Optimization of a compiler from PDDL to Picat (Short Paper)

Francesco Contaldo, Marco De Bortoli, and Agostino Dovier

Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Abstract. Picat is a new constraint logic programming language that has obtained promising results in international competitions. These results have been achieved thanks to several features. The most effective of them is an efficient handling of a tabling technique applied to search algorithms. A compiler from PDDL to Picat, which automatically enables to run PDDL models in Picat, has been recently developed. This paper describes a method which automatically optimizes the output of such a compiler, using a different representation of the states that better takes advantage of Picat's Tabling technique.

1 Introduction

One of the first and most important research areas in AI is planning. It deals with the search for a strategy, in order to resolve a specific problem, in a given world with fixed rules. The operation of finding this sequence of actions, performed 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 ancestor of modern planners is STRIPS (Stanford Research Institute Problem Solver [5]), the language used by Shakey the robot. It was developed at the end of the 60's and was the first robot that could reason about its own actions and plan what to do based on a given goal.

The popularity of planning was increasing, but there were no common guidelines or modeling style, at least until the 1998, when PDDL (Planning Domain Definition Language) was proposed [9]. Since then, also thanks to its use in the International Planning Competitions [1], PDDL and its extensions (e.g., [6]) are the reference for planning domain modeling.

Picat [11] is a recently designed rule-based, declarative programming language, provided with a large number of features inherited from several programming paradigms, such as imperative, scripting, logic and constraint programming, and external interfaces with constraint solvers, SAT solvers, and MIP solvers. One of the most successful features of Picat up to now is the planner module, that allows to obtain efficient results on IPC benchmarks. Benchmarks can be encoded directly in the planner module of Picat (as done, e.g., in [2, 10]); however, due to the relevance of PDDL in the community, it will be useful to directly use them within Picat. For this purpose a first compiler from PDDL to Picat (written in Picat) has been presented in [3].

In this paper we propose a method, inspired by *Planning Graph* technique [7], that optimizes the compiler in order to produce an encoding that better exploits the Picat's tabling capabilities. In particular it uses the Structured state representation (as opposed to Factored state representation used by PDDL).

2 Preliminaries

We assume the reader has a basic knowledge of Planning, PDDL, and Picat (Prolog-like) syntax.

State representation used in PDDL is known as *Factored Representation*: a *state* is identified by a set of atoms, i.e. a set of predicates denoting that some primitive properties of the world are true. An atom can be *rigid*, if it represents a never changing property (e.g. the fact that in location (x, y) there is a wall), or *fluent*, that can be false or true depending on the current *state* (e.g. the fact that in location (x, y) at time t there is a robot). Below there is an example of a factored representation (inspired by the problem known as *Nomystery*), encoded in PDDL and Picat, respectively:

```
{at(truck1,loc1),...,connected(loc1,loc2)}
$[at(truck1,loc1),..,connected(loc2,loc3)]
```

In a *Structured Rapresentation* the *state* is represented using internal data structures, such as lists or sets, and where it is possible it enables a reduction of the symmetries. The use of data structures is very suitable for the tabling technique used by Picat. In fact the states that are memorized during the plan seek can share common elements present in the structures used to represent them, reducing the usage of memory during the computation.

Considering the *Nomystery* domain described above, one of its possible structured state representations do not use the predicate `connect`, because it is a *rigid* predicate that describes the map and there is no need to replicate it in every state. However, it is possible to go further, memorizing the state simply as:

```
s({loc1,loc2,loc3})
```

In this state representation the three trucks are directly encoded using their positions, avoiding the presence of symmetric states during the search. Since all the trucks are identical there will be no significant difference among two equals states in which the position of two trucks is exchanged. E.g. one of the two states can be $at(truck1, loc1), at(truck2, loc2)$ and the other one can be $at(truck2, loc1), at(truck1, loc2)$. At the end "s" can have other arguments, e.g., storing the set of desired truck destinations, or other domain informations.

Picat implements two main search techniques to find a plan: depth-first and resource-bounded. Both of the previously mentioned approaches exploit the Picat's tabling technique that enables the planner to memorize the visited states and thus to avoid the re-computation of already-visited branches. In *depth-first*, the solver keeps applying actions until it finds either a dead end, an already-visited state or a goal state, otherwise it backtracks and it explores other branches. In the second approach the planner adds a resource amount value that is used by the planner as convenience measure to decide whether it is advantageous to re-explore an already visited failing state. [11].

3 Optimizing the compiler

The goal of the optimization is to automatically transform a model based on a *factored state representation*, inherited from PDDL and obtained using the compiler presented in [3], into a model based on a *structured state representation* that exploits the Picat's Tabling technique [2].

The main intention is to identify the *rigid* elements and the *fluent* elements inside each main predicate (i.e., those that appear in the effect of some action, the fluent one) in order to reduce the size of them. At this point it is fundamental to distinguish the meaning of the adjectives *rigid* and *fluent*. In the first section these two attributes are used to classify the predicates in the domain. So, in this case the *rigid* elements represent the static information inside a single predicate, while the *fluent* elements represent the dynamic information that evolves according to each action taken by the planner. As a matter of fact the predicate `at(truck,location)` from *Nomystery* encodes a static information: the name of the truck. Clearly the name of an entity is something that cannot evolve or change during the different explored states so it is possible to define it *rigid*. Instead the location of each truck is more meaningful, it can assume different values during the seek in order to find the goal configuration. Thus each main predicate will be represented only by its fluent elements and, where it is possible, some predicates are merged together if they were sharing the same rigid element at the beginning of the compiling phase. Then the state representation is changed accordingly, introducing a term whose arguments are sets of list, one for each new main predicate, and all the actions are modified to deal with the new data structures.

The underlying algorithm of the compiler can be split in two main parts. The first one is a preprocessing step which is inspired by the *Planning Graph* technique. This step aims to "ground" as predicates as possible, starting from the initial problem state an iterative and terminating procedure tries to apply a fixed number of actions in order to instantiate the larger possible number of predicates. In this iterative procedure an action is applied only once and it stops when a fixed point is reached: there is no new action applicable to the current state or all the actions have been already applied. When the computational step has finished the analysis one starts. The goal of this last step is to identify the *rigid* elements inside each predicates performing an arithmetic mean of the positions of the elements that differ among the same grounded predicates. Then the analysis is lifted to the whole problem level working with all the elements of all the predicates at same time. The main idea that is behind this is step is to built a DAG of dependencies among all the predicates elements, thus, for each main predicates in the domains, there exists a directed edge from its *rigid* elements and its remaining elements that become temporary *fluent*. At the end the DAG is used to modify the main predicates of the domain, holding the original definition of the *rigid* predicates. The obtained result should be a more compact state representation since the static elements are deleted from the predicates, then the number of the predicates used can be reduced performing a merge between two predicates that shared the same rigid elements.

4 Results and conclusions

The effect of the compiler optimization has been tested on three models as benchmarks: *Floortile, Tetris, Nomystery*, from the IPC list of benchmarks.

Nomystery Instance	**Structured Representation**		**Factored Representation**	
	bound	time	bound	time(s)
Nomystery_0	6	1	6	2.70
Nomystery_1	9	18.96	9	34.04
Nomystery_2	13	159.05	13	376.13
Nomystery_3	15	125.58	15	421.43
SUM		304,59		834,30

Tetris Instance	**Structured Representation**		**Factored Representation**	
	bound	time	bound	time(s)
Tetris_0	2	0	2	0
Tetris_1	9	0.15	9	0.16
Tetris_2	10	0.05	10	0.352
Tetris_3	16	0.22	16	0.587
Tetris_4	16	5.39	16	11.554
Tetris_5	16	13.473	16	201.157
SUM		19.283		213.81

Floortile Instance	**Structured Representation**		**Factored Representation**	
	bound	time	bound	time(s)
Floortile_0	14	0.005	14	0.007
Floortile_1	15	0.057	15	0.088
Floortile_2	12	0.17	12	0.21
Floortile_3	14	3.09	14	3.194
Floortile_4	24	10.34	24	12.95
Floortile_5	26	309.41	26	308.37
SUM		323.072		325.019

Table 1. Iterative Deeping Results (Factored representation is what returned by the previous compiler, Structured Representation is after our automatic optimization)

The computational results, reported in Table 1, highlight an improvement of the performance in all instances tested of *Nomystery*, *Tetris* and for the most one of *Floortile*. Focusing on the global time, obtained by the summation of all the times for a specific problem, it can be observed how all the problems encoded using the optimization have better performance than the ones encoded with the factored representation. As for *Nomystery* and *Tetris* we have obtained an improvement of 60% and 90%, respectively. On the other hand, concerning the *Floortile* problem we have gained only 2 seconds in the global measurement. The running time required by the optimization process is on average around the 350 ms, thus, it is not an overhead for medium-big size problems.

We have also tested direct, manually optimized, Picat encodings of the three benchmarks. The running time on the instances tested are negligible (below 50ms — same results as state-of-the-art planners). Other work is needed to automatize the compiler process to lead to encodings with the same performances.

Acknowledgments. We thank Roman Barták and Neng-Fa Zhou for their help in various stages of this work. A. Dovier is partially supported by INdAM-GNCS 2015–2017 projects.

References

1. International planning competitions web site. http://ipc.icaps-conference.org/. Accessed: 2017-05-10.
2. Roman Barták, Agostino Dovier, and Neng-Fa Zhou. On modeling planning problems in tabled logic programming. In M. Falaschi and E. Albert, editors, *Proc. of PPDP*, Siena, Italy, July 14-16, 2015, pages 31–42. ACM, 2015.
3. Marco De Bortoli, Roman Barták, Agostino Dovier, and Neng-Fa Zhou. Compiling and executing PDDL in picat. In C. Fiorentini and A. Momigliano, editors, *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016.*, volume 1645 of *CEUR Workshop Proceedings*, pages 132–147.
4. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Perspectives on logic-based approaches for reasoning about actions and change. In M. 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*, LNCS 6565, 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. B. Cenk Gazen and Craig A. Knoblock. Combining the expressivity of UCPOP with the efficiency of graphplan. In S. Steel and R. Alami, editors, Proc. of *ECP'97, Toulouse, France*, LNCS 1348, pages 221–233. Springer, 1997.
8. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.
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. Neng-Fa Zhou, Roman Barták, and Agostino Dovier. Planning as tabled logic programming. *TPLP*, 15(4-5):543–558, 2015.
11. Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat.* Springer Briefs in Intelligent Systems. Springer, 2015.