On modeling planning problems in tabled logic programming

Roman Barták Agostino Dovier Neng-Fa Zhou

June 12, 2015

Abstract

Current research in planning focuses mainly on so called domain independent models using the Planning Domain Description Language (PDDL) as the domain modeling language. This declarative modeling approach embraces the idea of a physics-only model describing how actions change the world. However, PDDL omits information about why and when the actions should be applied to reach the goal, which significantly decreases the practical applicability of PDDL. There exist approaches such as Hierarchical Task Networks (HTN) and control rules that add this type of information to the model with the pay-off of increased efficiency but also with the downside of increased complexity and code sizes.

In this paper we propose a modeling framework for planning problems based on tabled logic programming that exploits a planner module in the Picat language. This modeling framework supports structured description of world states as well as inclusion of control knowledge and heuristics in the action model. By using problems from the International Planning Competition, we experimentally demonstrate that this modeling framework achieves results comparable to planners with control rules and HTN while keeping the size of the domain model much smaller. We also show that it gives much better solving efficiency than the state-of-the-art domain-independent PDDL planners.

1 Introduction

Shakey the robot [20] brought significant advancement to Artificial Intelligence (AI). It was the first robot that was able to reason about its own actions. It used the STRIPS (Stanford Research Institute Problem Solver) automated planner [6] to plan its actions based on a given goal. STRIPS is now referred as the formal specification language for the STRIPS planner and it is the base for most modern modeling languages for expressing automated planning problems. Its core idea is describing actions via pre-conditions and post-conditions (effects) that are sets of predicates used in the description of world states. The most famous AI modeling language based on STRIPS is the Planning Domain Definition Language (PDDL) [18], which was popularized thanks to its usage in the International Planning Competitions (IPC) [11]. Separating the planning domain model from the planning algorithm brought a lot of research in the area of so-called domain independent planners. However, almost no research was done in the area on how to actually describe planning domains. Moreover, despite intensive research the cutting-edge PDDL domain-independent planners still suffer from efficiency issues, which is the main reason why PDDL planners are rarely used in practice.

In this paper, we propose a declarative planning-domain modeling framework based on tabled logic programming, and we argue that modeling should play a much stronger role in automated planning. Planning and logic programming are two closely related areas. PLANNER [10], 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, tabling has been successfully used to solve specific planning problems such as Sokoban [26], the Petrobras planning problem [3], and several planning problems used in ASP competitions [27]. Tabling [24, 25] is a technique used in logic and functional programming systems, which caches the results of certain calculations in memory and reuses them in subsequent calculations through a quick table lookup. Like state marking used in search algorithms, tabling can prevent the same state from being expanded more than once during search. Based on experience with the ad-hoc planners we propose a generic approach for planning domain modeling in tabled logic programming. This approach utilizes the concept of state transitions and it is more flexible than PDDL by supporting structured representation of states and including control knowledge in the action model. By experimental comparison with the state-of-the-art optimal domain-independent planner SymbA [Torralba et al., 2014] and also with planners exploiting domain information via control rules [1] and hierarchical task networks [19] we demonstrate practical efficiency of the modeling framework.

The major contributions of this paper are: a novel planning domain modeling framework based on state transitions, guidelines for domain modeling, and experimental justification of efficiency of the modeling framework. As far as we know, no guidelines have been given before for domain modeling.

The paper is organized as follows. We first give some background information on planning. Then we describe the Picat programming language and its planner module, which is behind our modeling framework, and the syntax needed by the proposed modeling framework. After that, we focus on general principles of modeling planning domains in the proposed framework. We then apply these principles to model a specific planning domain, namely Cave Diving, from the International Planning Competition 2014 [11]. The paper is concluded by experimental comparison of the proposed Picat models with encodings for other state-of-the-art planning systems.

2 Background and Related Works

Classical AI planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state. Typically, the world state is described as a set of predicates that hold in the state, such as $at(truck, loc)$ saying that *truck* is located at *loc*. Actions are described using a triple $(Prec, Eff^+, Eff^-)$, where *Prec* is a set of predicates that must hold for the action to be applicable (preconditions), Eff^+ is a set of predicates that will hold after performing the action (positive effects), and finally Eff^- is a set of predicates that will not hold after performing the action (negative effects). We assume that $Eff^+ \cap Eff^- = \emptyset$. Formally, action a is applicable to state s if $Prec(a) \subseteq s$. The result of applying action a to state s is a new state $\gamma(s,a) = (s \setminus Eff^-(a)) \cup Eff^+(a)$. The set of predicates together with the set of actions – both sets are finite – is called a *planning domain*. The goal is specified as a set of predicates that must hold in the goal state, that is, if g is a goal then any state s that satisfies $g \subseteq s$ is a goal state. A *planning problem* is defined by the planning domain, the initial state s_0 and the goal g. The planning task is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ called a *plan* such that, for $i \in \{1, \ldots, n\}$, a_i is applicable to the state s_{i-1}, where s_0 is the initial state and $s_i = \gamma(s_{i-1}, a_i)$, and, furthermore, $g \subseteq s_n$.

The above view of planning is reflected in the most widely used domain-modeling language PDDL (Planning Domain Definition Language) [18]. This is a LISP-based language for encoding planning domains and problems (see Appendix A for an example). This language inherently uses the *factored representation* of states, which are represented as sets of values that might change. In the above description, we model states as sets of predicates that are true in these states. Modern planners are usually based on multi-valued state variables [2], but this is still a factored representation. A nice property of the factored representation of states is that the transition function γ as described above naturally covers the *frame problem* – values (predicates) that are not explicitly changed by the action remain the same. In this paper, we argue that a structured representation of states together with an explicit description of state transitions bring efficiency advantage to the planners.

2.1 Control Structures

It is a known fact that domain-dependent information can significantly improve efficiency of planning [9]. There are two attempts to add domain-dependent information to PDDL models: control rules and hierarchical task networks. Control rules [1] use linear temporal logic to formally describe constraints over a sequence of states visited during execution of the plan. This information can then be used during planning to forbid or to force certain actions in certain states. For example, in the famous Blockworld domain one may express a control rule that a given block can be put on top of an existing tower only if this tower is "good" in the sense that it is not necessary to take off any block from the tower later on. The idea of control rules was implemented in planners such as TLPlan [1] and TALPlanner [15] that proved to be much more efficient than so call domain-independent PDDL planners. The downside of control rules is that they are very complicated to express and the size of models increases significantly in comparison to PDDL models.

Hierarchical task networks (HTN) is another approach to include information about typical sequences of actions to reach a goal. In fact, rather than a goal state, HTN uses the concept of a task to be solved. The task can be solved by decomposing it into subtasks and this process is repeated until primitive tasks – actions – are obtained. There could be alternative ways for task decomposition. The most prominent HTN planner is SHOP2 [19], which also proved to be very efficient for planning in IPC 2002. The downside of HTN is again that the domain modeler needs to put a lot of information about how to actually solve the problem. Briefly speaking, the HTN structure can be seen as a set of recipes on how to reach the goal and the planner selects one of them such that its pieces are compatible.

2.2 Action Languages

Though prevailing in the planning community, PDDL is not the only modeling approach for describing planning problems or more generally to model and reason about dynamic domains. Reasoning about dynamic domains deals with *states* (or situations) represented by sets of values that might change, called *fluents*, and of *actions* that can be applied (action application is an *event*) if some properties (preconditions) hold and that modify, directly or indirectly, the values of fluents (effects). In this context three *calculi* have been devised, according to the focus on one of the entities reported above: the *situation calculus* [17], the *event calculus* [14], and finally the *fluent calculus* [22]. Based on them a plethora of modeling languages have been proposed. Gelfond and Lifschitz [8] surveyed the various proposals of languages used since the sixties for modeling planning problems, called them *action (description) languages*, reorganized them and provide their precise semantics based on Kripke Structures. For some of these languages it has been shown how to encode them systematically in constraint (logic) programming and in Answer Set Programming [5]. However, planners based on action languages suffer even more from efficiency issues and cannot compete with PDDL-based planners.

3 Picat and its Planner

Picat is a logic-based multi-paradigm programming language aimed for general-purpose applications. Picat is a rule-based language, in which predicates, functions, and actors are defined with pattern-matching rules. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, list comprehensions, constraints, and tabling.

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable rule* (also called a *commitment rule*) $Head, Cond$ `=>` $Body$, and the backtrackable rule $Head, Cond$ `?=>` $Body$. In a predicate definition, the $Head$ takes the form $p(t_1, \ldots, t_n)$, where p is called the predicate name, and n is called the arity. The condition $Cond$, which is an optional goal, specifies a condition under which the rule is applicable. For a call C, if C matches $Head$ and $Cond$ succeeds, then the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites C into $Body$. If the used rule is non-backtrackable, then the rewriting is a commitment, and the program can never backtrack to C. However, if the used rule is backtrackable, then the program will backtrack to C once $Body$ fails, meaning that $Body$ will be rewritten back to C, and the next applicable rule will be tried on C.

Briefly speaking, Picat programming is very similar to Prolog programming. By providing features like functions, list comprehensions etc., Picat programs are even more compact and declarative than equivalent Prolog programs. Moreover, the possibility of explicit non-determinism and unification gives the programmer better control of program execution to make the code even more efficient. Last but not least, the built-it tabling mechanism [25] simplifies coding of search algorithms. The power of tabling is in memorizing 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. The tabling mechanism is the driving force behind the Picat planner that will be described next. More details about the Picat language can be found in the Picat documentation [21].

3.1 Planning Domain Model

In general, planning problems can be specified as state transition systems, where actions are used to describe state transitions symbolically. To be more precise, the *planning-domain model* is a compact abstract description of the state transition system usually focusing on modeling state transitions (that implicitly include a model of states). The *planning problem* then consists of specification of an initial state, a description of objects in state representations, a set of actions for transforming states, and a goal condition characterizing the goal state.

Picat's `planner` module was designed to directly support this state-transition approach to domain modeling. The domain modeler firsts needs to decide the abstract model of states. As Picat is a logic programming system, a state can be any term. We will discuss state representations later in the text. Then the domain modeler needs to specify the state transitions – actions – and also the goal condition in the domain model.

The goal condition and state transitions are described as specific Picat rules. To specify the goal condition the domain modeler shall define the following Picat rule:

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

where `goal_condition` can be any Picat goal that checks whether `State` is a goal state.

To specify an (abstract) state transition the domain modeler shall define the following Picat rule:

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

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 should 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`. Optionally, the modeler can add calls to verify that the `NextState` is not a dead-end and how far it is to reach the goal state (heuristic). The heuristic is basically a function computing (a lower-bound for) the distance from the `NextState` to a goal state as in classical A* search. This function can be domain dependent or independent; the choice is up to the domain modeler. In the action rule, the heuristic is applied in the following call:

```
    heuristic(NextState) < current_resource(),
```

where the `current_resource()` is a built-in function of the planner giving the maximal allowed cost-distance to the goal. In the next section we will describe how this information is used during planning.

Note finally, 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 should be applied to a given state and if its application does not lead to a (best) plan then the other actions are not tried. This is done by using deterministic rules (`=>`).

In summary, the planning domain model consists of the description of state transitions – actions – and the specification of a goal condition. Picat rules are used for this description; they only have specific predicates in their heads. This model is separated from the planning algorithm (computational mechanism) that will be described next. It means that the domain models are problem independent and can be applied to any problem in the domain. Hence the Picat approach to planning is similar to other automated planning systems consisting of the planner that takes as input a planning-domain model and a planning-problem description.

3.2 Planning Approaches

The planning-domain model is specified as a set of Picat rules that are explored by the Picat planner. This planner uses basically two search approaches to find optimal plans. Both of them are based on depth-first search with tabling [25] and in some sense they correspond to classical forward planning. It means that they start in the initial state, select an action rule that is applicable to the current state, apply the rule to generate the next state, and continue until they find a state satisfying the goal condition (or the resource limit is exceeded).

The first approach starts with finding any plan first (in the depth-first-search manner). 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* [16]. Note that tabling is used there – the underlying solver remembers the best plans found for all visited states so when visiting the state next time, the plan from it can be reused rather than looked for again. More precisely, a state is expanded only if it is new and its resource limit is non-negative, or if the state has previously failed but the current occurrence has a higher resource limit than before. This planning algorithm is evoked using the following call:

```
 best_plan_upward(+InitState,+CostLimit,-Plan,-PlanCost)
```

This is where the user specifies the initial state and (optionally) the initial cost limit. The algorithm returns a cost-optimal plan and its cost. This approach can be also used to find the first plan using the call `plan(+S,+L,-P,-C)`.

Despite using tabling that prevents re-opening the same state, this approach still requires good control knowledge to find the initial plan (otherwise, it may be lost in a huge state space) or alternatively some good initial cost limit should be used to prevent exploring long plans.

The second approach exploits the idea of iteratively extending the plan length as proposed first for SAT-based planners [12]. It 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* [13], which starts a new round from scratch, Picat also reuses the states that were tabled in the previous rounds.

```
  best_plan(+InitState,+CostLimit,-Plan,-PlanCost)
```

This approach is more robust to weak control knowledge, but it has the disadvantage that it can only find the optimal plan, which could be more time consuming than finding any plan.

Note that the cost limit in the above calls is used to define the function `current_resource()` mentioned in the action rules. Briefly speaking the cost of the partial plan is subtracted from the cost limit to get the value of `current_resource()` that can be utilized to compare with the heuristic distance to the goal.

4 Modelling Principles

In the previous section, we described the Picat planning domain model in an abstract way. We will now discuss some typical modeling approaches used in this abstract state-transition model. The motivation is to provide some general guidelines for domain modeling in Picat based on experience with modeling specific planning domains. We are not aware of any such guidelines for other planning modeling formalism such as PDDL, though some ideas discussed further are known in the planning community.

4.1 Structured State Representation

Before describing state representations in Picat, let us look at the current widespread representation of states in planning. Since the Shakey project and its STRIPS planner [6] a logical model of states is used. In this model, a state is represented as a set (a conjunction) of atoms that are true in that state. For instance, this is a model of a state in the world with one crate and one truck moving between two locations:

$$\{at(truck, loc_1), at(crate, loc_2), connected(loc_1, loc_2)\}.$$

This approach was adopted in a so-called classical planning representation and it is used by the PDDL language [18]. Briefly speaking, each atom represents some primitive property of the world that can be true or not. Some atoms are not changing their truthfulness, they are called *rigid*, while other atoms – *fluents* – hold or do not hold in different states. If P is the set of of all atoms then $S \subseteq P$ is a model of a state describing which atoms hold at that state. Obviously, not every subset of P is a valid state, for example $\{at(truck, loc1), at(truck, loc2)\}$ is not a valid state as the truck must be at exactly one location. Such mutex relations between sets of atoms can be represented using *multi-valued state variables* [2], for example $loc(truck) = loc1$, that naturally models that exactly one atom from a given set of pairwise mutex atoms holds in any state. Then the state is represented as an assignment of values to these state variables (plus rigid atoms). Most current planners use this state-variable representation, but this is still a form of a so called *factored representation* of states. In Picat one can naturally represent sets of atoms as an ordered list of atoms (or similarly the assignment of values to state variables):

```
  [at(crate,loc2), at(truck,loc1), connected(loc1,loc2)].
```

Obviously, the ordering is used here to have a unique representation of a single set. Consequently, the PDDL model of states can be directly (automatically) translated to a Picat model. However, this is usually not the best way as Picat offers a richer representation of states that is known as a *structured representation.*

Representation of states is critical in the Picat domain model. As the Picat planner is based on tabling and states are memorized, two aspects of the state representation should be considered.

First, the state representation should be small so it consumes less memory (all of the visited states are stored in memory). Recall that Picat can use any term to represent a state. As known from implementations of logic programming systems, if two or more terms share a common subterm (for example $f(g(a), b)$ and $h(c, g(a))$

share a subterm $g(a)$) it is possible to store this common subterm just once, which not only decreases memory consumption but also makes unification tests faster. The Picat tabling mechanism can exploit this property via the hash-consing technique [25].

Second, two identical (or more general, two equivalent) states should be represented in the same way so the tabling mechanism can recognize them as such. For example, assume a world where identical trucks are moving between locations. One can identify each truck by its name (as done in PDDL) but if we swap two trucks then we get different states, though from the planning perspective, these states are equivalent – if there is a plan from one state then there is a plan from the other state. A more convenient model is identifying the trucks by their locations.

As we mentioned above, the world state in Picat is represented as a term. To give an example of such representation, let us consider a problem, where a truck is moving between different locations with the goal to transport cargo items from their current locations to some goal locations. This is known as a *Nomystery* domain and it was used for example in the International Planning Competition in 2011 and in the ASP Competition 2013 (for simplicity we ignore fuel consumption in our example). For this domain, we need to represent the location of truck and locations of cargo items. In PDDL (and also for ASP planners), the truck and cargo items have names and their location is represented using atoms $at(what, where)$. As we discussed above, this model contains redundant information that is not necessary for planning. To identify the truck, its location is enough. Similarly each cargo item can be fully identified by its current location and its goal location. When the cargo item is loaded on the truck, then the goal location is enough to identify the cargo item. Then the state can be modeled as a term:

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

where `TruckLoad` is an ordered list of constants describing goal locations of cargo items loaded on the truck and `Cargo` is an ordered list of pairs `(From,To)` modeling cargo items via their current and goal locations. For example, the following state (taken from the ASP competition model):

$$\{at(t_0, a), at(p_0, a), at(p_1, b), goal(p_0, b), goal(p_1, a)\}$$

is represented as follows

```
s(a, [], [(a,b),(b,a)])
```

Note that the movement of each cargo item can be reconstructed easily from the plan so the names are not really necessary there. For example, if the cargo item p_0 is loaded on the truck then we get a state:

```
s(a, [b], [(b,a)])
```

In particular, it is no longer necessary to represent the cargo location explicitly as it is identical to the truck location. Moreover, the state model also contains information that the cargo item is loaded on the truck. If more cargo items with identical goal locations are loaded on the truck then these cargo items become indistinguishable. It means that they are handled identically.

If there are more trucks then the state model can be modified to:

```
s(Trucks,Cargo)
```

where `Trucks` is an ordered list of elements `t(TruckLoc,TruckLoad)`. Still, the truck is enough to be identified by its location and its loaded cargo.

In summary, the state representation must contain state features that are changing, such as location of objects. If there are several objects with identical properties then their names are usually not necessary for planning. By ignoring the names of objects, these objects can be handled interchangeably during planning if their other properties (such as location) are identical. A collection of such objects can be represented as an ordered list of object models, where the object model describes changing properties of the object.

4.2 Control Knowledge

The idea of control knowledge is to suggest the planner which state transitions are good and which are bad. For example, in the above Nomystery problem, simple control knowledge is that the loaded cargo item is unloaded only when the truck is located in the cargo destination. Another control knowledge can be that if the truck is at location where some cargo item is located (and should be transported elsewhere) then this cargo item should be loaded on the truck. PDDL domain models focus on the description of “physics only” and do not contain any control knowledge. Control rules and hierarchical task networks may be used to restrict allowed sequences of actions (states), but they are quite complicated for modeling.

In the Picat domain model, there is a simple mechanism to influence action sequencing by adding control knowledge in the action description. In fact there are three basic ways of expressing control knowledge in Picat domain models. The simplest control knowledge is expressed by making the action description deterministic. It means that the planner will apply only that single action, if it is applicable to the current state, ignoring alternative actions. For example, to express that the cargo item should be unloaded at its destination, we can use the following action description:[1]

```
action(s(TruckLoc,TruckLoad,Cargo),
       NextState, Action, Cost),
    select(TruckLoc,TruckLoad,TruckLoadRest)
=>
    NextState = $s(TruckLoc,TruckLoadRest,Cargo),
    Action = $unload(TruckLoc),
    Cost = 1.
```

We use the Picat built-in call `select/3` to select an element from a list and to return the rest of the list. In this specific example, we verify that some cargo item with the destination equal to truck location is loaded on the truck. Notice that another “hidden” control knowledge is that cargo items delivered to their destinations can be ignored from the rest of planning, which is represented by not including them in the following states. One may notice that the action *unload* does not contain information about which item is being unloaded. As we mentioned in the previous section, the name of the item can be easily reconstructed from the sequence of actions preceding the *unload* action. In particular, there must be some action for loading the item whose destination is the current location. Moreover, as the action is deterministic (notice `=>`), all items belonging to the current location will be unloaded before another action will be tried.

Similarly, we can express the above-mentioned control knowledge that any item at the current location of the truck should be loaded on the truck:

```
action(s(TruckLoc,TruckLoad,Cargo),
       NextState, Action, Cost),
    select((TruckLoc,Dest),Cargo,CargoRest)
=>
    NewTruckLoad = insert_ordered(TruckLoad,Dest),
    NextState = $s(TruckLoc,NewTruckLoad,CargoRest),
    Action = $load(TruckLoc),
    Cost = 1.
```

Now we use the Picat built-in function `insert_ordered/3` to insert an item to a list so that the obtained list is still ordered (see the discussion on state representation). Notice also that by adding the cargo item among loaded items, we use just the destination location to identify that cargo item. This is fine because all cargo items for the same destination are indistinguishable since all of them will be unloaded at the destination.

Assume now that we are in the state where some cargo items should be unloaded and some other cargo items should be loaded. We have two deterministic actions so their order in the domain model will differentiate the order of these actions in the plan. For example, if the rule for *unload* is before the rule for *load* then in any generated plan all items (for that destination) will be unloaded first. Then the condition of the action rule for unloading will not be satisfied so the planner will explore the next applicable action rule that will add (one by one) all loading actions to the plan. This way we restrict the possible plans to those where unloading always precedes loading in a given location. This is actually a desirable behavior as it is a form of symmetry breaking [7] (discussed in the next section).

Finally, we can include the control knowledge within the condition of the rule (but going beyond the classical precondition from PDDL). For simplicity, assume that all locations are interconnected in the above transportation problem. Hence the truck can go to any location (different from the current one). However, it is useless to go just somewhere, but there should be a reason to go there. The reason could be that some cargo item is waiting at that location for delivery or that some loaded cargo item should be delivered there. This can be expressed as an extra condition in the action rule:

[1]Action rules correspond to operations in PDDL and they can contain variables – starting with the capital letter. As Picat supports functions, to differentiate the function call from a term, the prefix `$` is used in front of the term.

```
action(s(TruckLoc,TruckLoad,Cargo),
       NextState, Action, Cost),
     ( member(NextLoc,TruckLoad) ;
       member((NextLoc,_),Cargo)  )
?=>
    NextState = $s(NextLoc,TruckLoad,Cargo),
    Action = $move(TruckLoc,NextLoc),
    Cost = 1.
```

We use the Picat built-in call `member/2` to check membership of element in the list and semicolon to express the disjunctive condition (as in Prolog). Note that behind this control knowledge we use information about the goal in this planning domain, which is delivering cargo items. This particular control knowledge could be expressed in PDDL too but as the condition can be any call in Picat, our model can contain more complex conditions. For example, in the famous Blockworld domain, where the task is to build towers from given blocks, there is a known control rule saying that a block can be placed only to a so called good tower (the tower not violating the goal condition). The procedure for verifying whether a given tower is good can be defined in the model, and then the action rule can use this call:

```
action(s(A,Towers),
       NextState, Action, Cost),
    A != nil,
    select([Top|T],Towers,TowersRest),
    good_tower([A,Top|T])
=>
    NewTowers = insert_ordered(TowersRest,[A,Top|T]),
    NextState = $s(nil,NewTowers),
    Action = $stack(A,Top),
    Cost = 1.
```

In the above action rule we expect a state representing the block being hold by a crane (or `nil` if the crane is empty) and a list of towers, where each tower is an ordered list of its blocks (top-down). This example demonstrates that any external procedure can be used as a condition for action applicability.

In summary, the Picat domain model allows expressing control knowledge within the action rules by using

- deterministic rules and specific order of action rules to force specific actions and action sequences starting at a given state,
- stronger conditions for action applicability ensuring that useless actions are ignored.

4.3 Symmetry Breaking

One of the known issues in planning and other search algorithms is presence of symmetries [7]. The problem with symmetries is that during search "symmetrical" branches are re-explored even if the solution of them is known. For example, if we know that some search branch leads to a failure then any symmetrical search branch will also lead to a failure and it is a waste of time to explore such a branch. In the previous text we have already mentioned two types of symmetries appearing in planning.

The first type of symmetry is *object symmetry*, where different objects are treated as unique entities even if they are fully interchangeable. For example, two cargo items at the same location with identical destinations are fully symmetrical. We have proposed to remove this symmetry via a compact state representation, where these objects are represented identically (still both of them need to be included in the model). It means that name (id) of the object is not included in the representation, only the properties of the object that can change with time, for example location of cargo item, are used to identify the object. Consequently, actions manipulating these objects are identical and hence also fully interchangeable. Note that tabling plays an important role there. For example, if we unload one cargo item at a given location then unloading another cargo item at the same location is not an alternative action (during search) as it would lead to the same state (the call `select/3` is backtrackable so it selects all items from the list that correspond to a given pattern, for example `select(a,[a,a,b,c],Rest)` succeeds two times with `Rest=[a,b,c]`). So if the planner generates three unload actions at the same location then because ignoring the cargo ids in the model this sub-plan basically represents all (six) possible sequences of unload actions for these cargo items. Note finally, that the symmetrical objects must really be interchangeable. For example, if the cargo items have different weights and the truck has limited capacity, then we cannot ignore the cargo weight in the model. Still, some symmetry can be broken for example by forcing a specific order of loading (and unloading) actions by always loading (unloading) the items in the increasing weight. This approach was used in the Petrobras domain model [3].

The second type of symmetry is *action symmetry*, whose specific form we already mentioned in the previous paragraph. In general, action symmetry means that the order of actions can be changed without influencing the final state. For example, two independent actions can be used in any order while reaching the same state after applying both of them. In classical planning techniques these symmetries can be broken by allowing actions to run in parallel (the Graphplan algorithm [4] was probably the first one to use this concept) or by using so called partial order planning, where the plan is a partial order of actions rather than a linear order. The Picat planner is a sequential planner so we cannot exploit these techniques. Nevertheless, the Picat domain model allows other forms to break these symmetrical orders of actions. We already mentioned the possibility to force a given sequence of actions by combining deterministic action rules and their specific order (unloading before loading at a given location). Assume now a more general situation with more trucks, cranes used to load and unload cargo to trucks, and drivers that can walk between locations and that must be in the truck to drive trucks between locations (this is known as the Driverlog domain in International Planning Competitions). In this domain, we have load and unload actions, pickup and drop-off actions, walk and drive actions, and board and disembark actions. There are many symmetries between these actions, for example the crane can pickup cargo, then a driver boards a truck, drives it to the location with the crane, and finally the crane loads cargo to the truck. Obviously, the pickup action can be used anywhere in this sequence before loading. The situation is even more complicated as there might be another truck at another location and the plan for this truck can interleave with the previous plan, meaning that operations for this second truck can be placed inside the sequence of actions for the first truck. A similar behavior was observed in the Petrobras domain [3], where a model using a finite state automaton was suggested to solve these symmetries. The idea is that the modeler defines a continuous sequence of allowed actions that must be performed before trying another sequence of actions. For example, we may require the sequence to be as follows: the driver walks to the location of a truck, then boards the truck and optionally drives the truck to some location. After that the crane at that location (optionally) unloads the truck by performing a sequence of unload and drop-off actions. Then the crane loads the truck by performing actions pick-up and load. After that, the truck can drive somewhere else and the process with unloading and loading repeats. Finally, the driver debarks and walks somewhere. Obviously by forcing specific subsequences of actions the domain modeler restricts the possible plans so it must be done carefully such that at least one valid plan is still allowed. This concept is similar to modeling the problems as hierarchical task networks [19], where tasks are decomposed into valid sequences of subtasks until primitive tasks – actions – are obtained (we sketched above just one such decomposition).

Thanks to the flexibility of the state representation in the Picat domain model, the finite state automaton restricting sequences of actions can naturally be encoded in states and action rules. For demonstration purposes, assume a simplified version of the Driverlog domain where drivers can walk between locations, can board a truck, drive it, and finally debark. For simplicity, no goals are assumed. We only want to restrict that when one driver is acting no actions of another driver interleaves. The allowed sequence of actions can be encoded using a finite state automaton as in Figure 1.

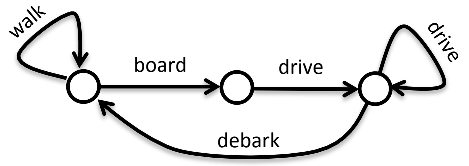

Figure 1: Finite state automaton describing allowed/required sequences of actions per driver.

The regular state of the world can be modeled as a term `s(Drivers,Trucks)`, where `Drivers` is an ordered list of locations of drivers and `Trucks` is an ordered list of truck locations. We will use intermediate states describing that some driver is walking, boarding a truck, or driving. The following action rules describe the FSA from Figure 1.

```
action(s(Drivers,Trucks), NextState, Action, Cost),
    select(DLoc,Drivers,DriversRest), % driver at Dloc
    link(DLoc,DLoc1) % can walk from DLoc to DLoc1
?=>
    NextState = $walking(DLoc1,s(DriversRest,Trucks)),
    Action = $walk(DLoc,DLoc1),
    Cost = 1.

action( s(Drivers,Trucks),
       NextState, Action, Cost),
    select(Loc,Drivers,DriversRest), % driver at Loc
    select(Loc,Trucks,TrucksRest)  % truck at Loc
?=>
    NextState = $boarded(Loc,s(DriversRest,TrucksRest)),
    Action = $board(Loc),
    Cost = 1.

action(walking(DLoc,S),
       NextState, Action, Cost),
    link(DLoc,DLoc1)  % can walk from DLoc to DLoc1
?=>
    NextState = $walking(DLoc1,S),
    Action = $walk(DLoc,DLoc1),
    Cost = 1.

action(walking(DLoc,s(Drivers,Trucks)),
       NextState, Action, Cost),
    select(DLoc,Trucks,TrucksRest)  % truck at DLoc
?=>
    NextState = $boarded(DLoc,s(Drivers,TrucksRest)),
    Action = $board(DLoc),
    Cost = 1.

action(boarded(Loc,S),
       NextState, Action, Cost),
    connect(Loc,Loc1)  % can drive from Loc to Loc1
?=>
    NextState = $driving(Loc1,S),
    Action = $drive(Loc,Loc1),
    Cost = 1.

action(driving(Loc,S),
       NextState, Action, Cost),
    connect(Loc,Loc1) % can drive from Loc to Loc1
?=>
    NextState = $driving(Loc1,S),
    Action = $drive(Loc,Loc1),
    Cost = 1.

action(driving(Loc,s(Drivers,Trucks)),
       NextState, Action, Cost)
?=>
    NewDrivers = insert_ordered(Drivers,Loc),
    NewTrucks = insert_ordered(Trucks,Loc),
    NextState = $s(NewDrivers,NewTrucks),
    Action = $debark(Loc),
    Cost = 1.
```

The above action model contains several action rules for the same action. We can join these rules into one rule by joining their conditions using a disjunction. Also, the order of actions can be changed, for example to prefer boarding before walking. Note also that the model with an FSA naturally describes some preconditions, for example that driving requires a driver aboard.

5 Cave Diving Domain Model

In the previous section we described basic principles of domain modeling with some particular examples. To give a more global view we shall now present a complete domain model. In the last International Planning Competition in 2014 [11] several new planning domains were introduced. To demonstrate a complete domain model we selected the Cave Diving domain, because this domain intrinsically requires planning capabilities. Moreover, based on the results of IPC 2014 this domain is very hard for current state-of-the-art planners (no competing planner solved more than 8 problems out of 20). Last but not least, a frequent critique of modeling approaches supporting domain dependent information is that if the modeler does not know how to solve the problem then neither the automated planner dependent on control knowledge can solve the problem. In this domain model we shall demonstrate that it is enough to include some common-sense information and still significantly beat the best domain-independent planners.

Let us first describe the planning domain (taken from the competition web site [11]). There is a set of divers, each can carry a given number of tanks of air. These divers must be hired to go into an underwater cave and either take photos or prepare the way for other divers by dropping full tanks of air. The cave is too narrow for more than one diver to enter at a time. The cave system is represented by an undirected acyclic graph. Divers have a single point of entry. Certain leaf nodes of the cave branches are objectives that the divers must photograph. Swimming and photographing both consume air tanks. Divers must exit the cave and decompress at the end. They can therefore only make a single trip into the cave. Certain divers have no confidence in other divers and will refuse to work if someone they have no confidence in has already worked. Divers have hiring costs inversely proportional to how hard they are to work with.

The goal is specified by a set of locations where pictures must be taken and a set of divers that must be decompressed at the end. In the model we use predicates defined in the PDDL problem data. In particular, PDDL does not support numbers[2] – predicates `next/2` are used for counting the tanks and predicate `zero_quantity/1` defines the number zero. Similarly, the available tanks are ordered using the predicate `next_tank/2` in PDDL data so there is no symmetry between the tanks. The other predicates define the cave system (`cave_entrance/1`, `connected/2`), the costs of actions (`other_cost/1`, `hiring_cost/2`), and properties of divers (`decompressing/1`, `precludes/2`).

As we mentioned above the most critical modeling decision is the representation of states. A simple hint is that the state must represent every property of the world that can be changed by actions. In the Cave Diving domain the following properties can change:

- the set of available divers (when a diver is hired, he is no longer available),
- the set of locations where pictures need to be taken (after taking the picture it is no more required to take another picture at that location),
- the set of available tanks (actually, knowing the first available tank is enough as the tanks are ordered via `next_tank/2`),
- the location and the number of full tanks placed in the cave system,
- the location of the diver in the cave system (if any diver is in water),
- the numbers of full and empty tanks carried by the diver (if the diver is in water).

Taking in account the above specification, we can define a reasonable part of the plan as follows. First, a diver is hired, then air tanks are prepared for that diver, after that the diver enters the cave and does in some order actions of swimming, taking pictures, picking up or dropping off the air tanks, and finally, the diver decompresses. After that another diver can do a similar sequence of actions which is repeated until all pictures are taken and all divers, that must be decompressed, are indeed decompressed. As we described above, to force a given sequence of actions, the modeler can define a finite state automaton that is then encoded in states. For the Cave Diving domain we can use the automaton depicted in Figure 2.

Note also, that we need to know some identification of divers to ensure that divers precluded by the current diver are excluded from further consideration and also to know the capacity of the diver regarding the number of air tanks he can carry. Nevertheless, as soon as the diver is hired his name is no more important, only his location and the composition of tanks he carries need to be represented in the state. In summary, the main state representation can be:

[2]Picat can use numbers, but to make the model as close as possible to the PDDL model we use the logical encoding of numbers from PDDL.

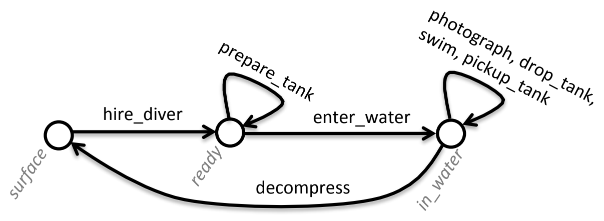

Figure 2: Finite state automaton for the Cave Diving domain

```
  surface(Divers,T,Cave,Pictures)
```

where `Divers` is an ordered list of divers' names, `Pictures` is an ordered list of locations, where some pictures should be taken, `T` is a name of the first available tank, and finally `Cave` is an ordered list with elements `(Loc, Num)`, where `Loc` is a location in cave and `Num` is a positive number indicating the number of full air tanks at that location. In the initial state, the list `Cave` is empty. In the final state, the list `Pictures` should be empty (all pictures are taken) and there is no available diver that should be decompressed. The following rule defines the final state:

```
final(surface(Divers,_,[],[])),
   not (member(D,Divers), decompressing(D))
=> true.
```

After hiring any diver, we change the state to the following one:

```
  ready(Cap,FullTanks,EmptyTanks,Divers,T,Cave,Pictures)
```

where we represent the free capacity of that diver and how many empty and full tanks of air he currently carries (initially zero). The following action rule does the transition:

```
action(surface(Divers,T,Cave,Pictures),
       NextState,Action,ActionCost),
   select(D,Divers,RDivers), % select next diver
   not  (precludes(D,DD),
         decompressing(DD), member(DD,RDivers))
=>
   Action = $hire_diver(D),
   hiring_cost(D, ActionCost),
   NextDivers = [D2 : D2 in RDivers, not precludes(D,D2)],
   capacity(D,Cap), zero_quantity(Zero),
   NextState =
       $ready(Cap,Zero,Zero,NextDivers,T,Cave,Pictures).
```

In this rule we use simple control knowledge forbidding hiring a diver who precludes another available diver that must decompress. Obviously, if such a diver is selected than no plan can reach the goal state as the precluded divers are removed from further consideration and hence cannot decompress.

In the state `ready` some number of tanks can be added to the diver while respecting his capacity limits.

```
action(ready(Cap,Loaded,Empty,Divers,T,Cave,Pictures),
       NextState,Action,ActionCost),
   next_quantity(RCap,Cap),
   next_quantity(Loaded,NLoaded),
   next_tank(T,NT)
?=>
   Action = $prepare_tank(T),
   other_cost(ActionCost),
   NextState =
      $ready(RCap,NLoaded,Empty,Divers,NT,Cave,Pictures).
```

After the tanks are added to the diver, the diver enters in water, which means another change of state. Notice that the action rule for preparing tanks precedes the rule for entering in water which means that during planning the actions for preparing tanks are tried before the action for entering in water. The following rule is used for this second transition

```
action(ready(Cap,Loaded,Empty,Divers,T,Cave,Pict),
       NextState,Action,ActionCost),
   cave_entrance(Loc)
=>
   Action = $enter_water,
   other_cost(ActionCost),
   NextState =
     $in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pict).
```

The rule again changes the state according to the FSA from Figure 2. The meaning of attributes is identical to the state `ready`:

```
 in_water(Cap,FullTanks,EmptyTanks,Divers,T,Cave,Pictures)
```

When the diver is in water, he can take pictures, if he is at proper location, he can drop off some full tank, to prepare it for future dives, he can drop off an empty tank, if he needs to pick up a full tank, and finally he can swim between locations. In the domain model, we just describe these actions with some simple control knowledge such as no dropping of full tank at cave entrance (it is not necessary there) and dropping off an empty tank only when some full tank can be picked up.

```
action(
   in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pictures),
        NextState,Action,ActionCost),
   select(Loc,Pictures,RPict),
   next_quantity(RLoaded,Loaded),
   next_quantity(Empty,NEmpty)
?=>
   Action = $photograph(Loc),
   other_cost(ActionCost),
   NextState =
     $in_water(Loc,Cap,RLoaded,NEmpty,Divers,T,Cave,RPict).

action(in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pict),
       NextState,Action,ActionCost),
   not cave_entrance(Loc),  % do not drop tanks
   next_quantity(Cap,NCap), % at cave entrance
   next_quantity(RLoaded,Loaded)
?=>
   Action = $drop_tank(Loc),  % drop full tank
   other_cost(ActionCost),
   (select((Loc,Tanks),Cave,RCave) ->
      NTanks = Tanks+1
   ;      % some tanks are at location Loc or
      RCave = Cave,
      NTanks = 1
   ),
   NCave = insert_ordered(RCave,(Loc,NTanks)),
   NextState =
     $in_water(Loc,NCap,RLoaded,Empty,Divers,T,NCave,Pict).

action(in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pict),
       NextState,Action,ActionCost),
   connected(Loc,NLoc),
   next_quantity(RLoaded,Loaded),
   next_quantity(Empty,NEmpty)
?=>
   Action = $swim(Loc,NLoc),
   other_cost(ActionCost),
   NextState =
      $in_water(NLoc,Cap,RLoaded,NEmpty,Divers,T,Cave,Pict).

action(in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pict),
       NextState,Action,ActionCost),
   select((Loc,Tanks),Cave,RCave),
   RTanks = Tanks-1,
   next_quantity(RCap,Cap),
   next_quantity(Loaded,NLoaded)
?=>
   Action = $pickup_tank(Loc),
   other_cost(ActionCost),
   (RTanks=0 ->
      NCave = RCave
    ;
      NCave = insert_ordered(RCave,(Loc,RTanks))
   ),
   NextState =
    $in_water(Loc,RCap,NLoaded,Empty,Divers,T,NCave,Pict).

action(in_water(Loc,Cap,Loaded,Empty,Divers,T,Cave,Pict),
       NextState,Action,ActionCost),
   zero_quantity(Cap),     % when missing capacity and
   member((Loc,_Tanks),Cave),    % full tank available
   next_quantity(Cap,NCap),
   next_quantity(REmpty,Empty)
?=>
   Action = $drop_tank(Loc),   % drop empty tank
   other_cost(ActionCost),
   NextState =
    $in_water(Loc,NCap,Loaded,REmpty,Divers,T,Cave,Pict).
```

The final action rule is for decompressing the diver. In this case the diver is no longer retained (recall, that each diver can dive at most once) and we switch back to the state `ready`. We only use simple control knowledge saying that the diver should not return with any full tank. This is just to generate good plans – if the diver returns with a full tank then this tank was not necessary and hence should never be picked up.

```
action(
   in_water(Loc,_Cap,Loaded,_Empty,Divers,T,Cave,Pictures),
       NextState,Action,ActionCost),
   zero_quantity(Loaded), % the diver should return
   cave_entrance(Loc)     % with no full tank
=>
   Action = $decompress,
   other_cost(ActionCost),
   NextState = $surface(Divers,T,Cave,Pictures).
```

For the sake of completeness, we enclose a full description of the Cave Diving domain in PDDL in Appendix A. As the reader can verify, we use exactly the same actions. The Picat domain model can exploit ad-hoc structured representation of states and some control knowledge. We did not use any heuristics in the model; only common-sense control knowledge is included. With this model, we can solve optimally all instances from IPC 2014 as shown in the next section.

6 Experimental Evaluation

To evaluate the efficiency of the proposed modeling framework we compared it experimentally with the state-of-the-art planning systems using the benchmark domains from International Planning Competitions [11]. We did the comparison in two different settings. First, we compared with planners that use some form of control knowledge, namely TLPlan [1] and TALPlanner [15] exploiting control rules and SHOP2 [19] based on hierarchical task networks. TLPlan and SHOP2 were winners of IPC 2002. Second, we compared with the state-of-art optimal PDDL planner SymbA, the winner of IPC 2014 deterministic optimal track.

For the comparison, we developed an encoding in Picat for each domain; some examples were given earlier in the text. The particular planning problem instances were translated automatically from their PDDL encodings.

6.1 Planners with Domain Knowledge

The expressive power of the proposed modeling framework is similar to exploiting control rules and hierarchical structures of tasks. Hence, it is natural to compare the framework to planners using these structures. We compared with TLPlan [1], the best hand-coded planner of IPC 2002, TALPlanner [15] another good planner based on control rules, and SHOP2 [19], the distinguished hand-coded planner of IPC 2002. Obviously, efficiency of planning depends on the quality of encoding so we took the existing domain encodings rather than encoding the domains for the other planners ourselves. In particular, we took the following domains: Depots, Zenotravel, Driverlog, Satellite, and Rovers from International Planning Competition 2002. The Picat encodings are available at: `picat-lang.org/aips02/`. The reason for taking benchmark models from a quite old competition is that this was the last time when these planners were allowed to compete so we can use the competition domains and results for direct comparison.

All planners found plans for all benchmark problems and the runtime to generate plans is negligible; every planner found a plan in a matter of milliseconds. Hence we focus on comparing the quality of obtained plans. We measure the quality of a plan as the sum of costs of all actions in the plan. Frequently, the cost of action is one so the quality of a plan corresponds to its length. Obviously, the smaller cost means a better plan. In recent IPCs, a so called quality score was introduced to compare quality of planners over a set of problems. For each problem, the quality number between 0 and 1 is assigned to a plan as follows. If the planner cannot find a plan then 0 is used as the plan quality. Let BC_p be the cost of the best plan obtained by any planner for problem p and let C_p be the cost of some plan for problem p. Then the score for this plan is $\frac{BC_p}{C_p}$. Hence, if the score of the plan is 1 then the plan is the best plan found. Otherwise, the score is smaller than 1. The quality score for a set of problems P is then $\sum_{p\in P}\frac{BC_p}{C_p}$. The higher quality score means an overall better planning system.

For TLPlan, TALPlanner, and SHOP2 we took the best plans reported in the results of IPC 2002. We first compare them to the first plans obtained by the Picat planner (via `plan/4`); the quality scores are reported in Table 1.

Table 1: Comparison of quality scores for the first plan found

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	20.87	20.60	**21.23**	19.27
Zenotravel	20	16.30	19.15	**19.55**	17.84
Driverlog	20	10.39	19.08	**19.24**	15.37
Satellite	20	**19.73**	19.64	17.77	18.39
Rovers	20	17.79	**19.04**	15.63	18.88
Total	102	85.08	**97.52**	93.41	89.75

The results show that the encoding of control rules for the TLPlan is really the best one for finding the first plan, while the Picat model does not excel. Nevertheless, this first experiment tells more about the craft of the domain modeler rather than giving a whole picture about the overall capabilities of the planner. Recall that the Picat planner has the possibility to improve the first plan found by looking for better plans in the branch-and-bound style. In the second experiment, we let the Picat planner improve the plan using the `best_plan_upward/4` planning algorithm. Table 2 shows the quality scores when we gave five minutes to the Picat planner to improve the plan (running under MacOS X 10.10 on 1.7 GHz Intel Core i7 with 8 GB RAM).

Table 2: Comparison of quality scores for the best plan (5 minutes)

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**21.94**	19.93	20.52	18.63
Zenotravel	20	**19.86**	18.40	18.79	17.14
Driverlog	20	17.21	17.68	**17.87**	14.16
Satellite	20	**20.00**	18.33	16.58	17.16
Rovers	20	**20.00**	17.67	14.61	17.57
Total	102	**99.01**	92.00	88.37	84.65

Now the situation is completely different and the Picat planner produces plans of much better quality. In fact, out of 102 problems, the Picat planner found best plans for 91 problems, while TLPlan for 23 problems, TALPlanner for 22 problems, and SHOP2 for 9 problems. Table 3 shows the number of best plans found per domain.

Table 3: The number of best plans found

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**19**	7	3	0
Zenotravel	20	**17**	6	7	3
Driverlog	20	**15**	5	6	1
Satellite	20	**20**	5	3	2
Rovers	20	**20**	0	3	3
Total	102	**91**	23	22	9

According to the initial experiments, the Picat models show promising efficiency both in terms of runtime and plan quality. The open question is how much effort is required to model the domains. This would require a sociological study with a large number of participants so we opted to demonstrate the modeling effort by comparing the sizes of domain models. The domain encodings are publicly available for the TLPlan [23] only so we can compare only with them and with the PDDL models. This was also the reason why we selected the above specific domains for experimental comparison. Table 4 shows the number of lines of domain models for Picat, TLPlan, and PDDL.

Table 4: The number of lines in domain models

Domain	Picat	TLPlan	PDDL
Depots	156	933	42
Zenotravel	109	308	61
Driverlog	190	1395	79
Satellite	132	186	75
Rovers	223	914	119
Total	810	3736	376

We are aware that the above comparison is only informative and not fully conclusive. TLPlan uses Lisp as the underlying language (similarly to PDDL) while Picat is a logic programming language with syntax close to Prolog. Taking into account the syntax differences between Lisp and Prolog, we still believe that the numbers of lines of the models provide an interesting comparison. Moreover, the Picat domain models also include code for transforming the initial states from PDDL to Picat. The comparison shows that Picat models are much closer to PDDL with respect to size than the TLPlan models. This also justifies why we compared the Picat planner with a PDDL-based planner in the next section.

6.2 Domain-Independent Planners

The Picat modeling approach is more expressive than STRIPS models in PDDL. More precisely any PDDL model can be translated to an equivalent Picat planning model. The Picat planning model is also separated from the solving mechanism as we mentioned earlier. However, Picat models are more expressive than PDDL in supporting a structured representation of states and control knowledge in action description. Hence one may tend to say that the comparison of PDDL-based planners to the Picat planner is not fully fair because Picat can exploit more information. Still, it is important to show that this extra information is useful for efficient planning. In particular, we argue that flexible modeling capabilities of Picat are more important for efficiency of planning than advanced domain-independent planning algorithms.

We have encoded in Picat several other domains used in the deterministic sequential track of IPC 2014 [11]. All of the encodings are available at: `picat-lang.org/ipc14/`. We have compared Picat encodings with the PDDL encodings solved with SymbA [Torralba et al., 2014], a domain-independent bidirectional A* planner which won the optimal sequential track of IPC 2014.

Table 5 shows the number of instances (#insts) in the domains used in IPC 2014 and the number of (optimally) solved instances by each planner. The results were obtained on a Cygwin notebook computer with 2.4GHz Intel i5 and 4GB RAM. Both Picat and SymbA were compiled using g++ version 4.8.3. For SymbA,

Table 5: The number of problems solved optimally.

Domain	# insts	Picat	SymbA
Barman	14	**14**	6
Cave Diving	20	**20**	3
Childsnack	20	**20**	3
Citycar	20	**20**	17
Floortile	20	**20**	**20**
GED	20	**20**	19
Parking	20	**11**	1
Tetris	17	**13**	10
Transport	20	**10**	8

a setting suggested by one of SymbA's developers was used. A time limit of 30 minutes was used for each instance as in IPC. For every instance that was solved by both SymbA and Picat, the plan quality is the same.

Picat solved more instances than SymbA for every domain except for *Floortile*, for which both systems solved all of the instances. The running times of the instances are not given, but the total runs for Picat were finished within 24 hours, while the total runs for SymbA took more than 72 hours. This demonstrates that planning models in Picat are much more efficient for planning than the PDDL models solved by winning planning systems. As we demonstrated before, the modeling effort necessary for Picat does not put significant burden in comparison to PDDL.

7 Conclusions

Tabled logic programming has been proved to work extremely well for specific problems such as Sokoban [26] and Petrobras planning problem [3]. In this paper we presented a declarative modeling framework for describing general planning domains. This framework sits on top of the Picat programming language and exploits its planning module based on tabling. The domain modeler only needs to specify actions as state transitions and the underlying planning algorithm automatically looks for plans. It is a similar concept to PDDL models, but the Picat domain models support richer concepts such as structured representations of states and inclusion of control knowledge in action rules to exploit determinism and symmetries. The initial experiments showed that this mechanism is very efficient in comparison with domain-independent planners, and achieves with less efforts comparable efficiency to planners that exploit domain knowledge via control rules and hierarchical task networks.

In the paper we described some modeling concepts based on our experience with modeling planning domains for International Planning Competitions. A deeper study of the influence of these concepts on efficiency of planning is necessary to understand better the modeling principles. A big open question is whether PDDL models can be automatically translated into Picat. A straightforward translation is obviously possible, but it does not lead to efficiently solvable models.

The approach proposed in this paper is somehow orthogonal to current research in automated planning that focuses on improving the domain-independent planning algorithms. We propose to put more effort on domain modeling by providing richer modeling capabilities. Our initial experiments have shown promising results in this direction.

Acknowledgments

Roman Barták was supported by the Czech Science Foundation under the project P103-15-19877S. Agostino Dovier was partially supported by INdAM 2014 and 2015 projects. Neng-Fa Zhou was supported in part by the NSF under grant number CCF1018006.

References

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

[2] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4), 625–655, 1995.

[3] Roman Barták and Neng-Fa Zhou. Using tabled logic programming to solve the Petrobras planning problem. *Theory and Practice of Logic Programming*, 14(4-5):697–710, 2014.

[4] Avrim Blum and Merrick Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90:281–300, 1997.

[5] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Multi-valued Action Languages with Constraints in CLP(FD). *Theory and Practice of Logic Programming*,10(2):167–235, 2010.

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

[7] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In IJCAI, pages 956–961, 1999.

[8] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16),193–210, 1998.

[9] Patrik Haslum and Ulrich Scholz. Domain knowledge in planning: Representation and use. In ICAPS Workshop on PDDL, 2003.

[10] Carl Hewitt. Planner: A language for proving theorems in robots. In IJCAI, pages 295–302, 1969.

[11] International Planning Competitions web site, `http://ipc.icaps-conference.org/`, Accessed April 5, 2015.

[12] Henry Kautz and Bart Selman. Planning as satisfiability. Proceedings of ECAI, 359–363, 1992.

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

[14] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events, *New Generation Computing*, 4, 67–94, 1986.

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

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

[17] John McCarthy. Situations, actions and causal laws. Technical Report Memo 2, Stanford University Artificial Intelligence Laboratory, Stanford, CA, 1963. Reprinted in: Marvin Minsky, editor. Semantic information processing. MIT Press, 1968

[18] Drew McDermott. The planning domain definition language manual. CVC Report 98-003, Yale Computer Science Report 1165, 1998.

[19] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: an HTN planning system. *J. Artificial Intelligence Research* (JAIR), 20:379–404, 2003.

[20] Nils J. Nilsson. Shakey The Robot, Technical Note 323. AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984.

[21] Picat web site, `http://picat-lang.org/`, Accessed April 5, 2015.

[22] Michael Thielscher. Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence*, 2: 179–192, 1998.

[23] TLPlan web site, `http://www.cs.toronto.edu/tlplan/`, Accessed April 5, 2015.

[Torralba et al., 2014] Alvaro Torralba, Vidal Alcazar, and Daniel Borrajo. Symba: A symbolic bidirectional a planner. In The 2014 International Planning Competition, pages 105–109, 2014.

[24] David S. Warren. Memoing for logic programs. *Comm. of the ACM*, Special Section on Logic Programming, 35:93–111, 1992.

[25] Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming*, 12(4-5):547–563, 2012.

[26] Neng-Fa Zhou and Agostino Dovier. A tabled Prolog program for solving Sokoban. *Fundamenta Informaticae*, 124(4):561–575, 2013.

[27] Neng-Fa Zhou. Combinatorial Search With Picat. *http://arxiv.org/abs/1405.2538*, 2014.

A Cave Diving domain in PDDL

```
(define (domain cave-diving-adl)
  (:requirements :typing :action-costs :adl)
  (:types location diver tank quantity)
  (:predicates
    (at-tank ?t - tank ?l - location)
    (in-storage ?t - tank)
    (full ?t - tank)
    (next-tank ?t1 - tank ?t2 - tank)
    (at-diver ?d - diver ?l - location)
    (available ?d - diver)
    (at-surface ?d - diver)
    (decompressing ?d - diver)
    (precludes ?d1 - diver ?d2 - diver)
    (cave-entrance ?l - location)
    (connected ?l1 - location ?l2 - location)
    (next-quantity ?q1 - quantity ?q2 - quantity)
    (holding ?d - diver ?t - tank)
    (capacity ?d - diver ?q - quantity)
    (have-photo ?l - location)
    (in-water )
  )

  (:functions
    (hiring-cost ?d - diver) - number
    (other-cost) - number
    (total-cost) - number
  )

  (:action hire-diver
    :parameters (?d1 - diver)
    :precondition (and  (available ?d1)
                        (not (in-water))
                  )
    :effect (and (at-surface ?d1)
                 (not (available ?d1))
                 (forall (?d2 - diver)
                     (when (precludes ?d1 ?d2)
                                (not (available ?d2))))
                 (in-water)
                 (increase (total-cost) (hiring-cost ?d1))
            )
  )

  (:action prepare-tank
    :parameters (?d - diver ?t1 ?t2 - tank ?q1 ?q2 - quantity)
    :precondition (and (at-surface ?d)
                       (in-storage ?t1)
                       (next-quantity ?q1 ?q2)
                       (capacity ?d ?q2)
                       (next-tank ?t1 ?t2)
                  )
    :effect (and (not (in-storage ?t1))
                 (not (capacity ?d ?q2))
                      (in-storage ?t2)
                      (full ?t1)
                      (capacity ?d ?q1)
                      (holding ?d ?t1)
                 (increase (total-cost) (other-cost ))
            )
  )
```

```
(:action enter-water
  :parameters (?d - diver ?l - location)
  :precondition (and (at-surface ?d)
                     (cave-entrance ?l)
                )
  :effect (and (not (at-surface ?d))
                    (at-diver ?d ?l)
               (increase (total-cost) (other-cost ))
          )
)

(:action pickup-tank
  :parameters (?d - diver ?t - tank ?l - location
               ?q1 ?q2 - quantity)
  :precondition (and (at-diver ?d ?l)
                     (at-tank ?t ?l)
                     (next-quantity ?q1 ?q2)
                     (capacity ?d ?q2)
                )
  :effect (and (not (at-tank ?t ?l))
               (not (capacity ?d ?q2))
                    (holding ?d ?t)
                    (capacity ?d ?q1)
               (increase (total-cost) (other-cost ))
          )
)

(:action drop-tank
  :parameters (?d - diver ?t - tank ?l - location
               ?q1 ?q2 - quantity)
  :precondition (and (at-diver ?d ?l)
                     (holding ?d ?t)
                     (next-quantity ?q1 ?q2)
                     (capacity ?d ?q1)
                )
  :effect (and (not (holding ?d ?t))
               (not (capacity ?d ?q1))
                    (at-tank ?t ?l)
                    (capacity ?d ?q2)
               (increase (total-cost) (other-cost ))
          )
)

(:action swim
  :parameters (?d - diver ?t - tank ?l1 ?l2 - location)
  :precondition (and (at-diver ?d ?l1)
                     (holding ?d ?t)
                     (full ?t)
                     (connected ?l1 ?l2)
                )
  :effect (and (not (at-diver ?d ?l1))
               (not (full ?t))
                    (at-diver ?d ?l2)
               (increase (total-cost) (other-cost ))
          )
)

(:action photograph
  :parameters (?d - diver ?l - location ?t - tank)
  :precondition (and (at-diver ?d ?l)
                     (holding ?d ?t)
                     (full ?t)
                )
    :effect (and (not (full ?t))
                     (have-photo ?l)
                (increase (total-cost) (other-cost ))
           )
  )

  (:action decompress
    :parameters (?d - diver ?l - location)
    :precondition (and (at-diver ?d ?l)
                     (cave-entrance ?l)
                )
    :effect (and (not (at-diver ?d ?l))
                     (decompressing ?d)
                (not (in-water))
                (increase (total-cost) (other-cost ))
           )
  )

)
```