{X}_1,\ldots,\mathsf{X}_n),\mathsf{null})$ The third argument of the call is the return value in case of search failure. Observe that FUNCTION, $\mathsf{X}_1,\ldots,\mathsf{X}_n$ are written in capital letters according to the usual Logic Programming syntax rules, inherited by Picat.

Precondition parsing

– **Disjunctions** are represented using *semicolons*:
$\mathsf{or}(\cdots)(\cdots)\cdots(\cdots)) \Rightarrow ((\cdots);(\cdots);\cdots;(\cdots))$
– **Negation** is dealt with similarly: $((\mathsf{not}(\cdots)) \Rightarrow \mathsf{not}(\cdots)$. However, variables need to be instantiated before a negated condition is analyzed in Picat, and therefore some extra care is needed, as explained below.
– **Predicate** checking:
$\mathsf{predicate}(\mathsf{x}_1,\ldots,\mathsf{x}_n) \Rightarrow \mathsf{member}((\mathsf{X}_1,\ldots,\mathsf{X}_n),\mathsf{PREDICATE})$
If the effect of an action changes the values of some of its preconditions, using select/2 instead of member/2 we can reduce (on average) the number of accesses to the same list during computation. Therefore, if this case is detected (by static analysis) select instead of member is used.
– **Rigid Predicate** checking:
$(\mathsf{predicate}(\mathsf{x}_1,\ldots,\mathsf{x}_n)) \Rightarrow \mathsf{predicate}(\mathsf{X}_1,\ldots,\mathsf{X}_n)$ In this case we do not need to check a value in a state, but simply to call a predicate.

Effect parsing

– **Adding a logical fact**:
$(\mathsf{predicate}(\mathsf{x}_1,\ldots,\mathsf{x}_n)) \Rightarrow$
$\quad \mathsf{PREDICATE\_1} = \mathsf{insert\_ordered\_wod}(\mathsf{PREDICATE\_0},(\mathsf{X}_1,\ldots,\mathsf{X}_N))$
insert_ordered_wod/2 is built on the built-in insert_ordered/2 but it avoids to inserting duplicates.
– **Removing a logical fact** (if it wasn't already done by *select*/2 in the preconditions):
$(\mathsf{not}(\mathsf{predicate}(\mathsf{x\_1},\ldots,\mathsf{x\_n})) \Rightarrow$
$\quad \mathsf{PREDICATE\_1} = \mathsf{delete}(\mathsf{PREDICATE\_0},(\mathsf{X}_1,\ldots,\mathsf{X}_N))$

Since select/2, insert_ordered_wod/2 and delete/2 require an empty list for the return value, subscripts are added to create auxiliary variables. An example, the translation of a variant of action drive (see section 2.1) is:

```
action(s(AT_0,CARRIED_0,DEST_0),NextState, Action, Cost),
   truck(T),
   AT_1 = select((T,LOC1),AT_0),
   connected(LOC1,LOC2)
?=>
   AT_2 = insert_ordered_wod(AT_1,(T,LOC2)),
   NextState = s(AT_2,CARRIED_0,DEST_0)
   Action = $move(T,LOC1,LOC2),
   Cost=1.
```

4.2 Typing

If all the objects are at the same level in PDDL, we can safely ignore the typing in Picat. Instead, if the PDDL domain has a hierarchy tree with more levels, some actions may not work properly. For instance a move action designed for trucks can be converted in a Picat program that could move cargos (if they are in the correct location for a move) without first connecting them to a truck. This issue can be solved by adding extra information about input objects using rigid facts, e.g.:

```
$location(loc1), $truck(t), $cargo(c1), ...
```

Thus, with static analysis, the compiler retrieves all the types from the PDDL problem file and adds them as rigid predicates. The check on type is added only when there is the risk of ambiguity. Since in PDDL you need to specify the types of the arguments used in the action, if we use the fuel predicate in the precondition ((fuel?t − truck?g − gas) is the predicate definition) together with the (?t − movable) declaration on the *parameters* section , the t variable would be bound to be a truck and not a generic movable, so a cargo would not match with t.

4.3 Action - costs

One of the most important features introduced in PDDL 2.1 is the concept of *numeric-fluent*. In the IPC, the rules require using *:action-costs*. This is a subset of *numeric-fluents*, and it permits the usage in *effect* of a 0-ary function, *:total-costs*, to store and modify the actual cost, depending on the action. Its value can be incremented with a non negative number, by either specifying it or calling a numeric fluent, as in the following examples:

```
(increase (total-cost) 10)
(increase (total-cost) (road-length ?l1 ?l2) )
```

Exploiting the *Cost* parameter of Picat, the translation is the following:

```
Cost = 10,
Cost_1 = Cost_0 + get(ROAD_LENGTH,(L1,L2),null).
```

4.4 Quantifiers

The translation of the *forall* quantifier makes use of the Picat built-in *foreach*. Let us start analyzing the preconditions part. We first compute a list of elements of the correct type (using list comprehension) and then loop on list elements:

```
( forall (?t - type) [preconditions] )
```

becomes (actually, the list is created once and stored as a rigid fact statically).

```
type_list(TYPE_list),
foreach(T in TYPE_list)
       [preconditions]
end
```

This approach cannot be used in the effect part, since lists need to be modified within the loop. The problem is solved by defininining and invoking an auxiliary recursive predicate updating the state.

4.5 Avoid floundering

In Logic Programming with Negation as Failure, when a negative atom is processed with some of its variables not bounded to ground terms the computation is said to *flounder* [8, pp. 88], possibly leading to unsoundness. This problem is inherited by Picat. Therefore, by applying static analysis, the compiler moves negative literals in preconditions after the others. The problem arises more generally when a variable occuring in the parameters section is not yet instantiated when the program enters the effect section. Consider this example drawn from the maintenance domain:

```
(:action workat
  :parameters (?day - day ?airport - airport)
  :precondition (today ?day)
  :effect (and  (not (today ?day))
     (forall (?plane - plane)
           (when (at ?plane ?day ?airport)
                 (done ?plane)))))
```

In this case, PDDL binds airport even if it is not affected by any precondition, so the done facts become true only for the planes placed in *that* airport today. In Picat the variable corresponding to airport will be instead unbound so the member((PLANE, DAY, AIRPORT), AT_0) predicate will be true for each plane placed in *any* airport today.

To fix this issue, a grounding of the variables used in effects and not in the preconditions is imposed by the compiler by adding "type" predicates for them. For this example the translation is:

```
action(s(AT_0,DONE_0,TODAY_0),NextState, Action, Cost),
   select((DAY),TODAY_0,TODAY_1),
   airport(AIRPORT)
?=>
   plane_list(PLANE_list),
   PLANE_l = [PLANE : PLANE in PLANE_list,member((PLANE,DAY,AIRPORT),AT_0)],
   DONE_1 = sort (DONE_0 ++ PLANE_l),
   NextState = $s(AT_0,DONE_1,TODAY_1),
   Action = $act_workat(AIRPORT,DAY),
   Cost=1.
```

4.6 Input problem handling and final state checking

The parser not only encodes state transitions, but also adds a section for handling the input and launch the search of a plan, besides final state checking. For example, in the nomystery domain:

```
pddl(IFacts,GFacts) =>
  initialize_table,
  AT_INIT = sort([(L,O) : $at(L,O) in IFacts]),
  FUEL_INIT = sort([(LEVEL,T) : $fuel(LEVEL,T) in IFacts]),
  IN_INIT = sort([(P,T) : $in(P,T) in IFacts]),
  AT_GOAL = sort([(L,O) : $at(L,O) in GFacts]),
  FUEL_GOAL = sort([(LEVEL,T) : $fuel(LEVEL,T) in GFacts]),
  IN_GOAL = sort([(P,T) : $in(P,T) in GFacts]),
  Facts = [Fact : Fact in IFacts, (
     Fact = $location(_);       Fact = $cargo(_);
     Fact = $truck(_);          Fact = $connected(_,_) )],
  cl_facts([$goal(AT_GOAL,IN_GOAL)|Facts],[]),
  best_plan_bb($s(AT_INIT,IN_INIT),99999999,Plan,PlanCost),
  writeln(plan=Plan), writeln(cost=PlanCost).
```

IFacts and GFacts contain the Init and the Goal facts as they are written in the problem file, besides the types of the objects inserted by a modified version of the instance parser (the original one doesn't consider them). For each non-rigid predicate, we create two lists collecting the objects which satisfy the property respectively for the initial and the final state. As said above, Facts collects rigid facts and object types. Then the goal condition lists are stored in a goal rigid predicate, and the *branch and bound* search is executed. To know if the final state is reached, all we need is to check if the goal preconditions lists are subsets of the current ones:

```
final(s(AT,IN)), goal(AT_GOAL,IN_GOAL),
subset(AT_GOAL,AT), subset(IN_GOAL,IN)  =>   true.
```

5 Experimental results

We report on experimental results made on some benchmarks from the IPC web site [14]. We used the PDDL encodings available there and run them with *Metric - FF 2.1*, a state-of-the-art PDDL planner declared Top Performer in the STRIPS Track of the 3rd International Planning Competition. We automatically converted them to Picat with our compiler and run Picat on them. Moreover, we also compared some of them with a direct, structured encoding in Picat retrieved from Picat web site. Tests are executed on a notebook with a CPU Intel Core I5 4210h at 3.42 Ghz and 8 gigabytes of RAM. We used the domains (descriptions are retrieved from official competition website):

Nomystery (nomys) (a complete version, using typing and action costs) A

truck moves in a weighted graph where a set of packages must be transported between nodes. Actions move the truck along edges and load/unload packages. Each move consumes the edge weight in fuel. The edge weights are uniformly drawn between 1 and an integer w. The initial fuel supply is set to $C \cdot M$ where M is the minimum required amount of fuel, calculated using a domain-specific optimal solver, and $C \geq 1$ is a (float) input parameter that denotes the ratio between the available fuel vs. the minimum amount required. The closer C to 1, the more constrained the problem. If $C = 1$ only the optimal plan solves the problem.

Hiking (hik) Imagine you want to walk with your partner a long clockwise circular route over several days (e.g. in the "Lake District" in NW England), and you do one leg each day. You want to start at a certain point and do the walk in one direction, without ever walking backwards. You have two cars which you must use to carry your tent/luggage and to carry you and your partner to the start/end of a leg, if necessary. Driving a car between any two points is allowed, but walking must be done with your partner and must start from the place where you left off. As you will be tired when you have walked to the end of a leg, you must have your tent up ready there so you can sleep the night before you set off to do the next leg the morning.

Maintenance (maint) There are mechanics who on any day may work at one of several cities where the airplane maintenance company has facilities. There are airplanes each of which has to be maintained during the given time period. The airplanes are guaranteed to visit some of the cities on given days. The problem is to schedule the presence of the mechanics so that each plane will get maintenance.

Tetris (tet) This is a simplified version of the well-known Tetris. All the pieces (1×1, 2×1, L) are randomly distributed on a $N \times N$ grid. The goal is to move them in order to free the upper half of the grid. The pieces can be rotated or translated. Each movement action has a different cost, accordingly to the size of the piece.

Results are reported in Tables 1 and 2. Different rows are used for different instances. Columns in Table 1 identify different Picat search techniques on the translated code. Table 2 shows the results of the the tests with FF, Picat on the automatically obtained code and Picat with a structured encoding (iterative deepening is used). For each test we report the value of the best result found and the running time in seconds. OOM stands for "Out of Memory".

In *nomystery*, after testing of the automatically translated code of the first PDDL domain, we tried a slightly different version, where load and unload actions are a little (more) deterministic in order to avoid situations in which the truck passes through a location where there is something to load (or to unload) without loading (or unloading) it (this change can be made at the PPDL level as well). Thanks to this change we obtained far better results, even if not at the same level of the structured version. We characterize these instances by adding letter "d". In Table 2 the "d" is within brackets to denote that this is the version used

problem	branch and bound		iterative deepening		first plan unbounded		best plan unbounded	
	bound	time	bound	time	bound	time	bound	time
nomys_p01	18	0.132	18	0.194	25	0.0	18	0.063
nomys_p02	21	0.781	21	0.807	31	0.0	21	0.24
nomys_p03	34	100	34	133	59	0.0	34	23.01
nomys_p04	-	OOM	-	OOM	110	0.0	-	OOM
nomys_d_p01	18	0.015	18	0.016	25	0.0	18	0.0
nomys_d_p02	21	0.022	21	0.046	31	0.0	21	0.016
nomys_d_p03	34	0.36	34	0.67	59	0.0	34	0.14
nomys_d_p04	48	13.61	48	22.75	110	0.0	48	4.75
hik_p_1_2_3	11	0.247	11	0.051	155	0.002	11	0.04
hik_p_1_2_5	31	29.13	25	1.495	1876	0.078	25	0.859
hik_p_1_2_8	-	OOM	45	43.47	15110	1.625	45	14.06
maint_p01	7	3.975	7	7.145	10	0.235	7	4.187
maint_p02	6	4.74	6	6.316	10	0.063	6	7.125
tet_p01	10	2.401	10	0.13	526	0.0	10	2.5
tet_p02	11	46.716	11	4.448	87973	0.593	11	131

Table 1. Picat search techniques comparison on the automatically obtained code

for the "Automatic" column only. FF times are better, but Picat returned the optimal plans in all the instances.

In *Hiking* the FF planner wins once again in search time, but fails in finding the best plan, except for the first instance. Since we need groups of actions to move from a place to another, it could happen that with a bound b the problem is resolvable, but with $b - 1$ it is not, making the planner think that b is the optimal bound when exploiting *branch and bound* search. In fact, this is the only case in which Picat did not return an optimal plan, except of course for the *first plan* search. Anyway, the *best plan unbounded* seems to be the best choice for this domain also from the performance perspective.

The *maintenance* domain, with its only action, is particularly favorable for FF, with all the instances executed almost immediately, but again it didn't always find the optimal plans.

Regarding the last domain, *Tetris*, the translated domains beat the FF planner in the first instance using *iterative deepening*. Focusing on the structured version, powered with a heuristic computing the shortest path for a block allowing the planner to cut off a lot of states, we also see that the *iterative deepening* search acts very well, beating FF in all the instances.

6 Conclusions

We have presented a compiler that translates PDDL domains into Picat. Results can be used as a base by Picat programmers to create more compact and more performing encodings, using all the features that this language offers. It can be seen, in a sense, as a good starting point for the automation of optimization stages, in order to obtain immediately fast Picat programs. At the moment, the

problem	FF		Picat Factored (Automatic)		Picat Structured	
	bound	time	bound	time	bound	time
nomys_p01 (d)	18	0.02	18	0.016	18	0.0
nomys_p02 (d)	23	0.04	21	0.016	21	0.0
nomys_p03 (d)	44	0.04	34	0.15	34	0.016
nomys_p04 (d)	74	0.02	48	5.2	48	0.444
hik_p_1_2_3	11	0	11	0.040	11	0.016
hik_p_1_2_5	32	0.18	25	0.859	25	0.48
hik_p_1_2_8	61	0.52	45	14.060	45	10.47
maint_p01	7	0	7	3.975	7	0.021
maint_p02	7	0	6	4.740	6	0.040
tet_p01	10	0.58	10	0.130	10	0.008
tet_p02	11	0.52	11	4.448	11	0.014

Table 2. Picat vs FF

execution on the translated programs is still much slower than state-of-the-art planner on the PDDL encodings, but we made a first step in the right direction of automatically getting Picat programs that perform similarly or better than state of the art PDDL planners.

Acknowledgments. The compiler was developed during a two months visiting period of Marco De Bortoli in Prague thanks to a scholarship program of the University of Udine. The authors would like to thank Jindrich Vodrazka for his help in the earlier encoding stages. Roman Bartak is supported by the Czech Science Foundation under the project P103-15-19877S. Agostino Dovier is partially supported by INdAM-GNCS 2015 and 2016 projects.

References

1. Roman Barták, Agostino Dovier, and Neng-Fa Zhou. On modeling planning problems in tabled logic programming. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 31–42. ACM, 2015.
2. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An investigation of multi-agent planning in CLP. *Fundam. Inform.*, 105(1-2):79–103, 2010.
3. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Multivalued action languages with constraints in CLP(FD). *TPLP*, 10(2):167–235, 2010.
4. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Perspectives on logic-based approaches for reasoning about actions and change. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, pages 259–279. Springer, 2011.
5. Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

6. Maria Fox and Derek Long. PDDL2.1: an extension to pddl for expressing temporal planning domains. In *Journal of Artificial Intelligence Research*, 2003.
7. Michael Gelfond and Vladimir Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
8. John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
9. D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998.
10. Nils J. Nilsson. Shakey The Robot, Technical Note. Technical Report 323, SRI International's Artificial Intelligence Center, Menlo Park, CA, 1984.
11. Neng-Fa Zhou, Roman Barták, and Agostino Dovier. Planning as tabled logic programming. *TPLP*, 15(4-5):543–558, 2015.
12. Neng-Fa Zhou and Agostino Dovier. A tabled prolog program for solving sokoban. *Fundam. Inform.*, 124(4):561–575, 2013.
13. Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer, 2015.
14. International planning competitions 2014 web site. https://helios.hud.ac.uk/scommv/IPC-14/. Accessed: 2016-02-23.
15. International planning competitions web site. http://ipc.icaps-conference.org/. Accessed: 2016-02-20.