Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

Domain Modeling for Planning as Logic Programming

Roman Barták, Jindřich Vodrážka
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Abstract

Planning as programming is an approach to automated planning, where the planning domain model is expressed as a program in some (declarative) programming language. Then the modeler can exploit all features of that language to encode control knowledge important for efficient planning. In this paper we study these features in the logic programming language Picat and its `planner` module. In particular, we use two planning benchmarks, Nomystery and Childsnack, to compare factored and structured representations of states extended by encodings of control knowledge.

Automated planning deals with the problem of finding a sequence of actions to reach some goal state from a given initial state. Since the Shakey project, the STRIPS planning model (Fikes and Nilsson 2002) is used to describe planning domains. In the STRIPS model, actions are described as entities requiring some properties of the world (preconditions) for their execution and changing some world properties (effects) after execution. The STRIPS formalism evolved into the Planning Domain Definition Language (PDDL) (McDermott 1998) introduced for International Planning Competitions (IPCs) and being a de-facto standard planning domain modeling language.

Planners based on heuristic search are currently the most widely studied so-called domain independent planners based on PDDL. These planners are expected to use "pure" domain models with information about how actions change the world but with no extra information about how these actions contribute to reaching the goal. Despite big progress in domain-independent planning such planners are still rarely used in practice due to efficiency issues. Therefore methods to enrich the planning domain model with domain-specific information useful for planning (Haslum and Sholz 2003) were proposed to improve efficiency of planners. There are planners that use state-centric domain control knowledge specified in temporal logic (Bacchus and Kabanza 2000; Kvarnström and Magnusson 2003). Action-centric control knowledge can be encoded in hierarchical task networks (Nau et al. 2003) and it is also possible to automatically recompile similar kind of action control knowledge into PDDL (Baier et al. 2007). The major disadvantage of all these domain-dependent approaches is big modeling burden put on the human modeler.

Another approach to improve efficiency of planning is based on direct encoding of planning problems in a modeling formalism closer to the solver. This approach is called *planning as programming* as the planning domain model is represented as an "executable" program to find the plan in some programming language. For example, planning is closely related to logic programming – PLANNER (Hewitt 1969), which was designed as a language for proving theorems and manipulating models in a robot, is perceived as the first logic programming language. Despite the amenability of Prolog to planning, Prolog is no longer a competitive tool for planning. Recently, the logic programming system Picat and its `planner` module was shown to solve realistic planning problems beyond capabilities of state-of-the-art PDDL planners (Barták and Zhou 2014). When modeling the planning domain properly this system can beat domain-independent planners on IPC benchmarks (Zhou et al. 2015) and is on par with domain-dependent planners (Barták et al. 2015) while preserving the size of models comparable to PDDL models. Nevertheless, so far there was a little understanding of how the domain modeling techniques contribute to efficiency of the Picat `planner` module.

In this paper we address the problem of understanding how domain modeling influences efficiency of planning in the Picat programming language. In particular we present various encodings of two IPC planning domains, Nomystery and Childsnack, with factored and structured representations of states and enhanced by control knowledge. We do experimental comparison of the models using two search techniques used by the Picat `planner` module to demonstrate how structured representation of states and various encodings of control knowledge contribute to efficiency.

Background on (Picat) Planning

Automated planning deals with the problem of finding a sequence of actions called a plan that changes the given state of the world to a state satisfying a certain goal condition. This so called classical planning corresponds to the problem of finding a path in a (huge) directed graph, where nodes describe states of the world and arcs correspond to state transitions via actions. Formally, let $\gamma(s, a)$ describe the state after applying action a to state s, if a is applicable to s (otherwise the function is undefined). Then the planning task is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ called a *plan* such that, s_0 is the initial state, for each $i \in \{1, \ldots, n\}$, a_i is applicable to state s_{i-1}, $s_i = \gamma(s_{i-1}, a_i)$, and s_n is a goal state. For solving cost-optimization problems, a non-negative cost is assigned to each action and the task is to find a plan with the smallest cost. The major difference from classical path-finding is that the state space for planning problems is enormous and does not fit in memory. Hence a compact representation of states and actions (and state transitions) is necessary. This representation is called a *domain model*.

Since the time of Shakey The Robot the *factored representation* of states is used, which has been reflected later in the design of the Planning Domain Definition Language (PDDL) (McDermott 1998). In this representation a state consists of a vector of attribute values and actions are changing values of certain variables (action effect) while requiring values of some attribute variables as preconditions. In a *structured representation* a state model describes objects possibly with attributes as well as relations between the objects. Action models based on first-order logic are close to this representation, but we are not aware of any widely-used planning domain modeling language based on structured representation that leads to efficient planners. The Picat `planner` module supports both factored and structured representations of states – the state is represented by any term.

The detailed description of Picat domain models can be found in (Barták et al. 2015). Briefly speaking the planning domain model in Picat is expressed as a set of action rules describing the transition function γ in the form:

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

This rule gets some `State` as its input and should produce `NextState` as its output together with some description of `Action` used for the transition and non-negative `Cost` of that action. If the plan's length is the only interest, then this cost equals one. The above pseudo-code gives the typical action rule, where `precondition` is evaluated first together with optional `control_knowledge` telling when the rule can be applied. These are arbitrary Picat calls. Then the effect of the action is described when building the `NextState` together with setting the action `Cost`.

Note, that in the above pseudo-code we used a non-deterministic version of the state-transition rule (`?=>`). Like in Prolog, the domain model consists of a set of rules that are tried in the top-down order (hence the order of rules matters). It means that during backtracking the next applicable rule (action) is explored provided that it is backtrackable. The domain modeler can specify that a given action rule should be applied to a given state and if its application does not lead to a (best) plan then the next action rules are not tried. This is done by using deterministic rules (`=>`). The Picat planner uses two search approaches to find optimal plans. Both of them are based on depth-first search with tabling and they correspond to classical forward planning by starting in the initial state and finding an action applicable to the state etc. until a plan is found (alternatives are explored upon backtracking).

The first approach starts with finding any plan using the depth-first search. The initial limit for plan cost can (optionally) be imposed. Then the planner tries to find a plan with smaller cost so a stricter cost limit is imposed. This process is repeated until no plan is found so the last plan found is an optimal plan. This approach is very close to *branch-and-bound* technique (Land and Doig 1960). Note that tabling is used there – the underlying solver remembers the best plans found for all visited states (with a given cost limit) so when visiting the state next time, the plan is retrieved rather than looked for again.

The second approach exploits the idea of iteratively increasing the cost limit and looking for a plan with a given cost limit. The approach first tries to find a plan with cost zero. If no plan is found, then it increases the cost by 1. In this way, the first plan that is found is guaranteed to be optimal. Unlike the *IDA* search algorithm* (Korf 1985), which starts a new round from scratch, Picat reuses the states that were tabled in previous rounds.

Picat uses linear tabling to memorize answers to calls which implicitly prevents loops and brings properties of graph search (not exploring the same state more than once) to classical depth-first search used by Prolog-like languages. Tabling can also be used to remember "best" answers so it can be exploited to find optimal plans. Picat supports so called resource-bounded search that prevents exploring paths exceeding a given cost limit (Zhou et al. 2015).

Domain Models

To demonstrate capabilities of the Picat planner module we have selected one traditional planning benchmark domain – Nomystery from IPC 2011 – and one recent domain – Childsnack from IPC 2014. The major criterion for selection was existence of natural control knowledge – in other words there is some "common sense" guidelines that humans would use to solve problems in these domains. We also looked for domains with a small number of operators as we needed to manually design several domain models.

In this section, we will explain factored and structured representations for each domain and we will describe encoding of control knowledge for these domains. The factored representations basically mimic the original PDDL encodings of the domains. The structured representations as well as control knowledge are tailored for each domain separately though the reader can observe some unifying principles.

One of the motivations for this research is showing how efficient domain models look like. Though we use Picat for domain modeling, we believe that similar knowledge (till some extent) can be encoded in PDDL. As PDDL uses a factored representation of states and actions with preconditions and effects only, we encoded the control knowledge for the factored representation also in the PDDL style.

Nomystery

In the Nomystery domain, there is a single truck with unlimited load capacity, but with a given (limited) quantity of fuel. The 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 so the initial fuel quantity limits how far the truck can move (no refueling is assumed).

The factored representation uses predicates `at/2` to define locations of the truck and of cargo items, and predicates `in/2` telling that a cargo item is loaded in the truck. There is also a single predicate `fuel/2` describing a fuel level of the truck. In Picat we encode these predicates as ordered lists containing the predicate arguments (ordered lists represent sets in a unique way). Together, we represent the state using a triple {`Fuel,AtPreds,InPreds`}, where `Fuel` is the current fuel level of the truck. Recall that there is a single truck so in each state there is a single predicate `fuel(Truck,Fuel)`. This is how the action *load* looks like in the Picat domain model with the factored representation of states (cargo C is loaded to truck T at location L):

```
action({Fuel,AtPred,InPred},NextState,
           Action,ActionCost),
  truck(T),
  member([T|L],AtPred),
  select([C|L],AtPred,RestAtPred), C!=T
?=>
  Action = $load(C,T,L),
  ActionCost = 1,
  NextState = {Fuel,RestAtPred,NewInPred},
  NewInPred = insert_ordered(InPred,[C|T]).
```

The above rule can be obtained automatically from the PDDL model. The predicate `truck(T)` is stored in the Picat database and represents the type of object in PDDL. Checking validity of action preconditions is realized via set operations `member` and `select` depending on whether the validity of the predicate will be changed (`select`) or not (`member`). Action effects are modeled by adding the new valid predicates to the state via `insert_ordered`.

The structured representation focuses on removing object symmetries by representing objects via their (possibly changing) attributes rather than via their names. For example, we can represent each cargo item as a pair `[CurrLoc|GoalLoc]` describing the current location and goal location of the item. The current state is then represented using the tuple {`Loc,Fuel,LCGs,WCGs`}, where `Loc` is the location of the truck, `Fuel` is the truck's fuel level, `LCGs` is an ordered list of destinations of loaded cargo items, and `WCGs` is an ordered list of waiting cargo items, each item is represented as we described above. Similar symmetry breaking properties can be achieved in PDDL by using "resource predicates". For example, the predicate `at(Loc,Dest,Q)` may describe how many packages are waiting at a given location for being delivered to a given destination.

This is how the action *load* looks like when using the structured state representation. Notice that the cargo item is identified in the action by its destination so some post-processing is necessary to put object names back to the plan.

```
action({Loc,Fuel,LCGs,WCGs},NextState,
           Action,ActionCost),
  select([Loc|CargoGoal],WCGs,WCGs1)
?=>
  Action = $load(Loc,CargoGoal),
  ActionCost = 1,
  NextState = {Loc,Fuel,LCGs1,WCGs1},
  LCGs1 = insert_ordered(LCGs,CargoGoal).
```

The control knowledge is expressed via ordering of action rules and using deterministic rules and via extra conditions for action applicability. The control knowledge for the Nomystery domain exploits information that there is no capacity restriction of the single truck. First, when the cargo item is delivered, it can be removed from the state representation because there will be no other actions related to this item (any action that manipulates an already delivered item only enlarges the plan). Second, the truck should load all cargo items at its current location before driving somewhere else. If any item is left there then the truck needs to return to that location to load that item later, which only enlarges the plan (recall, that there is a single truck in the domain). This can be achieved by putting the load action before the driving action and setting the action rule for loading to be deterministic (if anything can be loaded then the load action is used first). Finally, any loaded cargo is kept loaded until the truck reaches cargo's destination. This is achieved by unloading the cargo item only at its destination. We can make this rule even stronger – the truck does not leave a given location until all loaded cargo items destined to this location are unloaded. Together, we put the unload action before the drive action, we add extra condition that an item is unloaded only at its destination, and we make the action rule for unloading deterministic so all cargo items (for current location) are unloaded before trying another action. The only remaining non-determinism to be explored by search is where the truck should go.

If we want to keep the PDDL structure of actions with the factored representation and without deterministic rules then we can do it by strengthening action preconditions by using information about the goal. We can store this information in the Picat database (similarly to object types) and then actions can exploit it. This is how the action *load* looks like with encoded control knowledge in the PDDL style:

```
action({Fuel,AtPred,InPred},NextState,
           Action,ActionCost),
  truck(T), goal(G),
  member([T|L],AtPred),
  not (member([C1|T],InPred),
      member([C1|L],G)), % nothing to unload
  select([C|L],AtPred,RestAtPred), C!=T,
  not member([C|L],G)
?=>
  Action = $load(C,T,L),
  ActionCost = 1,
  NextState = {Fuel,RestAtPred,NewInPred},
  NewInPred = insert_ordered(InPred,[C|T]).
```

The action says that loading is possible only if there is nothing to unload (no loaded cargo item destined for the current location) and if the cargo item to be loaded belongs to a different location. Note that the model of actions is now dependent on the type of goal and when a different type of goal is used (for example the goal is to move the truck somewhere) then the model of actions would be different.

To demonstrate that the Picat model of actions is close to the PDDL model, this is how the action $load$ looks like in the original PDDL model:

```
(:action LOAD
  :parameters
   (?p - package
    ?t - truck
    ?l - location)
 :precondition
   (and (at ?t ?l) (at ?p ?l))
 :effect
   (and (not (at ?p ?l)) (in ?p ?t)
        (increase (total-cost) 1))
)
```

Childsnack

The task in the Childsnack domain is to plan how to make and serve sandwiches for a group of children in which some are allergic to gluten. There are two actions for making sandwiches from various ingredients. The first one makes a sandwich and the second one makes a sandwich taking into account that all ingredients are gluten-free. There are also actions to put a sandwich on a tray, to move the tray between locations, and to serve sandwiches. Problems in this domain define the ingredients to make sandwiches at the initial state. Goals consist of having all kids served with a sandwich to which they are not allergic.

The factored representation again mimics the original PDDL encoding that uses predicates to identify bread `at_kitchen_bread/1`, content `at_kitchen_content/1`, and sandwiches `at_kitchen_sandwich/1` in kitchen, sandwiches placed on trays `on_tray/2`, locations of trays `at/2`, and kids that have been served `served/1`. There are some rigid predicates defining that the content and bread is gluten-free, while this information must be kept as a fluent for sandwiches (`no_gluten_sandwich/1`). Notice that names of sandwiches are used to identify sandwiches so predicates `notexist/1` are used to indicate that there is a prospective sandwich. Hence the "life" of sandwich starts as `notexist` which then changes to `at_kitchen_sandwich` possibly accompanied by `no_gluten_sandwich`, followed by `on_tray`, and finally disappearing after being served to a child. In summary, the state is represented as tuple {`Bread, Content, Sandwiches, OnTray, SwNoGluten, SwNames, TrayLocs, Childs`}, where each component is an ordered list of names (constants) or pairs of names (for `OnTray` and `TrayLocs`). We used a complementary representation of predicate `served/1` so the list `Childs` represents not-yet served children. This is how the action put_on_tray looks like:

```
action({Bread,Content,Sandwiches,OnTray,
        SwNoGluten,SwNames,Trays,Childs},
        NextState,Action,ActionCost),
  member({Tr,kitchen},Trays),
  select(Sw,Sandwiches,RSandwiches)
?=>
  Action = $put_on_tray(Sw,Tr),
  NextState={Bread,Content,RSandwiches,
             NewOnTray,SwNoGluten,SwNames,
             Trays,Childs},
  NewOnTray=insert_ordered(OnTray,{Sw,Tr}),
  ActionCost = 1.
```

The structured representation removes object symmetries by ignoring names of bread, content, and sandwiches, and representing each of them using either constant `no_gluten` or constant `gluten`. The not-yet made sandwich is represented by constant `free`. For children we can also ignore the names so we can represent each children using a pair {`Loc,Type`}, describing location and "type" of children (`gluten` or `no_gluten`). Also the trays are represented by their location and loaded sandwiches as a pair {`Loc,Load`}, where `Loc` is the current location of the tray and `Load` is an ordered list of sandwiches loaded to that tray (recall that sandwich is only identified by its type). So the path of the sandwich starts as `free`. When the sandwich is made, its identification changes to `no_gluten` or `gluten`. If the sandwich is put on tray, it is placed to the corresponding `Load` list, and finally, if the sandwich is served then it disappears from the state. In summary, the structured state is represented by a tuple {`Bread, Content, Sandwiches, Trays, Childs`}, where the first three components are ordered lists of object types, `Trays` is an ordered list of pairs {`Location,Load`}, and `Childs` is an ordered list of pairs {`Loc,Type`}. This is how the action put_on_tray looks like:

```
action({Bread,Content,Sandws,Trays,Childs},
        NextState,Action,ActionCost),
  select(Type,Sandws,RSandws),
  Type!=free,
  select([kitchen|Load],Trays,RTrays)
?=>
  Action = $put_on_tray(Type),
  NextState = {Bread,Content,RSandws,
               NewTrays,Childs},
  NewTrays = insert_ordered(RTrays,
       [kitchen|insert_ordered(Load,Type)]),
  ActionCost = 1.
```

There is a deterministic method to find an optimal plan. First, for each children we need to make a sandwich. Gluten-free sandwiches are made first to ensure that there is enough gluten-free bread and content. Note that this requires to modify the representation of children as we need to distinguish between children with sandwich ready for them and children with no sandwich made for them yet. The factored representation can be extended by a new predicate for this property; the structured representation can add one argument to the model. Only when all sandwiches are made, they are all placed to a single tray (that may need to be moved to kitchen first). As no parallel plans are assumed, using more trays will not shorten the plan and hence a single tray is enough. That tray is then moved between locations and all children in each location are served before moving to another location. We used non-deterministic selection of the next location to visit to include some search decisions on the model.

The factored representation with control knowledge for put_on_tray in Picat is identical to the above presented rule – only the rule is made deterministic (=>) and placed after the action rules for $make\_sandwich$ and before the action rule for $move\_tray$ to ensure that loading is done after all sandwiches are made and before the tray leaves kitchen. Using the determinism also implicitly removes symmetries between sandwiches as they are loaded in a fixed order. If deterministic rules cannot be exploited then we need to break all these symmetries explicitly by assuming some order of objects (`@<`) as the following code shows:

```
action({Bread,Content,Sandwiches,OnTray,
       SwNoGlut,SwNames,Trays,Childs,Ready},
       NextState,Action,ActionCost),
  Childs = [],
  member({Tr,kitchen},Trays),
  not (member({Tr1,kitchen},Trays),Tr1@<Tr),
  select(Sw,Sandwiches,RSandwiches),
  not (member(Sw1,RSandwiches), Sw1@<Sw)
?=>
  Action = $put_on_tray(Sw,Tr),
  NextState={Bread,Content,RSandwiches,
             NewOnTray,SwNoGlut,SwNames,
             Trays,Childs,Ready},
  NewOnTray=insert_ordered(OnTray,{Sw,Tr}),
  ActionCost = 1.
```

The rule checks that sandwiches for all children are made (`Childs=[]`) and then it selects the first tray in the kitchen and the first sandwich to be loaded (the conditions say that no tray/sandwich with smaller id satisfies the conditions).

Experimental Evaluation

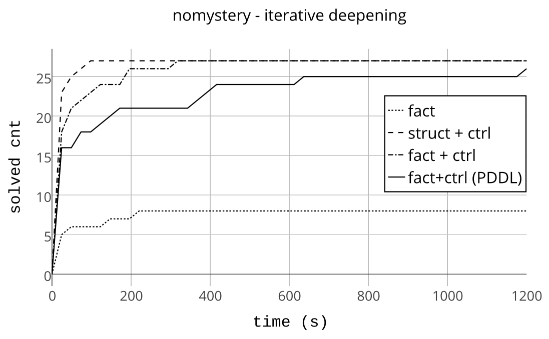
To show how presented modeling techniques influence efficiency of planning, we experimentally compared them using the existing problems from IPC. For each problem, we limited runtime to 30 minutes (if exceeded then the problem is treated as unsolved) and memory to 1GB. The experiments run on a computer with Picat 1.4 and Intel® Core^{TM} i7-960 running at 3.20GHz with 24 GB (1066 MHz). Table 1 shows the number of problem instances per domain and the number of instances solved optimally by our best model with structured representation and control knowledge. To demonstrate efficiency of Picat models in comparison with state-of-the-art planners we also include the number of problem instances solved optimally by best performing planner participating at the respective IPC. For the Nomystery domain it was Fast Downward Stone Soup 1 (but note that there were only 20 problem instances included for the Nomystery domain at IPC 2011 whereas our set of benchmarks contains 10 more instances). For the Childsnack domain the best planner (for this domain) was dynamic-gamer. Both PDDL planners run on a comparable computer (FSS1 on Intel Xeon 2.93 GHz with 6 GB and dynamic-gamer on AMD Processor 2.39 Ghz with 4 GB) with 30 minutes runtime limit.

We will present now the comparison of the following Picat models: pure factored representation, factored representation with control knowledge, structured representation with control knowledge, and factored representation with control knowledge in the PDDL style. Recall that the Picat planner uses either branch-and-bound or iterative deepening form of search so we will present the results separately for these two search approaches.

Table 1: The number of problem instances.

domain	#instances	#optimal	#state-of-the-art
Nomystery	30	30	"20" (FFSS1)
Childsnack	20	20	10 (dynamic-gamer)

Figure 1 shows how the number of problems solved optimally depends on time for the Nomystery domain; Figure 2 shows the same for the Childsnack domain (no problem was solved using the pure factored representation and the difference between the models with control knowledge are indistinguishable). The results are as expected – adding control knowledge helps significantly. The experiment also confirms that exploiting stronger modeling techniques such as structured representation of states, ordering of actions, and deterministic actions is beneficial. If we encode the same control knowledge in the PDDL style (via action preconditions), then the extra overhead decreases efficiency of planning.



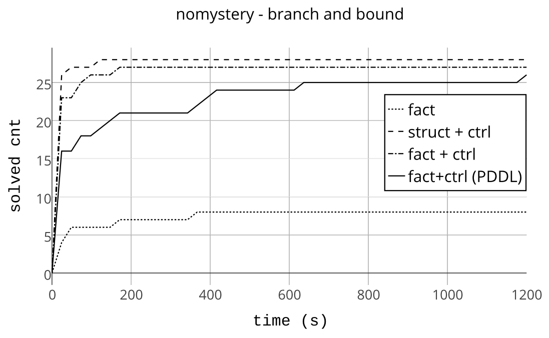


Figure 1: The number of problems solved within a given time for iterative deepening (top) and branch-and-bound (bottom) for the Nomystery domain.

Conclusions

Research in automated planning is dominated by interest in domain-independent planning techniques. Despite a big progress in recent years, these techniques are still far from being applicable to practical problems in areas such as computer games and robotics. Planning via programming might

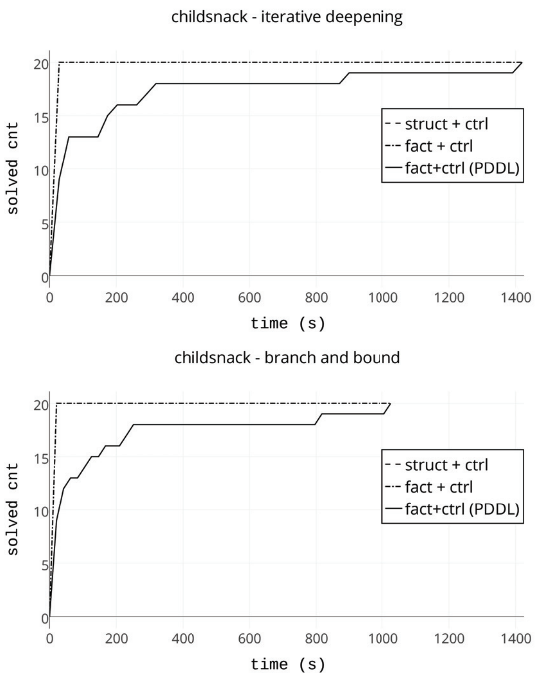


Figure 2: The number of problems solved within a given time for iterative deepening (top) and branch-and-bound (bottom) for the Childsnack domain.

be an approach to give practical efficiency for solving real-life problems without adding extra burden to planning domain modeling. In comparison to HTN the modeler is not required to provide a global structure of the plan and can express just local conditions for action applicability (see the Nomystery domain) though it is also possible to specify plan structure via suggested action sequences (see the Childsnack domain). The Picat models are also much smaller than models with control rules (Barták et al. 2015).

In this paper, we sketched the major ideas behind modeling planning domains in Picat. We gave examples of factored representation that can be obtained by direct translation of PDDL domains. We also presented structured representation that is more compact and can remove symmetries by representing objects via their properties rather than via their names. Both representations can be extended by control knowledge whose encoding exploits properties of Picat programming, namely using the order of action rules and deterministic selection of rules. We also showed that control knowledge can be encoded in the PDDL style though the performance is not as good as direct encoding. The initial experiments with encoding domain control knowledge in PDDL showed similar behavior – improvement of performance but not as significant as with the Picat models (Chrpa and Barták 2016). The models are not much larger than original PDDL models while achieving better efficiency by exploiting action-oriented control knowledge and object symmetry breaking. The major open question is how to automatically enhance the model for example by removing object symmetries (Riddle et al. 2015) and by discovering useful control knowledge.

Acknowledgements. Research is supported by the Czech Science Foundation under the project P103-15-19877S.

References

Bacchus F. and Kabanza F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191.

Baier J.; Fritz C.; and McIlraith S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling – ICAPS*, 26–33.

Barták R. and Zhou N-F. 2014. Using tabled logic programming to solve the Petrobras planning problem. *Theory and Practice of Logic Programming*, 14(4-5):697–710.

Barták R.; Dovier A.; and Zhou N-F. 2015. On modeling planning problems in tabled logic programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming – PPDP'15*, 32–42.

Chrpa, L.; Barták R. 2016. Guiding Planning Engines by Transition-based Domain Control Knowledge. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning – KR 2016*.

Fikes R. E. and Nilsson N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2 (3-4): 189–208.

Haslum O. and Scholz U. 2003. Domain knowledge in planning: Representation and use. In *ICAPS Workshop on PDDL*.

Hewitt C. 1969. Planner: A language for proving theorems in robots. In *Proceedings of IJCAI*, 295–302.

Korf R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1), 97–109.

Kvarnström J. and Magnusson M. 2003. Talplanner in the third international planning competition: Extensions and control rules. *J. Artificial Intelligence Research (JAIR)*, 20:343–377.

Land A. H. and Doig A. G. 1960. An automatic method of solving discrete programming problems. *Econometrica* 28(3): 497–520.

McDermott D. 1998. The planning domain definition language manual. CVC Report 98-003, Yale Computer Science Report 1165.

Nau D.S.; Au T-Ch.; Ilghami O.; Kuter U.; Murdock J.W.; Wu D.; and Yaman F. 2003. SHOP2: an HTN planning system. *J. Artificial Intelligence Research* (JAIR), 20:379–404.

Riddle P.; Barley M.; and Franco S. 2015. Automated Transformation of Problem Representations. In *Proceedings of The 8th Annual Symposium on Combinatorial Search*.

Zhou N-F.; Barták R.; and Dovier A. 2015. Planning as Tabled Logic Programming. *Theory and Practice of Logic Programming*, 16(4-5): 543–558.