

Compiling and Executing PDDL in Picat

Marco De Bortoli, Roman Bartak, Agostino Dovier, Neng-Fa Zhou

Università degli Studi di Udine

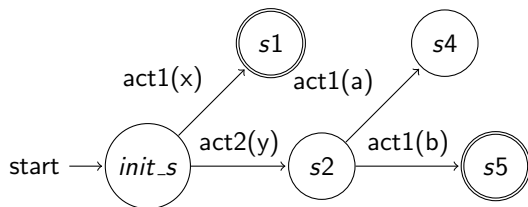
CILC 2016

Outline

- Introduction to classical planning
- The PDDL Language
- The Picat Language
- The compiler
- Experimental results
- Conclusions and further work

Classical Planning

- Classical Planning is the simplest version of automated Planning and Scheduling, a core branch of Artificial Intelligence involved in the realization of a strategy, intended as an accurate sequence of actions, to resolve a specific problem in a world described by state variables.
- Automated Planning is composed of two main components:
 - ▶ the domain
 - ▶ the planner
- The operation of finding this sequence of actions by the planner can be seen as the search for a path in a direct graph, where every node represents a state of the world, and every edge corresponds to an action.



The PDDL Language

- The Planning Domain Definition Language was created in 1998 by Drew McDermott to standardize all the languages used for classical automated planning and to be the official language of the 1998/2000 IPC, the first International Planning Competition
- Only “physics only” principles are used for describing a domain, making the PDDL planner totally domain-independent.

The PDDL Language

An example

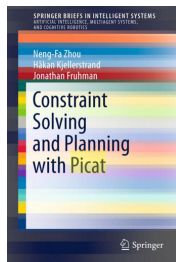
Example 1: A more-level Type Hierarchy

```
(:types movable loc — object
      truck cargo — movable)
(:predicates
  (at ?n — movable ?l — loc)
  (inPred ?c — cargo ?t — truck)
  (goal ?c — cargo ?l — loc)
)
(:action move
 :parameters (?t — truck ?l ?l1 — loc )
 :precondition (and
   (at ?t ?l)
 )
 :effect (and (not (at ?t ?l))
   (at ?t ?l1)
 )
)
```

The Picat Language

Introduction to Picat

- Developed in 2012 by Neng-Fa Zhou
- It aims to collect the main characteristics of various kind of programming language. Based on B-Prolog engine, but far more expressive and scalable.
- Tabling can be used to store the results of certain function calls in memory, allowing the program to do a quick table lookup instead of repeatedly calculating a value. It guarantees recursive program termination, preventing infinite loops and redundancy



The Picat Language

Planning in Picat

Picat provides to the user an interesting planner module:

- it allows the user to write domains that are at the same time more sophisticated and more compact than those we could obtain in PDDL, thanks to the expressiveness of the language
- we can mix deterministic and non-deterministic rules to model the domain as we want
- a *state* can be represented by any possible kind of term provided by Picat
- tabling to improve the plan creation, without user intervention

Planning in Picat

Action and goal check definition

Example 2: Final state check

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

Example 3: State transition

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


Planning in Picat

Plan finding techniques

- deep-first searches:
 - ▶ *best_plan_unbounded(S,Limit,Plan,PlanCost)*
 - ▶ *plan_unbounded(S,Limit,Plan,PlanCost)*
- resource-bounded searches:
 - ▶ *plan(S,Limit,Plan,PlanCost)*
 - ▶ iterative deepening with *best_plan(S,Limit,Plan,PlanCost)*
 - ▶ branch and bound with *best_plan_bb(S,Limit,Plan,PlanCost)*

Planning in Picat

Factored and Structured State Representation

- Factored State Representation is the only possible in PDDL, so it is one of the most known in planning.
- Structured representation goal is to make the *State* as small as possible by a massive usage of Picat expressiveness, because the smaller the *States* are the better is due to tabling.

Example 4: PDDL Factored State Representation from Nomystery

```
{at(c1, loc1), at(c2, loc2), at(t, loc3),  
connected(loc1, loc2), connected(loc2, loc3), truck(t)}
```

Example 5: Factored State Representation in Picat

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

Example 6: Structured State Representation in Picat

```
s(TruckLoc, TruckLoad, Cargo)
```

The Compiler

In this section we present the main contribution of the thesis, namely the definition and implementation of a tool for automatic conversion of PDDL files into Picat, developed in Picat.

- It provides support to many PDDL features, like object typing (not naturally supported by Picat planner), functions (numeric and not), calculated action costs, quantifiers and others, that are not even supported by some PDDL planners.
- It includes the *pi2pddl* parser written by Neng-Fa Zhou, only for the problem instances conversion
- Since the Factored Representation is the only available in PDDL, a *state* in a Picat translated domain takes the following form:

`s(PREDICATE_1, PREDICATE_2, ... , PREDICATE_N)`

- PDDL objects types are respresented by Picat rigid facts.

The Compiler

Discrimination between fluents and “rigids” predicates by a static analysis;
(*pd* ?*x1* ... ?*xn*) is translated as:

- *member(PD,(X1,...,XN))* if the predicate is a fluent.
- *PD1=select(PD0,(X1,...,XN))* if the predicate is a fluent and we have (*not(pd?x1...?xn)*) in the effect.
- *pd(X1,...,XN)* if the predicate is “rigid”

Lists to represent fluents and Picat rigid predicates to represent static predicates and PDDL types.

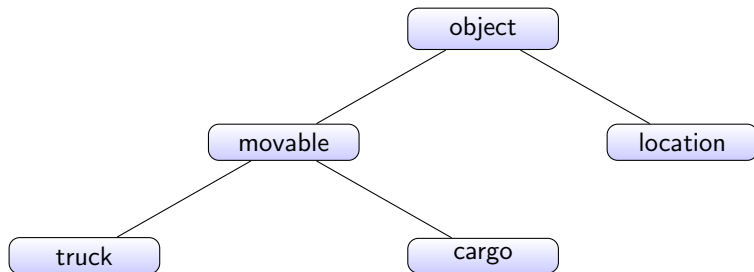
The Compiler

Typing

If our domain presents an hierarchy tree with more levels, some actions may not work properly, due to the extra information from PDDL *parameters* section missing in Picat:

```
(predicate (at ?t - movable ?l - loc) ... )  
(action move  
  (parameters (?t - truck ...
```

These ambiguities are resolved by binding *?t* to a *truck* with the rigid predicate *truck(T)* (only when strictly necessary)



The Compiler

Avoid Floundering

Negative preconditions in Picat can cause floundering; the compiler tries to resolve this issue:

- by moving them after the positive ones

and if it is not enough

- by adding “type” predicates for unbounded variables (like we did..)

The second operation is also carried out if after preconditions parsing some variables are still unbounded (this can occur while using forall quantifiers in effect section) as in the following action:

```
(: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))))))
```

The Compiler

A translated action

```
(:action move
:parameters (?t - truck ?l1 ?l2 - loc )
:precondition (and
    (at ?t ?l1)
    (connected ?l1 ?l2)
)
:effect (and (not (at ?t ?l1))
    (at ?t ?l2))
)
```

```
action(s(AT0,CARRIED0,INPRED0),NextState, Action, Cost),
truck(T),
select((T,L),AT0,AT1),
connected(L1,L2)
?=>
AT2 = insert_ordered_wod(AT1,(T,L2)),
NextState = $s(ATP2,CARRIED0,INPRED0),
Action = $act_move(T,L1,L2),
Cost=1.
```

The Compiler

Final state checking

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:

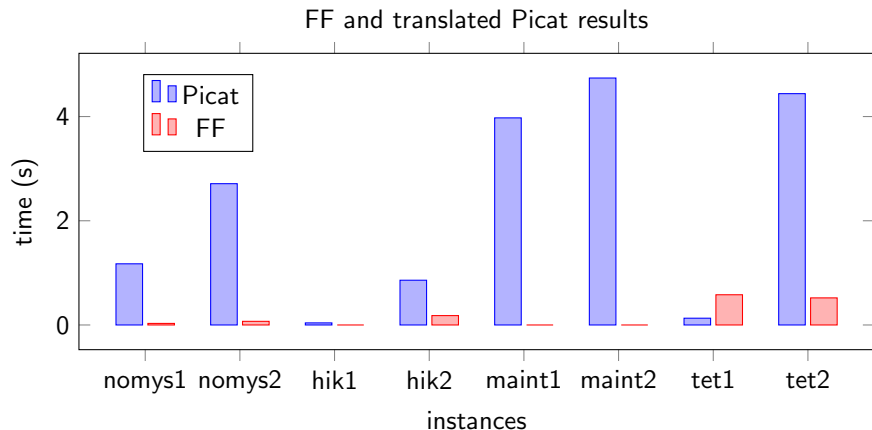
```
final(s(AT, IN)), goal(AT_GOAL, IN_GOAL),  
  subset(AT_GOAL, AT),  
  subset(IN_GOAL, IN) =>  
true.
```


Experimental results

- PDDL files, both domain and instances, refer to specifications from official IPC web site, while the Picat equivalents are translated by the compiler. Where available, also the structured ad-hoc versions of Picat will be tested.
- We will try all kinds of searches available in Picat, reporting bounds and execution times in seconds.
- The tests are executed on a notebook with a CPU Intel Core I5 4210h at 3.42 Ghz and 8 gigabytes of RAM.
- The PDDL planner employed in these tests is *Metric - FF 2.1*, a state-of-the-art PDDL planner declared Top Performer in the Strips Track of the 3rd International Planning Competition. It is implemented in C, and it is based on FF (Fast Forward), a forward chaining heuristic state space planner, awarded for Outstanding Performance at the 2nd International Planning Competition and Top Performer in the Strips Track of the 3rd International Planning Competition.

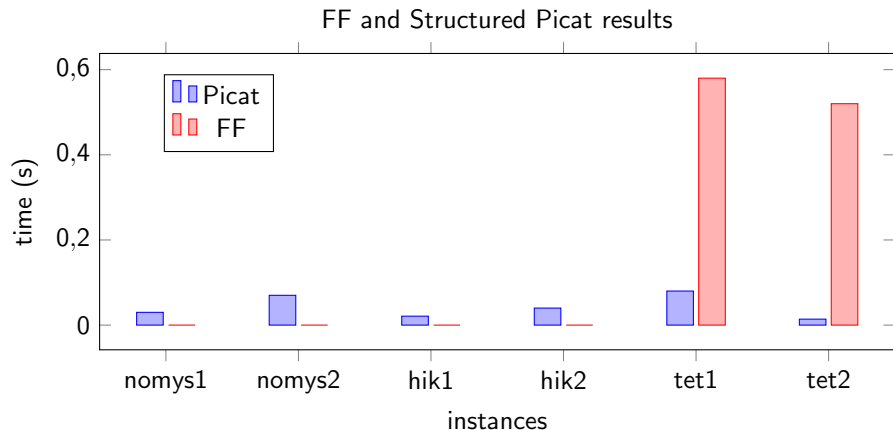
Experimental results

In the charts reported below we summed up the outcomes of all the problems, collecting different results from tested domains.



Experimental results

They confirms the well-known fact that generally in planning does not exist a perfect planner able to obtain the best results from all the domains and instances, but the performance of each one is very sensitive to the single case.



Conclusions and further work

- Results as a base for Picat programmers to create more compact and more performing encodings
- A good starting point for the automation of optimization stages

Thanks for your attention