Proceedings, Fifteenth International Conference on
Principles of Knowledge Representation and Reasoning (KR 2016)

Guiding Planning Engines by Transition-Based Domain Control Knowledge

Lukáš Chrpa
PARK Research Group
School of Computing & Engineering
University of Huddersfield

Roman Barták
Constraint & Logic Programming Research Group
Faculty of Mathematics and Physics
Charles University in Prague

Abstract

Domain-independent planning requires only to specify planning problems in a standard language (e.g. PDDL) in order to utilise planning in some application. Despite a huge advancement in domain-independent planning, some relatively-easy problems are still challenging for existing planning engines. Such an issue can be mitigated by specifying Domain Control Knowledge (DCK) that can provide better guidance for planning engines.

In this paper, we introduce *transition-based DCK*, inspired by Finite State Automata, that is efficient as demonstrated empirically, planner-independent (can be encoded within planning problems) and easy to specify.

Introduction

Despite a huge advancement of domain-independent planning engines, they might still struggle with problems that are easy to solve for a domain-dependent approach because “raw” specification of these problem might not be very informative for domain-independent planning engines. This issue can be (to some extent) addressed by specifying Domain Control Knowledge (DCK) that can guide the search that planning engines perform. DCK can be specified, for instance, in form of Control rules (Minton and Carbonell 1987), Hierarchical Task Networks (HTNs) (Georgievski and Aiello 2015), and Macro-operators (Korf 1985). DCK can be exploited by specifically tailored planning engines such as TALPlanner (Kvarnström and Doherty 2000) for Control Rules, and SHOP2 (Nau et al. 2003)) for HTNs. Some kinds of DCK such as macro-operators can also be (automatically) encoded into planning problems and thus any standard planning engine can exploit such DCK.

In this paper, we introduce *transition-based Domain Control Knowledge* that is inspired by Finite State Automata. Roughly speaking, transition-based DCK represents knowledge about dependencies between planning operators that is used to restrict the number of their instances that can be applied in each step of the planning process. We will show that transition-based DCK can be directly encoded into planning problems, so standard planning engines can reason with it. Like Finite State Automata, transition-based DCK can be specified in a schematical way, which we believe is easy to use for domain engineers. To demonstrate the efficiency of transition-based DCK we use six benchmark domains and six state-of-the-art domain-independent planning engines.

Classical Planning

Classical Planning deals with finding a partially or totally ordered sequence of actions transforming the static and fully observable environment from an initial state to a desired goal state (Ghallab, Nau, and Traverso 2004).

In the classical representation, the environment is described by *predicates* that are constructed in form $\mathit{pred\_name}(x_1, \ldots, x_k)$ such that $\mathit{pred\_name}$ is a unique predicate name and $x_1, \ldots x_k$ are predicate arguments, where each argument is either a variable symbol or a constant. *Planning states* are defined as sets of grounded predicates. We say that $o = (\mathit{name}(o), \mathit{pre}^+(o), \mathit{pre}^-(o), \mathit{eff}^-(o), \mathit{eff}^+(o), \mathit{cost}(o))$ is a *planning operator*, where $\mathit{name}(o) = \mathit{op\_name}(x_1, \ldots, x_k)$ ($\mathit{op\_name}$ is a unique operator name and $x_1, \ldots x_k$ are variable symbols, arguments, appearing in the operator) and $\mathit{pre}^+(o), \mathit{pre}^-(o), \mathit{eff}^-(o)$ and $\mathit{eff}^+(o)$ are sets of predicates with variables taken only from $x_1, \ldots x_k$ representing o's positive and negative preconditions, and negative and positive effects, and $\mathit{cost}(o)$ is a numerical value representing o's cost[1]. *Actions* are instances of planning operators. An action a is *applicable* in a planning state s if and only if $\mathit{pre}^+(a) \subseteq s \wedge \mathit{pre}^-(a) \cap s = \emptyset$. Application of a in s (if possible) results in a planning state $(s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)$.

A *planning domain model* is specified by a set of constants (domain-specific objects), a set of predicates and a set of planning operators accommodating these constants and predicates. A *planning problem* is specified via a (planning) domain model, a set of constants (problem-specific objects), an initial planning state, and a set of goal predicates. A goal predicate is an *open goal* if it has not yet been achieved in the current planning state. A *solution plan* of the planning problem is a sequence of actions such that their consecutive application starting from the initial planning state results in a planning state containing all the goal predicates.

[1]Implicitly, $\mathit{cost}(o) = 1$.

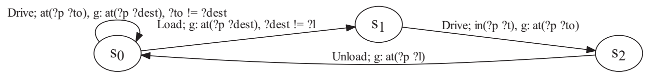
Drive; at(?p ?to), g: at(?p ?dest), ?to != ?dest
Load; g: at(?p ?dest), ?dest != ?l
s1
Drive; in(?p ?t), g: at(?p ?to)
s0
Unload; g: at(?p ?l)
s2

Figure 1: Transition-based DCK for our simple logistic domain.

Running Example

We consider a simple logistic domain, where packages have to be delivered from their initial locations to their goal locations by trucks that can carry at most one package each. No other constraints are defined, i.e., all locations are connected and a package can be carried by any of the trucks. We define three predicates: (at ?x ?l) – an object ?x (either truck or package) is at location ?l; (in ?p ?t) – a package ?p is in the truck ?t; (free ?t) – a truck ?t is empty (no package is in it). Then, we define three planning operators: Drive(?t ?from ?to) – a truck ?t moves from the location ?from to a location ?to; Load(?t ?p ?l) – a package ?p is loaded into the truck ?t at the location ?l; and Unload(?t ?p ?l) – a package ?p is unloaded from the truck ?t at the location ?l.

Transition-based Domain Control Knowledge

Generally speaking, Domain Control Knowledge (DCK) provides a guidance to planning engines and thus makes the planning process more efficient. Inspired by *Finite State Automata*, we designed a *transition-based DCK* that is easy to specify and can be encoded into planning domain models and problems (described later in the text). In principle, our transition-based DCK consists of a set of DCK states and transitions that refer to actions that can be applied under specified conditions in a given planning state. The formal definition of transition-based DCK follows.

Definition 1. *Transition-based DCK is a quadruple $\mathcal{K} = (S, O, T, s_0)$, where S is a set of DCK states, $s_0 \in S$ is the initial DCK state, O is a set of planning operators, and T is a set of transitions in the form (s, o, C, s'), where $s, s' \in S$, $o \in O$ and C is a set of constraints, where each constraint is in the form:*

- *$p, \neg p$ — a predicate p must or must not be present in the current planning state*
- *g:p — a predicate p is an open goal in the current planning state*

In our running example, we assume that the packages are at their initial locations and the trucks are empty. We may observe that i) an empty truck has to be moved only to locations where some package is waiting for being delivered, and ii) if a package is loaded to the truck (in its initial location), then the truck has to move to package's goal location, where the package is then unloaded. Such an observation can be encoded as a transition-based DCK with s_0 as an initial DCK state as depicted in Figure 1.

A (classical) planning problem can be solved by a generic algorithm that starting in the initial planning state, iterates by non-deterministically selecting an action (an instance of a planning operator defined in the domain model of the problem) that is applicable in the current planning state and by

```
(:action Drive-empty
:parameters (?t - truck ?from ?to ?dest - location ?p - package)
:precondition (and (at ?t ?from)(at ?p ?to)(DCK-state s0)
        (open-goal-at ?p ?dest)(not (= ?to ?dest)))
:effect (and (not (at ?t ?from))(at ?t ?to))
)
(:action Drive-full
:parameters (?t - truck ?from ?to - location ?p - package)
:precondition (and (at ?t ?from)(DCK-state s1)
        (in ?p ?t)(open-goal-at ?p ?to))
:effect (and (not (at ?t ?from))(at ?t ?to)
        (not (DCK-state s1))(DCK-state s2))
)
(:action Load
:parameters (?t - truck ?p - package ?l ?dest - location)
:precondition (and (at ?t ?l)(at ?p ?l)(free ?t)(DCK-state s0)
        (open-goal-at ?p ?dest)(not (= ?to ?dest)))
:effect (and (not (at ?p ?l))(not (free ?t))(in ?p ?t)
        (not (DCK-state s0))(DCK-state s1))
)
(:action Unload
:parameters (?t - truck ?p - package ?l - location)
:precondition (and (at ?t ?l)(in ?p ?t)(DCK-state s2)
        (open-goal-at ?p ?l))
:effect (and (not (in ?p ?t))(free ?t)(at ?p ?l)
        (not (open-goal-at ?p ?l))
        (not (DCK-state s2))(DCK-state s0))
)
```

Figure 2: Modified planning operators (in PDDL) of our simple logistics domain with respect to the transition-based DCK as in Figure 1. Additional arguments and predicates introduced by the DCK are in italics.

applying the action (updating the current planning state) until the goal is reached (all the goal predicates are present in the current planning state). Transition-based DCK can be embedded into the generic algorithm as follows. Let s_Π be the current planning state and $s_\mathcal{K}$ be the current DCK state (we start in the initial planning state and in the initial DCK state). An action a can be selected for being applied if and only if a is applicable in s_Π and there is a transition $(s_\mathcal{K}, o, C, s'_\mathcal{K})$ such that a is an instance of o and for each element in C which is in form $p, \neg p$, or g:p it is the case that a corresponding instance of p is in s_Π, not in s_Π, or is an open goal respectively. After a is applied the current planning state is updated accordingly and $s'_\mathcal{K}$ becomes the current DCK state. The goal predicate is considered to be an open goal only before it is achieved for the first time. Reopening open goals requires more complex encoding and, moreover, we believe that it is not very useful to direct the search toward goals that need to be achieved more than once.

We can easily observe that the augmented generic planning algorithm is sound, i.e., a plan that is returned by the algorithm is a solution plan of the problem given on the input. This is because exploiting transition-based DCK only puts further restrictions on action selection. A *well-defined transition-based DCK* does not prune all solution plans of the problem, so it remains solvable. The DCK from our running example is well defined for problems where none of the packages that has to be delivered is initially loaded in any of the trucks and at least one truck is initially empty.

Encoding Transition-based DCK into Planning Problems

In the case of our transition-based DCK, we have to encode DCK states, transitions, and constraints under which the transitions can be performed. The encoding will be described in the following paragraphs.

DCK states can be encoded by adding a supplementary predicate with one argument, e.g., (DCK-state ?s), into a planning domain model. Concrete DCK states are encoded as domain-specific objects (e.g. s0, s1, s2). The initial DCK state is encoded into the initial planning state by a corresponding instance of the supplementary predicate. In our running example, the initial DCK state is s_0, therefore, (DCK-state s0) will be added to the initial planing state of each planning problem in our simple logistics domain.

Open goals can be encoded by adding supplementary "twin" predicates that have the same arguments as the goal predicates. Instances of these supplementary predicates that correspond to the goal predicates are added into the initial planning state. In our running example, if we have a problem with two goal predicates, (at pkg1 loc1) and (at pkg2 loc2), then the supplementary predicates, (open-goal-at pkg1 loc1) and (open-goal-at pkg2 loc2) are added into the initial planning state. Also, planning operators that achieve goal predicates (have them in the positive effects) are extended by adding the corresponding supplementary predicates into their negative effects. In our running example, the Unload operator achieves goal predicates, i.e., (at ?p ?l), so (open-goal-at ?p ?l) is added to its negative effects. Clearly, if no transition in the DCK considers open goals, there is no need to encode them.

Transitions in our transition-based DCK incorporate information about what planning operators and under which constraints they can be applied in given DCK states and how DCK states change. Such information can be encoded within the planning operators defined in the corresponding domain model. For each planning operator o we identify how many transitions (from T) refer to it, i.e., $t(o) = |\{t \mid t \in T, t = (s, o, C, s')\}|$. If for an operator o, $t(o) = 0$, then o can never be applied and thus will be removed from the domain model. If for an operator o, $t(o) > 1$, then we create $t(o)$ "clones" of o, i.e., we create $t(o)$ operators that are identical to o but having a unique operator name. In our running example, we have two transitions referring to the Drive operator. So, we create two operators, for instance, Drive-empty and Drive-full that have the identical structure to the original Drive operator (arguments, preconditions, and effects).

A transition (s, o, C, s') is encoded into o (or its corresponding "clone") as follows. Ensuring that instances of o can be applied only if s is the current DCK state is done by adding the corresponding supplementary predicate representing the DCK state (i.e., (DCK-state s)) into $pre^+(o)$. If $s' = s$, then the DCK state does not change, so effects of o remain intact. Otherwise, (DCK-state s) is added into $eff^-(o)$ and (DCK-state s') into $eff^+(o)$, so after applying an instance of o, the current DCK state changes to s'. The Load operator from our running example is modified by adding (DCK-state s0) into the positive preconditions as well as into the negative effects, and by adding (DCK-state s1) into the positive effects.

Each constraint $c \in C$ is encoded into o as follows depending on its form:

- p — add p into $pre^+(o)$
- $\neg p$ — add p into $pre^-(o)$
- g:p — add the "open goal twin" predicate of p into $pre^+(o)$

If some of the predicates that are added into the o's precondition contain arguments that are not defined in o, the list of o's arguments is updated accordingly.

The transition-based DCK as defined in Figure 1 is encoded within the planning operators of our simple logistic domain as depicted in Figure 2. Recall that Drive-empty and Drive-full are "clones" of the original Drive operator.

Solution plans of the DCK enhanced problems may not entirely correspond with solution plans of the original problems because of using different operator names for "cloned operators", and using extra arguments for accommodating additional preconditions. Hence, to get a valid solution plan for the original problem we have to rename all "clones" to the name of the corresponding original operators and remove all the extra arguments. For example, if the solution plan of the DCK enhanced simple logistic problem contains an action Drive-full(truck1 loc1 loc2 pkg1), then it has to be renamed to Drive and the extra argument pkg1 has to be removed, so we obtain Drive(truck1 loc1 loc2) which is a valid action for the corresponding original problem.

Experimental Evaluation

Our experiments aim to demonstrate how transition-based DCK influences the planning process in terms of planners' runtimes and quality of solution plans. We encoded the DCK into the domain models and problems of six domains – Barman, CaveDiving, ChildSnack, CityCar, Hiking, and Nomystery. For descriptions of these domains and related transition-based DCKs we specified, the reader is referred to our workshop paper (Chrpa and Barták 2015). The problem sets for these domains were taken from the agile track of the 8th International Planning Competition (IPC-8), except Nomystery, where the problem set was taken from the satisficing track of the IPC-7. All the problem sets consist of 20 problems. For comparing how the DCK influences the planning process, we have used six state-of-the-art planners that accommodate various planning techniques: LAMA (Richter and Westphal 2010), the winner of the IPC-7, and MpC, Probe, Mercury, Yahsp3, and Bfs-f that performed well in the IPC-8 (Vallati, Chrpa, and McCluskey 2014).

For analysis of planners' performance and quality of solution plans, we use IPC score as defined in the IPC-8 (Vallati et al. 2015). With regards to runtime, the score is calculated as follows. For an encoding e of a problem p, $IPC_t(p, e)$ is 0 if p is unsolved in e, and $1/(1+\log_{10}(T_{p,e}/T^*_p))$, where $T_{p,e}$ is the CPU-time needed to solve p in e and T^*_p is the smallest CPU-time needed to solve p in any considered encoding, otherwise. With regard to quality of solution plans (i.e., the sum of costs of all their actions), the score is calculated as follows. For an encoding e of a problem p, $IPC_q(p, e)$ is 0 if p is unsolved in e, and $(Q^*_p/Q_{p,e}))$, where $Q_{p,e}$ is the cost of the solution plan of p in e and Q^*_p is the smallest cost of the solution plan of p in any considered encoding, otherwise. Notice that the maximum IPC score for a particular p and e is 1. As in the agile track of IPC-8, we set the time limit to 5

Planner	Barman				CaveDiving				ChildSnack				CityCar				Hiking				Nomystery			
	Coverage		Δ IPC		Coverage		Δ IPC		Coverage		Δ IPC		Coverage		Δ IPC		Coverage		Δ IPC		Coverage		Δ IPC	
	O	E	T	Q	O	E	T	Q	O	E	T	Q	O	E	T	Q	O	E	T	Q	O	E	T	Q
Lama	16	20	+8.2	+6.9	6	7	+4.1	+1.0	0	19	+19.0	+19.0	5	20	+15.4	+13.9	5	19	+15.9	+13.9	14	14	+0.3	0.0
Mercury	7	20	+13.4	+15.3	2	3	+1.8	+1.0	5	20	+17.5	+15.0	2	20	+17.9	+18.8	8	17	+11.5	+7.0	13	15	+2.1	+2.0
MpC	0	20	+20.0	+20.0	4	4	+1.4	0.0	7	20	+11.4	+10.9	9	20	+12.8	+15.8	7	3	-3.0	-3.9	6	5	-1.2	-1.0
Probe	18	20	-2.7	+2.8	1	7	+6.4	+6.0	0	15	+15.0	+15.0	8	20	+12.8	+17.3	13	19	+10.9	+5.8	5	11	+7.0	+5.9
Yahsp	0	20	+20.0	+20.0	N/A	N/A	N/A	N/A	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	13	10	+0.6	+4.6	8	12	+7.2	+4.1
Bfs-f	20	20	+3.8	-1.5	7	7	+2.0	0.0	8	8	-0.8	-0.2	5	20	+17.5	+16.7	2	14	+13.4	+11.6	14	15	+4.7	+1.3

Table 1: Comparison between original (O) and enhanced (E) domain models. Δ IPC stands for a difference of the time (T) and quality (Q) IPC score of the original and the corresponding enhanced problem encodings (positive numbers refer to better performance/quality of the enhanced problems).

minutes per problem. All the experiments were run on Intel Xeon 2.53 Ghz with 2GB of RAM, CentOS 6.5.

Table 1 presents the results of comparison of planners' performance and quality of solution plans of the original problem encodings and the encodings enhanced by our transition-based DCK. Notice that Yahsp3 could not solve some problem sets (denoted as "N/A" in the table), since it does not fully support negative preconditions. Remarkable results have been achieved in Barman and CityCar domains, where using DCK allowed every planner (except Yashp3 in CityCar) to solve all the problems within the 5 minute limit. The reason is that in these domains the problems can be solved by using the "reach goals one-by-one" strategy that can be easily captured by transition-based DCK. Good results were achieved in the Childsnack domain, where our transition-based DCK can prevent "trapping" in dead-ends. In CaveDiving and Nomystery the results were modest. The reason is that transition-based DCK does not address the "combinatorial" part of these domains. In Hiking, the results of applying transition-based DCK were mixed, i.e., achieving good improvement for Lama, Mercury, Probe, and Bfs-f, while having rather detrimental effect on MpC and Yahsp3. MpC uses a structure, similar to Planning Graph (Blum and Furst 1997), that allows applying more actions in one step. This strategy is useful in Hiking. Transition-based DCK, however, interferes with such a strategy. In terms of quality of solution plans, there are no (or very marginal) differences in the Cave Diving and Nomystery domains (the quality score differs because the number of solved problems differs). This is because there is usually no other way how to solve these problems, so solution plans are very similar (some actions might be in a different order). In the other domains, the quality results are mixed (sometimes DCK enhanced problems lead to better solution plans, sometimes worse). Notice that in Hiking, Yahsp3 was able to provide considerably better plans (often more than 10x) when DCK was considered.

Conclusions

In this paper, we introduced transition-based DCK, inspired by Finite State Automata, that is planner-independent (can be encoded within planning problems), efficient and easy to specify due to its "schematical" representation.

In future, we plan to (semi)automatise the process of extracting transition-based DCK. We believe that taking inspiration from macro-operator generating techniques and/or tools for plan analysis will be a useful step towards such an achievement. Also, we plan to extend the concept of transition-based DCK to non-classical planning (e.g. temporal or continuous planning).

Acknowledgements The research was funded by the UK EPSRC Autonomous and Intelligent Systems Programme (grant no. EP/J011991/1) and by the Czech Science Foundation (project no. P103-15-19877S). The authors would like to acknowledge the use of the University of Huddersfield Queensgate Grid in carrying out this work.

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Chrpa, L., and Barták, R. 2015. Enhancing domain-independent planning by transition-based domain control knowledge. In *The 33rd Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*.

Georgievski, I., and Aiello, M. 2015. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence* 222:124–156.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning, theory and practice*. Morgan Kaufmann.

Korf, R. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

Kvarnström, J., and Doherty, P. 2000. TALplanner: a temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30(1-4):119–169.

Minton, S., and Carbonell, J. G. 1987. Strategies for learning search control rules: An explanation-based approach. In *Proceedings of IJCAI*, 228–235.

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

Richter, S., and Westphal, M. 2010. The LAMA planner: guiding cost-based anytime planning with landmarks. *Journal Artificial Intelligence Research (JAIR)* 39:127–177.

Vallati, M.; Chrpa, L.; Grzes, M.; McCluskey, T. L.; Roberts, M.; and Sanner, S. 2015. The 2014 international planning competition: Progress and trends. *AI Magazine* 36(3):90–98.

Vallati, M.; Chrpa, L.; and McCluskey, T. L. 2014. *The Eighth International Planning Competition. Description of Participant Planners of the Deterministic Track.*