Procedural Domain Control Knowledge and State-of-the-Art Planners

Jorge A. Baier
Department of Computer Science,
University of Toronto,
Toronto, ON M5S 3G4, CANADA
jabaier@cs.toronto.edu

Introduction

Domain control knowledge (DCK) imposes domain-specific constraints on the definition of a valid plan. As such, it can be used to impose restrictions on the course of action that achieves the goal. While DCK sometimes reflects a user's desire to achieve the goal a particular way, it is most often constructed to aid in plan generation by reducing the plan search space. Moreover, if well-crafted, DCK can eliminate those parts of the search space that necessitate backtracking. In such cases, DCK together with blind search can yield valid plans significantly faster than state-of-the-art (SOA) planners that do not exploit DCK. Indeed most planners that exploit DCK, such as TLPLAN (Bacchus & Kabanza 1998) or TALPLANNER (Kvarnström & Doherty 2000), do little more than blind depth-first search with cycle checking in a DCK-pruned search space. Since most DCK reduces the search space but still requires a planner to backtrack to find a valid plan, it should prove beneficial to exploit better search techniques. In this paper we explore ways in which SOA planning techniques and existing SOA planners can be used in conjunction with DCK, with particular focus on *procedural* DCK.

As a simple example of DCK, consider the `trucks` domain of the 5th International Planning Competition, where the goal is to deliver packages between certain locations using a limited capacity truck. When a package reaches its destination it must be delivered to the customer. We can write simple and natural procedural DCK that significantly improves the efficiency of plan generation for instance: *Repeat the following until all packages have been delivered: Unload everything from the truck, and, if there is any package in the current location whose destination is the current location, deliver it. After that, if any of the local packages have destinations elsewhere, load them on the truck while there is space. Drive to the destination of any of the loaded packages. If there are no packages loaded on the truck, but there remain packages at locations other than their destinations, drive to one of these locations.*

Procedural DCK (as used in HTN (Nau *et al.* 1999) or Golog (Levesque *et al.* 1997)) is action-centric. It is much like a programming language, and often times like a plan skeleton or template. It can (conditionally) constrain the order in which domain actions should appear in a plan. In order to exploit it for planning, we require a procedural DCK specification language. To this end, we propose a language based on GOLOG that includes typical programming languages constructs such as conditionals and iteration as well as nondeterministic choice of actions in places where control is not germane. We argue that these action-centric constructs provide a natural language for specifying DCK for planning. We contrast them with DCK specifications based on linear temporal logic (LTL) which are state-centric and though still of tremendous value, arguably provide a less natural way to specify DCK. We specify the syntax for our language as well as a PDDL-based semantics following Fox & Long (2003).

With a well-defined procedural DCK language in hand, we examine how to use SOA planning techniques together with DCK. Of course, most SOA planners are unable to exploit DCK. As such, we present an algorithm that translates a PDDL2.1-specified ADL planning instance and associated procedural DCK into an equivalent, program-free PDDL2.1 instance whose plans provably adhere to the DCK. Any PDDL2.1-compliant planner can take such a planning instance as input to their planner, generating a plan that adheres to the DCK.

Since they were not designed for this purpose, existing SOA planners may not exploit techniques that optimally leverage the DCK embedded in the planning instance. As such, we investigate how SOA planning techniques, rather than planners, can be used in conjunction with our compiled DCK planning instances. In particular, we propose domain-independent search heuristics for planning with our newly-generated planning instances. We examine three different approaches to generating heuristics, and evaluate them on three domains of the 5th International Planning Competition. Our results show that procedural DCK improves the performance of SOA planners, and that our heuristics are sometimes key to achieving good performance.

This document is an extended abstract of the ICAPS-07 paper by Baier, Fritz, & McIlraith (2007).

A Language for Procedural Control

The language we use to represent procedural DCK is a variant of the agent programming language Golog (Levesque *et al.* 1997), developed by the cognitive robotics community.

The set of programs over a planning domain D, types T, and a set of *program variables* V can be defined inductively.

Let the set of operators defined in D, possibly with arguments in V be $\mathcal{O}$. The most simple programs are as follows.

1. nil: Represents the empty program.
2. o: Is a single operator instance, where $o \in \mathcal{O}$.
3. **any**: A keyword denoting "any action".
4. $\phi?$: A *test action*. ϕ is a Boolean formula with quantifiers on the language of D, possibly including arguments in V.

If σ_1, σ_2 and σ are programs, so are the following:

1. $(\sigma_1; \sigma_2)$: A sequence of programs.
2. $\mathbf{if}\, \phi\, \mathbf{then}\, \sigma_1\, \mathbf{else}\, \sigma_2$: A conditional sentence.
3. $\mathbf{while}\, \phi\, \mathbf{do}\, \sigma$: An iteration.
4. σ^*: A non-deterministic iteration.
5. $(\sigma_1 | \sigma_2)$: Non-deterministic choice between programs.
6. $\pi(x\text{-}t)\, \sigma$: Non-deterministic choice of variable $x \in V$ of type $t \in T$.

Here are some examples that give a sense of the language's expressiveness and semantics.

- $\mathbf{while}\, \neg clear(B)\, \mathbf{do}\, \pi(b\text{-}block)\, putOnTable(b)$: while B is not clear, choose any b of type block and put it on the table.
- $(\mathbf{any})^*; loaded(A, Truck)?$: Perform any sequence of actions until A is loaded in the $Truck$. A plan under this control is any plan where $loaded(A, Truck)$ is true in the final state.
- $(load(C, P); fly(P, LA) \,|\, load(C, T); drive(T, LA))$: Either load C on the plane P or on the truck T, and perform the right action to move the vehicle to LA.

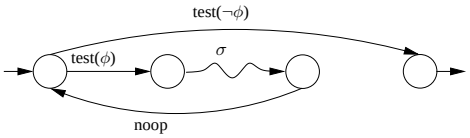
In this extended abstract we do not formalize the semantics of the languages. The interested reader is referred to the longer version of this paper (Baier, Fritz, & McIlraith 2007).

Compiling Control into the Action Theory

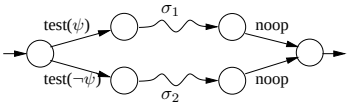
In order to enable any PDDL2.1 compliant planner to use our DCK, we define a compilation from programs in our DCK to PDDL2.1. More specifically, given a planning instance comprised of a domain D and a planning problem P, and a program σ to control the search for plans, we generate a new planning instance with domain D_σ and problem P_σ, such that, roughly, a sequence of actions in D_σ is a plan for the new problem if and only if it is a plan for the original problem and it is consistent with the program.

We here focus on the intuition behind the compilation using an example. Consider the program $\mathbf{while}\, \phi\, \mathbf{do}\, (\mathbf{if}\, \psi\, \mathbf{then}\, a\, \mathbf{else}\, b); c$, where a, b, and c are domain actions and ϕ and ψ are formulae over features of the domain. In any particular state of the domain we can determine whether these formulae hold or not. Intuitively, the compilation works by creating an automaton whose accepted language is precisely the set of legal executions of the program. We achieve this by inductively compiling each occurrence of a programming construct into an automaton and nesting the set of resulting automata appropriately. To describe the state of the automaton and its transitions in PDDL2.1, we need add a few bookkeeping predicates and functions to the domain and generate some additional actions.

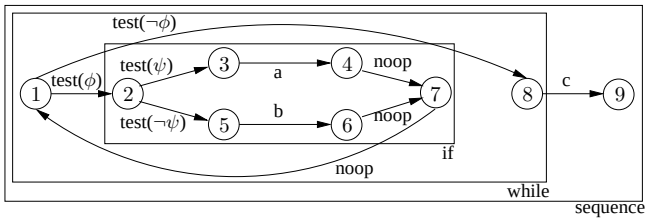
As an illustrative example, let us compile the above program step-by-step graphically. The automaton that we generate to capture the semantics of a while loop $\mathbf{while}\, \phi\, \mathbf{do}\, \sigma$, looks like this:



That is, if the condition holds, we enter the sub-automaton for the body (σ) and after executing the body, we return to the state where the condition is checked. Does the condition not hold, we go on to the next state after the while loop. Similarly for a conditional statement $\mathbf{if}\, \psi\, \mathbf{then}\, \sigma_1\, \mathbf{else}\, \sigma_2$ we have:



The automaton for a sequence of actions is a simple chain of three states. In total, we generate the following automaton for above mentioned program $(\mathbf{while}\, \phi\, \mathbf{do}\, (\mathbf{if}\, \psi\, \mathbf{then}\, a\, \mathbf{else}\, b); c)$:



The numbers in the nodes denote the abstract program state which we encode in the PDDL translation using a new function `state`. This plays a central part in enforcing the search control: by modifying the preconditions of domain actions and generating the preconditions of additional actions (like `test` and `noop`) accordingly, we limit actions in their applicability to only those states where they are allowed by the automaton. For instance, in the initial state (1) of the example, the only action possible is one of two `test` actions, namely either $test(\phi)$ or $test(\neg\phi)$, depending on whether ϕ holds in the initial state.

In the full version of this paper we proove that our compilation is correct, that is, that it allows the same execution traces as the program does. And we also show that the compilation result is at most linearly bigger than the program itself. This is significant, as previously proposed compilation approaches of other DCK languages caused an exponential blow-up.

Heuristics for Compiled Domains In the extended version we give details of 3 approaches to integrating our compiled domains with domain-independent heuristic planners that use the well-known technique of relaxing negative effects of actions.

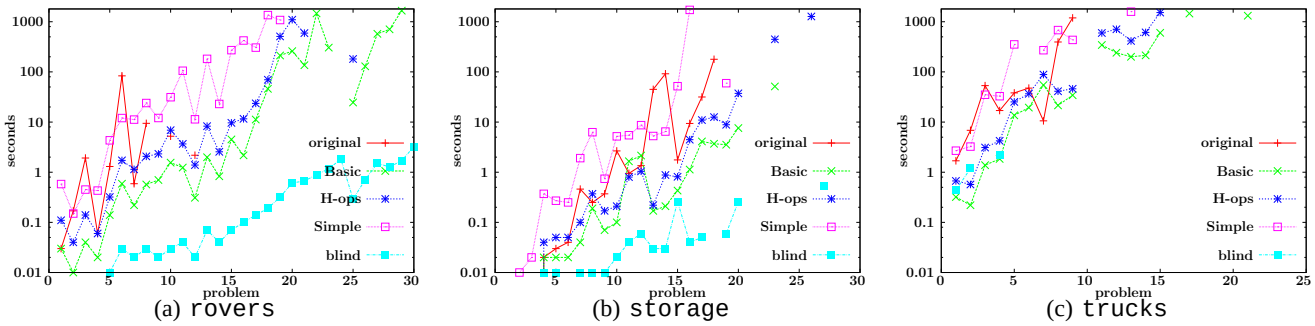


Figure 1: Running times of the three heuristics and the original instance; logarithmic scale; run on an Intel Xeon, 3.6GHz, 2GB RAM

		original	*Simple*	*Basic*	*H-ops*	*blind*
Trucks	#n	1	0.31	0.41	0.26	19.85
	#s	9	9	15	14	3
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	1.1	1.03	1.02	1.04	1.04
	ℓ_{max}	1.2	1.2	1.07	1.2	1.07
Rovers	#n	1	0.74	1.06	1.06	1.62
	#s	10	19	28	22	30
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	2.13	1.03	1.05	1.21	1.53
	ℓ_{max}	4.59	1.2	1.3	1.7	2.14
Storage	#n	1	1.2	1.13	0.76	1.45
	#s	18	18	20	21	20
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	4.4	1.05	1.01	1.07	1.62
	ℓ_{max}	21.11	1.29	1.16	1.48	2.11

Table 1: Comparison between different approaches to planning (with DCK). #n is the average factor of expanded nodes to the number of nodes expanded by *original* (i.e., #n=0.26 means the approach expanded 0.26 times the number of nodes expanded by original). #s is the number of problems solved by each approach. ℓ_{avg} denotes the average ratio of the plan length to the shortest plan found by any of the approaches (i.e., ℓ_{avg}=1.50 means that on average, on each instance, plans where 50% longer than the shortest plan found for that instance). ℓ_{min} and ℓ_{max} are defined analogously.

Our first approach—the *Simple* heuristic—consists of using the compiled problem directly in the planner. On the other hand, *Basic* uses the SOA techniques to guide the search but *ignoring* the program. Finally, *H-ops* computes the heuristics over a *relaxed version of the program*.

Implementation and Experiments

Our planner is a modified version of TLPLAN, which does a best-first search using an FF-style heuristic. We performed our experiments on the *trucks*, *storage* and *rovers* domains (30 instances each). We wrote DCK for these domains. For lack of space, we do not show the DCK in detail, however for trucks we used the control shown as an example in the Introduction. We ran our three heuristic approaches (*Basic*, *H-ops*, and *Simple*) and cycle-free, depth-first search on the translated instance (*blind*). Additionally, we ran the original instance of the program (DCK-free) using the domain-independent heuristics provided by the planner (*original*). Table 1 shows various statistics on the performance of the approaches. Furthermore, Fig. 1 shows times for the different heuristic approaches.

Not surprisingly, our data confirms that DCK helps to improve the performance of the planner, solving more instances across all domains. In some domains (i.e. storage and rovers) blind depth-first cycle-free search is sufficient for solving most of the instances. However, quality of solutions (plan length) is poor compared to the heuristic approaches. In trucks, DCK is only effective in conjunction with heuristics; blind search can solve very few instances.

We observe that *H-ops* is the most informative (expands fewer nodes). This fact does not pay off in time in the experiments shown in the table. Nevertheless, it is easy to construct instances where the *H-ops* performs better than *Basic*. This happens when the DCK control restricts the space of valid plans (i.e., prunes out valid plans). We have experimented with various instances of the storage domain, where we restrict the plan to use only one hoist. In some of these cases *H-ops* outperforms *Basic* by orders of magnitude.

References

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Math and AI* 22(1-2):5–27.

Baier, J. A.; Fritz, C.; and McIlraith, S. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS-07*. to appear.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Kvarnström, J., and Doherty, P. 2000. TALPlanner: A temporal logic based forward chaining planner. *Annals of Math and AI* 30(1-4):119–169.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *JLP* 31(1-3):59–83.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *IJCAI-99*, 968–975.