Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners (extended version)

Jorge A. Baier Christian Fritz Sheila A. McIlraith
Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, CANADA
{jabaier,fritz,sheila}@cs.toronto.edu

Abstract

Domain control knowledge (DCK) has proven effective in improving the efficiency of plan generation by reducing the search space for a plan. *Procedural* DCK is a compelling type of DCK that supports a natural specification of the skeleton of a plan. Unfortunately, most state-of-the-art planners do not have the machinery necessary to exploit procedural DCK. To resolve this deficiency, we propose to compile procedural DCK directly into PDDL2.1, thus enabling any PDDL2.1-compatible planner to exploit it. The contribution of this paper is threefold. First, we propose a PDDL-based semantics for an Algol-like, procedural language that can be used to specify DCK in planning. Second, we provide a polynomial algorithm that translates an ADL planning instance and a DCK program, into an equivalent, program-free PDDL2.1 instance whose plans are only those that adhere to the program. Third, we argue that the resulting planning instance is well-suited to being solved by domain-independent heuristic planners. To this end, we propose three approaches to computing domain-independent heuristics for our translated instances, sometimes leveraging properties of our translation to guide search. In our experiments on familiar PDDL planning benchmarks we show that the proposed compilation of procedural DCK can significantly speed up the performance of a heuristic search planner. Our translators are implemented and available on the web.

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.

Background

A Subset of PDDL 2.1

A *planning instance* is a pair $I = (D, P)$, where D is a domain definition and P is a problem. To simplify notation, we assume that D and P are described in an ADL subset of PDDL. The difference between this ADL subset and PDDL 2.1 is that no concurrent or durative actions are allowed.

Following convention, domains are tuples of finite sets $(PF, Ops, Objs_D, T, \tau_D)$, where PF defines domain predicates and functions, Ops defines operators, $Objs_D$ contains domain objects, T is a set of types, and $\tau_D \subseteq Objs_D \times T$ is a type relation associating objects to types. An operator (or action schema) is also a tuple $\langle O(\vec{x}), \vec{t}, Prec(\vec{x}), \mathit{Eff}(\vec{x})\rangle$, where $O(\vec{x})$ is the unique operator name and $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables. Furthermore, $\vec{t} = (t_1, \ldots, t_n)$ is a vector of types. Each variable x_i ranges over objects associated with type t_i. Moreover, $Prec(\vec{x})$ is a boolean formula with quantifiers (BQF) that specifies the operator's preconditions. BFQs are defined inductively as follows. Atomic BFQs are either of the form $t_1 = t_2$ or $R(t_1, \ldots, t_n)$, where t_i ($i \in \{1, \ldots, n\}$) is a term (i.e. either a variable, a function literal, or an object), and R is a predicate symbol. If φ is a BFQ, then so is $Qx\text{-}t\,\varphi$, for a variable x, a type symbol t, and $Q \in \{\exists, \forall\}$. BFQs are also formed by applying standard boolean operators over other BFQs. Finally $\mathit{Eff}(\vec{x})$ is a list of conditional effects, each of which can be in one of the following forms:

$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow R(\vec{x}, \vec{y}), \tag{1}$$
$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow \neg R(\vec{x}, \vec{y}), \tag{2}$$
$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow f(\vec{x}, \vec{y}) = obj, \tag{3}$$

where φ is a BFQ whose only free variables are among $\vec{x}$ and $\vec{y}$, R is a predicate, f is a function, and obj is an object After performing a ground operator – or *action* – $O(\vec{c})$ in a certain state s, for all tuples of objects that may instantiate $\vec{y}$ such that $\varphi(\vec{c}, \vec{y})$ holds in s, effect (1) (resp. (2)) expresses that $R(\vec{c}, \vec{y})$ becomes true (resp. false), and effect (3) expresses that $f(\vec{c}, \vec{y})$ takes the value obj. As usual, states are represented as finite sets of atoms (ground formulae of the form $R(\vec{c})$ or of the form $f(\vec{c}) = obj$).

Planning problems are tuples $(Init, Goal, Objs_P, \tau_P)$, where $Init$ is the initial state, $Goal$ is a sentence with quantifiers for the goal, and $Objs_P$ and τ_P are defined analogously as for domains.

Semantics: Fox & Long (2003) have given a formal semantics for PDDL 2.1. In particular, they define when a sentence is *true* in a state and what *state trace* is the result of performing a set of *timed actions*. A state trace intuitively corresponds to an execution trace, and the sets of timed actions are ultimately used to refer to plans. In the ADL subset of PDDL2.1, since there are no concurrent or durative actions, time does not play any role. Hence, state traces reduce to sequences of states and sets of timed actions reduce to sequences of actions.

Building on Fox and Long's semantics, we assume that $\models$ is defined such that $s \models \varphi$ holds when sentence φ is true in state s. Moreover, for a planning instance I, we assume there exists a relation $Succ$ such that $Succ(s, a, s')$ iff s' results from performing an executable action a in s. Finally, a sequence of actions $a_1 \cdots a_n$ is a plan for I if there exists a sequence of states $s_0 \cdots s_n$ such that $s_0 = Init$, $Succ(s_i, a_{i+1}, s_{i+1})$ for $i \in \{0, \ldots, n-1\}$, and $s_n \models Goal$.

Domain-Independent Heuristics for Planning

In sections to follow, we investigate how procedural DCK integrates into SOA domain-independent planners. Domain-independent heuristics are key to the performance of these planners. Among the best known heuristic-search planners are those that compute their heuristic by solving a relaxed STRIPS planning instance (e.g., as done in HSP (Bonet & Geffner 2001) and FF (Hoffmann & Nebel 2001) planners). Such a relaxation corresponds to solving the same planning problem but on an instance that ignores deletes (i.e. ignores negative effects of actions).

For example, the FF heuristics for a state s is computed by expanding a *relaxed planning graph* (Hoffmann & Nebel 2001) from s. We can view this graph as composed of *relaxed states*. A relaxed state at depth $n + 1$ is generated by *adding* all the effects of actions that can be performed in the relaxed state of depth n, and then by copying all facts that appear in layer n. The graph is expanded until the goal or a fixed point is reached. The heuristic value for s corresponds to the number of actions in a *relaxed plan* for the goal, which can be extracted in polynomial time.

Both FF-like heuristics and HSP-like heuristics can be computed for (more expressive) ADL planning problems.

A Language for Procedural Control

In contrast to state-centric languages, that often use LTL-like logical formulae to specify properties of the states traversed during plan execution, procedural DCK specification languages are predominantly action-centric, defining a plan template or skeleton that dictates *actions* to be used at various stages of the plan.

Procedural control is specified via *programs* rather than logical expressions. The specification language for these programs incorporates desirable elements from imperative programming languages such as iteration and conditional constructs. However, to make the language more suitable to planning applications, it also incorporates nondeterministic constructs. These elements are key to writing flexible control since they allow programs to contain missing or open program segments, which are filled in by a planner at the time of plan generation. Finally, our language also incorporates property testing, achieved through so-called *test actions*. These actions are not real actions, in the sense that they do not change the state of the world, rather they can be used to specify properties of the states traversed while executing the plan. By using test actions, our programs can also specify properties of executions similarly to state-centric specification languages.

The rest of this section describes the syntax and semantics of the procedural DCK specification language we propose to use. We conclude this section by formally defining what it means to plan under the control of such programs.

Syntax

The language we propose is based on GOLOG (Levesque *et al.* 1997), a robot programming language developed by the cognitive robotics community. In contrast to GOLOG, our language supports specification of types for program variables, but does not support procedures.

Programs are constructed using the implicit language for actions and boolean formulae defined by a particular planning instance I. Additionally, a program may refer to variables drawn from a set of program variables V. This set V will contain variables that are used for nondeterministic choices of arguments. In what follows, we assume $\mathcal{O}$ denotes the set of operator names from Ops, fully instantiated with objects defined in I or elements of V.

The set of programs over a planning instance I and a set of program variables V can be defined by induction. In what follows, assume ϕ is a boolean formula with quantifiers on the language of I, possibly including terms in the set of program variables V. Atomic programs are as follows.

1. nil: Represents the empty program.
2. o: Is a single operator instance, where $o \in \mathcal{O}$.
3. $\mathbf{any}$: A keyword denoting “any action”.
4. $\phi?$: A *test action*.

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. $\mathbf{while}\,\phi\,\mathbf{do}\,\sigma$: A while-loop.
4. σ^*: A nondeterministic iteration.
5. $(\sigma_1|\sigma_2)$: Nondeterministic choice between two programs.
6. $\pi(x\text{-}t)\,\sigma$: Nondeterministic choice of variable $x \in V$ of type $t \in T$.

Before we formally define the semantics of the language, we show 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 $Truck$. Plans under this control are such that $loaded(A, Truck)$ holds 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.

Semantics

The problem of planning for an instance I under the control of program σ corresponds to finding a plan for I that is also an execution of σ from the initial state. In the rest of this section we define what those legal executions are. Intuitively, we define a formal device to check whether a sequence of actions $\vec{a}$ corresponds to the execution of a program σ. The device we use is a nondeterministic finite state automaton with ε-transitions (ε-NFA).

For the sake of readability, we remind the reader that ε-NFAs are like standard nondeterministic automata except that they can transition without reading any input symbol, through the so-called ε-transitions. ε-transitions are usually defined over a state of the automaton and a special symbol ε, denoting the empty symbol.

An ε-NFA $A_{\sigma,I}$ is defined for each program σ and each planning instance I. Its alphabet is the set of operator names, instantiated by objects of I. Its states are *program configurations* which have the form $[\sigma, s]$, where σ is a program and s is a planning state. Intuitively, as it reads a word of actions, it keeps track, within its state $[\sigma, s]$, of the part of the program that remains to be executed, σ, as well as the current planning state after performing the actions it has read already, s.

Formally, $A_{\sigma,I} = (Q, \mathcal{A}, \delta, q_o, F)$, where Q is the set of program configurations, the alphabet $\mathcal{A}$ is a set of domain actions, the transition function is $\delta : Q \times (\mathcal{A} \cup \{\varepsilon\}) \rightarrow 2^Q$, $q_0 = [\sigma, Init]$, and F is the set of final states. The transition function δ is defined as follows for atomic programs.

$$\delta([a, s], a) = \{[nil, s']\} \text{ iff } Succ(s, a, s'),\ \text{s.t. } a \in \mathcal{A}, \tag{4}$$

$$\delta([\mathbf{any}, s], a) = \{[nil, s']\} \text{ iff } Succ(s, a, s'),\ \text{s.t. } a \in \mathcal{A}, \tag{5}$$

$$\delta([\phi?, s], \varepsilon) = \{[nil, s]\} \text{ iff } s \models \phi. \tag{6}$$

Equations 4 and 5 dictate that actions in programs change the state according to the $Succ$ relation. Finally, Eq. 6 defines transitions for $\phi?$ when ϕ is a sentence (i.e., a formula with no program variables). It expresses that a transition can only be carried out if the plan state so far satisfies ϕ.

Now we define δ for non-atomic programs. In the definitions below, assume that $a \in \mathcal{A} \cup \{\varepsilon\}$, and that σ_1 and σ_2 are subprograms of σ, where occurring elements in V may have been instantiated by any object in the planning instance I.

$$\delta([(\sigma_1; \sigma_2), s], a) = \bigcup_{[\sigma_1', s'] \in \delta([\sigma_1, s], a)} \{[(\sigma_1'; \sigma_2), s']\}\ \text{ if } \sigma_1 \neq nil, \tag{7}$$

$$\delta([(nil; \sigma_2), s], a) = \delta([\sigma_2, s], a), \tag{8}$$

$$\delta([\mathbf{if}\,\phi\,\mathbf{then}\,\sigma_1\,\mathbf{else}\,\sigma_2, s], a) = \begin{cases} \delta([\sigma_1, s], a) & \text{if } s \models \phi, \\ \delta([\sigma_2, s], a) & \text{if } s \not\models \phi, \end{cases}$$

$$\delta([(\sigma_1|\sigma_2), s], a) = \delta([\sigma_1, s], a) \cup \delta([\sigma_2, s], a),$$

$$\delta([\mathbf{while}\,\phi\,\mathbf{do}\,\sigma_1, s], a) = \begin{cases} \{[nil, s]\} & \text{if } s \not\models \phi \text{ and } a = \varepsilon, \\ \delta([\sigma_1; \mathbf{while}\,\phi\,\mathbf{do}\,\sigma_1, s], a) & \text{if } s \models \phi, \end{cases}$$

$$\delta([\sigma_1^*, s], a) = \delta([(\sigma_1; \sigma_1^*), s], a)\ \ \text{if } a \neq \varepsilon \tag{9}$$

$$\delta([\sigma_1^*, s], \varepsilon) = \delta([(\sigma_1; \sigma_1^*), s], \varepsilon) \cup \{[nil, s]\}, \qquad (10)$$

$$\delta([\pi(x\text{-}t)\, \sigma_1, s], a) = \bigcup_{(o,t)\in\tau_D\cup\tau_P} \delta([\sigma_1|_{x/o}, s], a). \qquad (11)$$

where $\sigma_1|_{x/o}$ denotes the program resulting from replacing any occurrence of x in σ_1 by o. For space reasons we only explain two of them. First, a transition on a sequence corresponds to transitioning on its first component first (Eq. 7), unless the first component is already the empty program, in which case we transition on the second component (Eq. 8). On the other hand, a transition of σ_1^* represents two alternatives: executing σ_1 at least once, or stopping the execution of σ_1^*, with the remaining program nil (Eq. 9, 10).

To end the definition of $A_{\sigma,I}$, Q corresponds precisely to the program configurations $[\sigma', s]$ where σ' is either nil or a subprogram of σ such that program variables may have been replaced by objects in I, and s is any possible planning state. Moreover, δ is assumed empty for elements of its domain not explicitly mentioned above. Finally, the set of accepting states is $F = \{[nil, s] \mid s$ is any state over $I\}$, i.e., those where no program remains in execution. We can now formally define an execution of a program.

Definition 1 (Execution of a program)**.** A sequence of actions $a_1 \cdots a_n$ is an execution of σ in I if $a_1 \cdots a_n$ is accepted by $A_{\sigma,I}$.

The following remark illustrates how the automaton transitions in order to accept executions of a program.

Remark 1. Let $\sigma = (\mathbf{if}\, \varphi\, \mathbf{then}\, a\, \mathbf{else}\, b; c)$, and suppose that $Init$ is the initial state of planning instance I. Assume furthermore that a, b, and c are always possible. Then $A_{\sigma,I}$ accepts ac if $Init \models \varphi$.

Proof. Suppose $q \vdash_a q'$ denotes that $A_{\sigma,I}$ can transition from q to q' by reading symbol a. Then if $Init \models \varphi$ observe that $[\sigma, Init] \vdash_a [nil; c, s_2] \vdash_c [nil, s_3]$, for some planning states s_2 and s_3.

Now that we have defined those sequences of actions corresponding to the execution of our program, we are ready to define the notion of planning under procedural control.

Definition 2 (Planning under procedural control)**.** A sequence of action $\vec{a}$ is a *plan for instance I under the control of program σ* if $\vec{a}$ is a plan in I and is an execution of σ in I.

Compiling Control into the Action Theory

This section describes a translation function that, given a program σ in the DCK language defined above together with a PDDL2.1 domain specification D, outputs a new PDDL2.1 domain specification D_σ and problem specification P_σ. The two resulting specifications can then be combined with any problem P defined over D, creating a new planning instance that embeds the control given by σ, i.e. that is such that only action sequences that are executions of σ are possible. This enables any PDDL2.1-compliant planner to exploit search control specified by any program.

To account for the state of execution of program σ and to describe legal transitions in that program, we introduce a few bookkeeping predicates and a few additional actions. Figure 1 graphically illustrates the translation of an example program shown as a *finite state automaton*. Intuitively, the operators we generate in the compilation define the transitions of this automaton. Their preconditions and effects condition on and change the automaton's state.

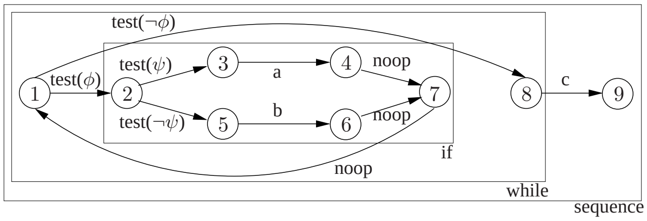


Figure 1: Automaton for $\mathbf{while}\, \phi\, \mathbf{do}\, (\mathbf{if}\, \psi\, \mathbf{then}\, a\, \mathbf{else}\, b); c$.

The translation is defined inductively by a function $C(\sigma, n, E)$ which takes as input a program σ, an integer n, and a list of program variables with types $E = [e_1\text{-}t_1, \ldots, e_k\text{-}t_k]$, and outputs a tuple (L, L', n') with L a list of domain-independent operator definitions, L' a list of domain-dependent operator definitions, and n' another integer. Intuitively, E contains the program variables whose scope includes (sub-)program σ. Moreover, L' contains restrictions on the applicability of operators defined in D, and L contains additional control operators needed to enforce the search control defined in σ. Integers n and n' abstractly denote the program state before and after execution of σ.

We use two auxiliary functions. $Cnoop(n_1, n_2)$ produces an operator definition that allows a transition from state n_1 to n_2. Similarly $Ctest(\phi, n_1, n_2, E)$ defines the same transition, but conditioned on ϕ. They are defined as:[1]

$$Cnoop(n_1, n_2) = \langle noop\_n_1\_n_2(), [\,], state = s_{n_1}, [state = s_{n_2}]\rangle$$

$$Ctest(\phi, n_1, n_2, E) = \langle test\_n_1\_n_2(\vec{x}), \vec{t}, Prec(\vec{x}), E\!f\!f(\vec{x})\rangle \text{ with}$$
$$(\vec{e\text{-}t}, \vec{x}) = mentions(\phi, E),\ \vec{e\text{-}t} = e_1\text{-}t_1, \ldots, e_m\text{-}t_m,$$
$$Prec(\vec{x}) = \big(state = s_{n_1} \wedge\ \phi[e_i/x_i]_{i=1}^m \wedge$$
$$\bigwedge_{i=1}^m bound(e_i) \rightarrow map(e_i, x_i)\big),$$
$$E\!f\!f(\vec{x}) = [state = s_{n_2}] \cdot [bound(e_i), map(e_i, x_i)]_{i=1}^m.$$

Function $mentions(\phi, E)$ returns a vector $\vec{e\text{-}t}$ of program variables and types that occur in ϕ, and a vector $\vec{x}$ of new variables of the same length. Bookkeeping predicates serve the following purposes: *state* denotes the state of the automaton; *bound*(e) expresses that the program variable e has been bound to an object of the domain; *map*(e, o) states that this object is o. Thus, the implication *bound*$(e_i) \rightarrow$ *map*(e_i, x_i) forces parameter x_i to take the value to which e_i is bound, but has no effect if e_i is not bound.

Consider the inner box of Figure 1, depicting the compilation of the if statement. It is defined as:

$$C(\mathbf{if}\, \phi\, \mathbf{then}\, \sigma_1\, \mathbf{else}\, \sigma_2, n, E) = (L_1 \cdot L_2 \cdot X, L'_1 \cdot L'_2, n_3)$$
$$\text{with } (L_1, L'_1, n_1) = C(\sigma_1, n+1, E),$$
$$(L_2, L'_2, n_2) = C(\sigma_2, n_1 + 1, E),\ n_3 = n_2 + 1,$$
$$X = [\, Ctest(\phi, n, n+1, E),\ Ctest(\neg\phi, n, n_1 + 1, E),$$
$$Cnoop(n_1, n_3),\ Cnoop(n_2, n_3)\,]$$

and in the example we have $\phi = \psi, n = 2, n_1 = 4, n_2 = 6, n_3 = 7, \sigma_1 = a$, and $\sigma_2 = b$.

[1] We use $A \cdot B$ to denote the concatenation of lists A and B.

The inductive definitions for other programs σ are:

$$\begin{aligned}
&C(nil,n,E) = ([\,],[\,],n)\\
&C(O(\vec{r}),n,E) = ([\,],[\langle O(\vec{x}),\vec{t},\mathit{Prec}'(\vec{x}),\mathit{Eff}'(\vec{x})\rangle],n+1) \text{ with}\\
&\quad \langle O(\vec{x}),\vec{t},\mathit{Prec}(\vec{x}),\mathit{Eff}(\vec{x})\rangle \in \mathit{Ops},\ \ \vec{r}=r_1,\ldots,r_m,\\
&\quad \mathit{Prec}'(\vec{x}) = (\mathit{state}=s_n \wedge\\
&\qquad \bigwedge_{i \text{ s.t. } r_i\in E} \mathit{bound}(r_i) \rightarrow \mathit{map}(r_i,x_i) \wedge \bigwedge_{i \text{ s.t. } r_i\notin E} x_i = r_i),\\
&\quad \mathit{Eff}'(\vec{x}) = [\,\mathit{state}=s_n \Rightarrow \mathit{state}=s_{n+1}] \cdot\\
&\qquad\quad [\mathit{state}=s_n \Rightarrow \mathit{bound}(r_i) \wedge \mathit{map}(r_i,x_i)]_{i \text{ s.t. } r_i\in E}\\
&C(\phi?,n,E) = (\,[Ctest(\phi,n,n+1,E)],\ [\,],\ n+1)\\
&C((\sigma_1;\sigma_2),n,E) = (L_1\cdot L_2,\ L'_1\cdot L'_2,\ n_2) \text{ with}\\
&\quad (L_1,L'_1,n_1)=C(\sigma_1,n,E),(L_2,L'_2,n_2)=C(\sigma_2,n_1,E)\\
&C((\sigma_1|\sigma_2),n,E) = (L_1\cdot L_2\cdot X, L'_1\cdot L'_2, n_2+1) \text{ with}\\
&\quad (L_1,L'_1,n_1)=C(\sigma_1,n+1,E),\\
&\quad (L_2,L'_2,n_2)=C(\sigma_2,n_1+1,E),\\
&\quad X=[\,\mathit{Cnoop}(n,n+1),\ \mathit{Cnoop}(n,n_1+1),\\
&\qquad\quad \mathit{Cnoop}(n_1,n_2+1),\ \mathit{Cnoop}(n_2,n_2+1)\,]\\
&C(\mathbf{while}\,\phi\,\mathbf{do}\,\sigma,n,E) = (L\cdot X,L',n_1+1) \text{ with}\\
&\quad (L,L',n_1)=C(\sigma,n+1,E),\ X=[Ctest(\phi,n,n+1,E),\\
&\quad Ctest(\neg\phi,n,n_1+1,E),\mathit{Cnoop}(n_1,n)]\\
&C(\sigma^*,n,E) = (L\cdot[\mathit{Cnoop}(n,n_2),\mathit{Cnoop}(n_1,n)],L',n_2)\\
&\quad \text{with } (L,L',n_1)=C(\sigma,n,E), n_2=n_1+1\\
&C(\pi(x\text{-}t,\sigma),n,E) = (L\cdot X,L',n_1+1) \text{ with}\\
&\quad (L,L',n_1)=C(\sigma,n,E\cdot[x\text{-}t]),\\
&\quad X=[\langle free\_n_1(x),t,\ \mathit{state}=s_{n_1},\\
&\qquad\quad [\mathit{state}=s_{n_1+1},\neg\mathit{bound}(x),\forall y.\neg\mathit{map}(x,y)]\rangle\,]
\end{aligned}$$

The atomic program **any** is handled by macro expansion to above defined constructs.

As mentioned above, given program σ, the return value (L,L',n_{final}) of $C(\sigma,0,[\,])$ is such that L contains new operators for encoding transitions in the automaton, whereas L' contains restrictions on the applicability of the original operators of the domain. Now we are ready to integrate these new operators and restrictions with the original domain specification D to produce the new domain specification D_σ.

D_σ contains a constrained version of the operators $O(\vec{x})$ of the original domain D also mentioned in L'. Let $[\langle O(\vec{x}),\vec{t},\mathit{Prec}_i(\vec{x}),\mathit{Eff}_i(\vec{x})\rangle]_{i=1}^n$ be the sublist of L' that contains additional conditions for operator $O(\vec{x})$. The operator replacing $O(\vec{x})$ in D_σ is defined as:

$$\langle O'(\vec{x}),\ \vec{t},\ \mathit{Prec}(\vec{x}) \wedge \bigvee_{i=1}^{n} \mathit{Prec}_i(\vec{x}),\ \mathit{Eff}(\vec{x}) \cup \bigcup_{i=1}^{n} \mathit{Eff}_i(\vec{x})\rangle$$

Additionally, D_σ contains all operator definitions in L. Objects in D_σ are the same as those in D, plus a few new ones to represent the program variables and the automaton's states s_i ($0 \le i \le n_{\text{final}}$). Finally D_σ inherits all predicates in D plus *bound*(x), *map*(x,y), and function $state$.

The translation, up to this point, is problem-independent; the problem specification P_σ is defined as follows. Given any predefined problem P over D, P_σ is like P except that its initial state contains condition *state* $= s_0$, and its goal contains *state* $= s_{n_{\text{final}}}$. Those conditions ensure that the program must be executed to completion.

As is shown below, planning in the generated instance $I_\sigma = (D_\sigma,P_\sigma)$ is equivalent to planning for the original instance $I = (D,P)$ under the control of program σ, except that plans on I_σ contain actions that were not part of the original domain definition (*test*, *noop*, and *free*).

Theorem 1 (Correctness)**.** Let *Filter*$(\vec{a},D)$ denote the sequence that remains when removing from $\vec{a}$ any action not defined in D. If $\vec{a}$ is a plan for instance $I_\sigma = (D_\sigma,P_\sigma)$ then *Filter*$(\vec{a},D)$ is a plan for $I=(D,P)$ under the control of σ. Conversely, if $\vec{a}$ is a plan for I under the control of σ, there exists a plan $\vec{a}'$ for I_σ, such that $\vec{a} =$ *Filter*$(\vec{a}',D)$.
Proof. See the appendix.

Now we turn our attention to analyzing the succinctness of the output planning instance relative to the original instance and control program. Assume we define the size of a program as the number of programming constructs and actions it contains. Then we obtain the following result.

Theorem 2 (Succinctness)**.** If σ is a program of size m, and k is the maximal nesting depth of $\pi(x\text{-}t)$ statements in σ, then $|I_\sigma|$ (the overall size of I_σ) is $O(km)$.
Proof. See the appendix.

The encoding of programs in PDDL2.1 is, hence, in worst case $O(k)$ times bigger than the program itself. It is also easy to show that the translation is done in time linear in the size of the program, since, by definition, every occurrence of a program construct is only dealt with once.

Exploiting DCK in SOA Heuristic Planners

Our objective in translating procedural DCK to PDDL2.1 was to enable *any* PDDL2.1-compliant SOA planner to seamlessly exploit our DCK. In this section, we investigate ways to best leverage our translated domains using domain-independent heuristic search planners.

There are several compelling reasons for wanting to apply domain-independent heuristic search to these problems. Procedural DCK can take many forms. Often, it will provide explicit actions for some parts of a sequential plan, but not for others. In such cases, it will contain unconstrained fragments (i.e., fragments with nondeterministic choices of actions) where the designer expects the planner to figure out the best choice of actions to realize a sub-task. In the absence of domain-specific guidance for these unconstrained fragments, it is natural to consider using a domain-independent heuristic to guide the search.

In many domains it is very hard to write deterministic procedural DCK, i.e. DCK that restricts the search space in such a way that solutions can be obtained very efficiently, even using blind search. An example of such a domain is one where plans involve solving an optimization sub-problem. In such cases, procedural DCK will contain open parts (fragments of nondeterministc choice within the DCK), where the designer expects the planner to figure out the best way of completing a sub-task. However, in the absence of domain-specific guidance for these open parts, it is natural to consider using a domain-independent heuristic to guide the search.

In other cases, it is the choice of action arguments, rather than the choice of actions that must be optimized. In particular, fragments of DCK may collectively impose global constraints on action argument choices that need to be enforced by the planner. As such, the planner needs to be *aware* of the procedural control in order to avoid backtracking. By way of illustration, consider a travel planning domain comprising two tasks "buy air ticket" followed by "book hotel". Each DCK fragment restricts the actions that can be used, but leaves the choice of arguments to the planner. Further suppose that budget is limited. We would like our planner to realize that actions used to complete the first part should save enough money to complete the second task. The ability to do such lookahead can be achieved via domain-independent heuristic search.

In the rest of the section we propose three ways in which one can leverage our translated domains using a domain-independent heuristic planner. These three techniques differ predominantly in the operands they consider in computing heuristics.

Direct Use of Translation (*Simple*) As the name suggests, a simple way to provide heuristic guidance while enforcing program awareness is to use our translated domain directly with a domain-independent heuristic planner. In short, take the original domain instance I and control σ, and use the resulting instance I_σ with any heuristic planner.

Unfortunately, when exploiting a relaxed graph to compute heuristics, two issues arise. First, since both the map and $bound$ predicates are relaxed, whatever value is already assigned to a variable, will remain assigned to that variable. This can cause a problem with iterative control. For example, assume program $\sigma_L \overset{\text{def}}{=} \mathbf{while}\,\phi\,\mathbf{do}\,\pi(c\text{-}crate)\,unload(c,T)$, is intended for a domain where crates can be only unloaded sequentially from a truck. While expanding the relaxed plan, as soon as variable c is bound to some value, action $unload$ can only take that value as argument. This leads the heuristic to regard most instances as unsolvable, returning misleading estimates.

The second issue is one of efficiency. Since fluent $state$ is also relaxed, the benefits of the reduced branching factor induced by the programs is lost. This could slow down the computation of the heuristic significantly.

Modified Program Structure (*H-ops*) The *H-ops* approach addresses the two issues potentially affecting the computation of the *Simple* heuristic. It is designed to be used with planners that employ relaxed planning graphs for heuristic computation. The input to the planner in this case is a pair $(I_\sigma, HOps)$, where $I_\sigma = (D_\sigma, P_\sigma)$ is the translated instance, and $HOps$ is an additional set of planning operators. The planner uses the operators in D_σ to generate successor states while searching. However, when computing the heuristic for a state s it uses the operators in $HOps$.

Additionally, function *state* and predicates *bound* and *map* are *not* relaxed. This means that when computing the relaxed graph we actually delete their instances from the relaxed states. As usual, *deletes* are processed before *adds*. The expansion of the graph is stopped if the goal or a fixed point is reached. Finally, a relaxed plan is extracted in the usual way, and its length is reported as the heuristic value. In the computation of the length, auxiliary actions such as tests and noops are ignored.

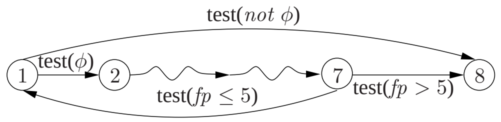


Figure 2: *H-ops* translation of $\mathbf{while}$ loops. While computing the heuristics, pseudo-fluent fp is increased each time no new effect is added into the relaxed state, and it is set to 0 otherwise. The loop can be exited if the last five (7-2) actions performed didn't add any new effect.

The un-relaxing of $state$, $bound$ and map addresses the problem of reflecting the reduced branching factor provided by the control program while computing the heuristics. However, it introduces other problems. Returning to the σ_L program defined above, since $state$ is now un-relaxed, the relaxed graph expansion cannot escape from the loop, because under the relaxed planning semantics, as soon as ϕ is true, it remains true forever. A similar issue occurs with the nondeterministic iteration. Furthermore, we want to avoid state duplication, i.e. having $state$ equal to two different values at the same time in the same relaxed state. This could happen for example while reaching an $\mathbf{if}$ construct whose condition is both true and false at the same time (this can happen because p and $not\text{-}p$ can both be true in a relaxed state).

This issue is addressed by the $HOps$ operators. To avoid staying in the loop forever, the loop will be exited when actions in it are no longer adding effects. Figure 2 provides a graphical representation. An important detail to note is that the loop is not entered when ϕ is not found true in the relaxed state. (The expression $not\ \phi$ should be understood as negation as failure.) Moreover, the pseudo-fluent fp is an internal variable of the planner that acts as a real fluent for the $HOps$. A similar approach is adopted for nodeterministic iterations, whose description we omit here.

Since loops are guaranteed to be exited, the computation of *H-ops* is guaranteed to finish because at some relaxed state the final state of the automaton will be reached. At this point, if the goal is not true, no operators will be possible and a fixed point will be produced immediately.

For $\mathbf{if}$'s, if the condition is both true and false at the same time, the $\mathbf{then}$ part is processed first, followed by the $\mathbf{else}$ part. The objective of this is avoidance of state duplication. However, this new interpretation of the $\mathbf{if}$ introduces a new problem. This problem occurs when, while performing the actions of one of the parts, no action is possible anymore. Intuitively, this could happen because the heuristics has chosen the wrong subprogram to execute actions from. Indeed, if there exists an execution of the program from state s that executes the "then" part of the $\mathbf{if}$, it can happen that, during the computation of the heuristic for s, the "else" part forces some actions to occur that are not possible. Under normal circumstances, the non existence of any possible action produces a fixed point. Because the goal is not reached on such a fixed point, the heuristic regards the goal as unreachable, which could be a wrong estimation.

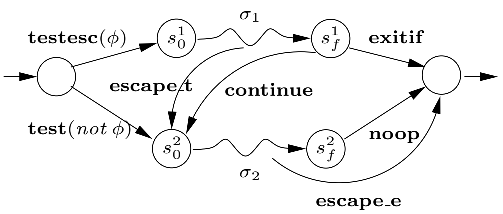


Figure 3: *H-ops* translation for **if - then - else**. Action $\mathbf{testesc}(\phi)$ is possible if condition ϕ is true. If condition $\neg\phi$ is also true in the relaxed state, the $\mathbf{testesc}(\phi)$ dds a fact $escape\_active$ that will enable the execution of **continue** and **escape_t** and **escape_e**. Actions **escape_t** and **escape_e** are possible only when no other actions are possible. This is checked using the pseudo-fluent fp described in Figure 2. Action **exitif** is only possible if $escape\_active$ is true. Both the **noop** and the **escape_e** actions delete the fact $escape\_active$. Nested **if** constructs are handled using a parameterized version of the $escape\_active$ predicate.

To solve this problem, $HOps$ considers new "escape" actions, that are executable only when no more actions are possible. Escapes can be performed only inside "then" or "else" bodies. After executing an escape, the simulation of the program's execution jumps to the else part if the escape occurs in the "then" part, or to the end of the **if** , if the escape occurs in the then part. Figure shows a graphical representation of the $HOps$ generated for the **if** .

A Program-Unaware Approach (*Basic*) Our program-unaware approach (*Basic*) completely ignores the program when computing heuristics. Here, the input to the planner is a pair (I_σ, Ops), where I_σ is the translated instance, and Ops are the *original* domain operators. The Ops operators are used exclusively to compute the heuristic. Hence, *Basic*'s output is not at all influenced by the control program.

Although *Basic* is program unaware, it can sometimes provide good estimates, as we see in the following section. This is especially true when the DCK characterizes a solution that would be naturally found by the planner if no control were used. It is also relatively fast to compute.

Implementation and Experiments

Our implementation[2] takes a PDDL planning instance and a DCK program and generates a new PDDL planning instance. It will also generate appropriate output for the *Basic* and *H-ops* heuristics, which require a different set of operators. Thus, the resulting PDDL instance may contain definitions for operators that are used only for heuristic computation using the `:h-action` keyword, whose syntax is analogous to the PDDL keyword `:action`.

Our planner is a modified version of TLPLAN, which does a best-first search using an FF-style heuristic. It is capable of reading the PDDL with extended operators.

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. 4 shows times for the different heuristic approaches.

		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.

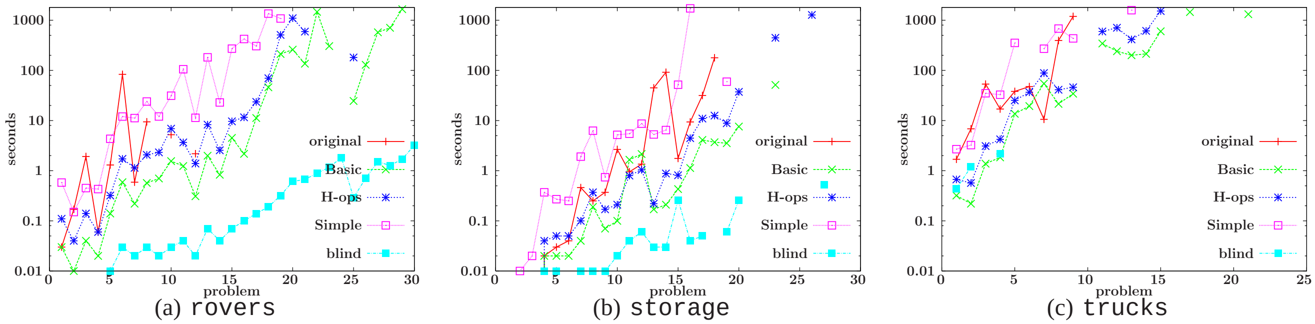
Not surprisingly, our data confirms that DCK helps to improve the performance of the planner, solving more instances across all domains. In some domains (i.e. storage and rovers) blind depth-first cycle-free search is sufficient for solving most of the instances. However, quality of solutions (plan length) 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.

Summary and Related Work

DCK can be used to constrain the set of valid plans and has proven an effective tool in reducing the time required to generate a plan. Nevertheless, many of the planners that exploit it use arguably less natural state-centric DCK specification languages, and their planners use blind search. In this paper we examined the problem of exploiting procedural DCK with SOA planners. Our goal was to specify rich DCK naturally in the form of a program template and to

[2] Available at `www.cs.toronto.edu/kr/systems`



(a) rovers (b) storage (c) trucks

Figure 4: Running times of the three heuristics and the original instance; logarithmic scale; run on an Intel Xeon, 3.6GHz, 2GB RAM

exploit SOA planning techniques to actively plan towards the achievement of this DCK. To this end we made three contributions: provision of a procedural DCK language syntax and semantics; a polynomial-time algorithm to compile DCK and a planning instance into a PDDL2.1 planning instance that could be input to any PDDL2.1-compliant planner; and finally a set of techniques for exploiting domain-independent heuristic search with our translated DCK planning instances. Each contribution is of value in and of itself. The language can be used without the compilation, and the compiled PDDL2.1 instance can be input to any PDDL2.1-compliant SOA planner, not just the domain-independent heuristic search planner that we propose. Our experiments show that procedural DCK improves the performance of SOA planners, and that our heuristics are sometimes key to achieving good performance.

Much of the previous work on DCK in planning has exploited state-centric specification languages. In particular, TLPlan (Bacchus & Kabanza 1998) and TALPlanner (Kvarnström & Doherty 2000) employ declarative, state-centric, temporal languages based on LTL to specify DCK. Such languages define necessary properties of states over fragments of a valid plan. We argue that they could be less natural than our procedural specification language.

Though not described as DCK specification languages there are a number of languages from the agent programming and/or model-based programming communities that are related to procedural control. Among these are EAgle, a goal language designed to also express intentionality (dal Lago, Pistore, & Traverso 2002). Moreover, Golog is a procedural language proposed as an alternative to planning by the cognitive robotics community. It essentially constrains the possible space of actions that could be performed by the programmed agent allowing non-determinism. Our DCK language can be viewed as a version of Golog. Further, languages such as the Reactive Model-Based Programming Language (RMPL) (Kim, Williams, & Abramson 2001) – a procedural language that combines ideas from constraint-based modeling with reactive programming constructs – also share expressive power and goals with procedural DCK. Finally, Hierarchical Task Network (HTN) specification languages such as those used in SHOP (Nau *et al.* 1999) provide domain-dependent hierarchical task decompositions together with partial order constraints, not easily describable in our language.

A focus of our work was to exploit SOA planners and planning techniques with our procedural DCK. In contrast, well-known DCK-enabled planners such as TLPlan and TALPlanner use DCK to prune the search space at each step of the plan and then employ blind depth-first cycle-free search to try to reach the goal. Unfortunately, pruning is only possible for maintenance-style DCK and there is no way to plan towards achieving other types of DCK as there is with the heuristic search techniques proposed here.

Similarly, Golog interpreters, while exploiting procedural DCK, have traditionally employed blind search to instantiate nondeterministic fragments of a Golog program. Most recently, Claßen *et al.* (2007) have proposed to integrate an incremental Golog interpreter with a SOA planner. Their motivation is similar to ours, but there is a subtle difference: they are interested in combining *agent programming* and efficient planning. The integration works by allowing a Golog program to make explicit calls to a SOA planner to achieve particular conditions identified by the user. The actual planning, however, is not controlled in any way. Also, since the Golog interpreter executes the returned plan immediately without further lookahead, backtracking does not extend over the boundary between Golog and the planner. As such, each fragment of nondeterminism within a program is treated independently, so that actions selected locally are not informed by the constraints of later fragments as they are with the approach that we propose. Their work, which focuses on the semantics of ADL in the situation calculus, is hence orthogonal to ours.

Finally, there is related work that compiles DCK into standard planning domains. Baier & McIlraith (2006), Cresswell & Coddington (2004), Edelkamp (2006), and Rintanen (2000), propose to compile different versions of LTL-based DCK into PDDL/ADL planning domains. The main drawback of these approaches is that translating full LTL into ADL/PDDL is worst-case exponential in the size of the control formula whereas our compilation produces an addition to the original PDDL instance that is linear in the size of the DCK program. Son *et al.* (2006) further show how HTN, LTL, and Golog-like DCK can be encoded into planning instances that can be solved using answer set solvers. Nevertheless, they do not provide translations that can be integrated with PDDL-compliant SOA planners, nor do they propose any heuristic approaches to planning with them.

Acknowledgments We are grateful to Yves Lespérance and the ICAPS anonymous reviewers for their feedback. This research was funded by Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Ministry of Research and Innovation (MRI).

References

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

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proc. of the 21st National Conference on Artificial Intelligence (AAAI-06)*, 788–795.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an integration of Golog and planning. In *Proc. of the 20th Int'l Joint Conference on Artificial Intelligence (IJCAI-07)*, 1846–1851.

Cresswell, S., and Coddington, A. M. 2004. Compilation of LTL goal formulas into PDDL. In *Proc. of the 16th European Conference on Artificial Intelligence (ECAI-04)*, 985–986.

dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In *Proc. of AAAI/IAAI*, 447–454.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proc. of the 16th Int'l Conference on Automated Planning and Scheduling (ICAPS-06)*, 374–377.

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

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *Proc. of the 17th Int'l Joint Conference on Artificial Intelligence (IJCAI-01)*, 487–493.

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.

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

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. of the 16th Int'l Joint Conference on Artificial Intelligence (IJCAI-99)*, 968–975.

Rintanen, J. 2000. Incorporation of temporal logic control into plan operators. In Horn, W., ed., *Proc. of the 14th European Conference on Artificial Intelligence (ECAI-00)*, 526–530. Berlin, Germany: IOS Press.

Son, T. C.; Baral, C.; Nam, T. H.; and McIlraith, S. A. 2006. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic* 7(4):613–657.

Proofs

We here provide the proofs of the two theorems, that is, we prove the correctness (sound and completeness) of our translations, and we prove the succinctness of the resulting PDDL planning instance.

Correctness (Theorem 1)

We divide our proof into two parts: a soundness and a completeness result. Throughout the proof, we denote by $I_{\sigma,n,n'}$ the planning instance that results by first invoking $C(\sigma, n, [\,])$ and then following the remaining steps of the compilation, if such a call to C returns (L, L', n') for some L and some L'. Moreover, $I_{\sigma,n,n'}$'s initial state requires *state* $= s_n$ in the initial state, and the goal requires *state* $= s_{n'}$. Note that I_σ, as it is defined in the compilation section, corresponds to $I_{\sigma,0,n_{\text{final}}}$.

We start by proving a few intermediate results.

Lemma 1. Let σ be a program, let I be a planning instance with initial state $Init$, and let $I_{\sigma,n,n'}$ be the instance generated by the compilation with the usual operator lists L and L'. Assume σ_1 is a subprogram of σ, such that $C(\sigma_1, n_1, E_1)$ was invoked during the top-level compilation, returning (L_1, L_1', n_1'). Finally, let $\vec{a} = a_0 a_1 \cdots a_p$ be a plan for $I_{\sigma,n,n'}$. If a_j is an action such that $Succ(Init, a_0 \cdots a_j, s')$ and $s' \models state = s_k$, for some s' and some k such that $n_1 \leq k < n_1'$, then a_j is an instance of an operator in $L_1 \cdot L_1'$.

Proof. Assume that a_j is an instance of an operator in $L \cdot L'$ but not in $L_1 \cdot L_1'$. Since all operators that where generated by C while compiling a subprogram of σ' are also in $L_1 \cdot L_1'$, there must be another subprogram of σ, say σ'', that is not a subprogram of σ' such that the compilation of σ'' generated an operator not in $L_1 \cdot L_1'$ that is possible when $state = s_k$. The recursive definition of the C operator does not admit this. If σ' and σ'' are two non-overlapping subprograms, the new preconditions that restrict the $state$ variable are defined in such a way that they can never overlap for the same value of $state$. □

The following lemma intuitively states that a plan for a program contains sub-plans for all executed sub-programs.

Lemma 2. Let σ be a program with no program variables. Let I be a planning instance with initial state $Init$, and let $I_{\sigma,n,n'}$ be the instance generated by the compilation. Assume σ_1 is a subprogram of σ, such that $C(\sigma_1, n_1, [])$ was invoked during the compilation of σ, returning (L_1, L_1', n_1'). Furthermore, let $\vec{a} = a_0 a_1 \cdots a_p$ be a plan for $I_{\sigma,n,n'}$ such that, when executed in $Init$, generates the sequence of states $s_1 s_2 \cdots s_p$. Finally, assume there exist two integers i and j, such $0 \leq i \leq j \leq p$ and such that $s_i \models state = s_{n_1}$, $s_j \models state = s_{n_1'}$ and for all r such that $i < r < j$, $s_r \models state = s_u$ with $n_1 \leq u < n_1'$.

Then, for any planning instance I', which is just like I except that the initial is s_i and the goal is empty, we have that

$\vec{a}' = a_i a_{i+1} \cdots a_j$ is a plan for I'_{σ_1,n_1,n'_1}, which is the instance that results from compiling σ' by calling $C(\sigma_1, n_1, [])$ on I'.

Proof. By Lemma 1, actions in $a_i a_{i+1} \cdots a_j$ are instances of operators in I'_{σ',n_1,n'_1}. Moreover, since the initial state of I'_{σ',n_1,n'_1} is s_i, the sequence $\vec{a}'$ is also executable on I'_{σ',n_1,n'_1}, as while executing $\vec{a}'$ on I'_{σ',n_1,n'_1} the planning states traversed are identical to those states traversed while performing the subsequence $\vec{a}'$ of $\vec{a}$ in $I_{\sigma,n,n'}$. Finally, after performing $\vec{a}'$, we reach a state where $state = s_{n'_1}$, and hence $\vec{a}'$ is a plan for I'_{σ',n_1,n'_1}. □

We are now ready to prove the soundness part of the theorem.

Proof of Theorem 1:

⇒ (Soundness):

Given a plan $\vec{a}$ for instance $I_\sigma = (D_\sigma, P_\sigma)$, show that Filter$(\vec{a}, D)$ is a plan for $I = (D, P)$ under the control of σ.

We prove this in several steps.

Lemma 3. Let σ be a program, $I = (D, P)$ a planning instance, and $\vec{a}$ a plan for planning instance $I_\sigma = (D_\sigma, P_\sigma)$. Then *Filter*$(\vec{a}, D)$ is a plan for I.

Proof: Note that the preconditions of actions in D_σ are strictly more restrictive than their counterparts in D, as the original preconditions are conjoined with additional ones. Thus, whenever an action a of D_σ is executable in a state s and a is a domain action as opposed to any of the newly introduces bookkeeping actions, then the corresponding action a' in D is executable in s as well. Further, note that the additional effects of a in D_σ compared to a' in D only affect the new bookkeeping predicates and functions (bound, map, and state). Therefore, since the initial and goal state of I_σ differ from their counterparts in I only in terms of these bookkeeping predicates and functions, *Filter*$(\vec{a}, D)$ achieves the goal of P and thus *Filter*$(\vec{a}, D)$ is a plan for $I = (D, P)$. □

To prove that the action sequence *Filter*$(\vec{a}, D)$ is also a plan under the control of σ, we have to show that the automaton $A_{\sigma,I}$ accepts it. We do this by induction over the structure of the program σ.

Lemma 4. Let σ be a program without the $\pi(x\text{-}t)$ construct, $I = (D, P)$ a planning instance, and $\vec{a}$ a plan for planning instance $I_{\sigma,n,n'} = (D_\sigma, P_\sigma)$. Then *Filter*$(\vec{a}, D)$ is an execution of σ in I.

Proof: Throughout this proof we will refer to the compilation result $C(\sigma, n, E) = (L, L', n')$ used to construct $I_{\sigma,n,n'}$. Since there are no $\pi(x\text{-}t)$ constructs, we can assume that the E argument of C is always empty and can ignore any *bound* and *map* preconditions and effects upon these predicates for now. The program does not contain any program variables.

The proof proceeds by induction over the structure of σ as follows:

$\sigma = nil$**:** By definition of C, both L and L' are empty, and therefore no operators are included in D_σ. Thus the plan must be empty. The empty sequence is accepted by $A_{\sigma,I}$, because $[nil, s]$ is a final state.

$\sigma = a, a \in \mathcal{A}$**:** By definition of the translation, the only operator in D_σ is action a. Thus, the only potentially possible action in any state where *state* $= s_n$ is a. Since the goal, by construction, requires $state = s_{n+1}$, $\vec{a}$ must be $[a]$, and a must be possible in the initial state. From Eq. 4 we know that $[a]$ is accepted by $A_{\sigma,I}$.

$\sigma = \phi?$**:** By definition of the translation, the only operator in D_σ is $test\_n\_n_1$, which is potentially possible in any state where *state* $= s_n$. Since the goal, by construction, requires $state = s_{n+1}$, $\vec{a} = [test\_n\_n_1]$, and since this is a plan, we know that its preconditions are satisfied in the initial state, hence $Init \models \phi$ and thus $A_{\sigma,I}$ accepts $[\,] =$ *Filter*$([test\_n\_n_1], D)$ by Eq. 6.

These are the base cases. Now for the induction steps:

$\sigma = (\sigma_1; \sigma_2)$**:** Assume that $C(\sigma_1, n, E)$ and $C(\sigma_2, n_1, E)$ where invoked while compiling σ, for some n_1.

By construction of I, any plan $\vec{a} = a_0 a_1 \cdots a_n$ for I_σ can be partitioned into two parts $\vec{a}_1$ and $\vec{a}_2$ such that $\vec{a} = \vec{a}_1\vec{a}_2$, and such that $state = s_{n_1}$ in the state s' that results after performing $\vec{a}_1$ over I_σ.

Let us define $I' = I$, then, by Lemma 2, $\vec{a_1}$ is a plan for I'_{n,n_1}. Moreover, let us define I'' as a planning instance whose initial state is s' but with no information about the state. By Lemma 2, $\vec{a}_2$ is a plan for I''_{n_1,n_2}.

By induction hypothesis we know that the automaton $A_{\sigma_1,I'}$ accepts any plan for I'_{σ_1,n,n_1} for I'. Analogously, $A_{\sigma_2,I''}$ accepts any plan for I''_{σ_2,n_1,n_2}.

It now follows from the definition of δ (Eq. 7) and a similar argument as in the proof for Lemma 2 that $\vec{a}_1\vec{a}_2$ is also accepted by $A_{\sigma,I}$.

$\sigma = (\sigma_1|\sigma_2)$**:** From the definition of C we know that any plan for I_{σ,n,n_2+1} must start with either *noop*_n_$(n{+}1)$ or *noop*_n_$(n_1{+}1)$. After that, by induction hypothesis and Lemma 2, the only possible action sequences are those that are plans for $I_{\sigma_1,n+1,n_1}$ or I_{σ_2,n_1+1,n_2}. These sequences are accepted by their respective automata $A_{\sigma_1,I}$ and $A_{\sigma_2,I}$. By its definition, the language accepted by $A_{\sigma,I}$ is the union of the two languages of these automata, and the additional *noop* actions are filtered out.

$\sigma = \textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2$**:** From the definition of C for this case we know that any plan for I_{σ,n,n_3} must start with either $test\_n\_n'$ or $test\_n\_n''$, with $n' = n + 1$ and $n'' = n_1 + 1$, depending on whether ϕ holds in the initial state. After that, by induction hypothesis and Lemma 2, the only possible action sequences are those that are plans for I_{σ_1,n',n_1} or I_{σ,n'',n_2}. These sequences are accepted by their respective automata $A_{\sigma_1,I}$ and $A_{\sigma_2,I}$, by induction hypothesis. By its definition, the language accepted by $A_{\sigma,I}$ is the one accepted by the former if ϕ holds in the initial state, and otherwise the language of the latter. The *noop* and $test$ actions are filtered out.

$\sigma = \textbf{while}\,\phi\,\textbf{do}\,\sigma'$**:** From the definition of C for this case we know that any plan for $I_{\sigma,n,n'}$, with $n' = n_1 + 1$, must start with either $test\_n\_n''$, with $n'' = n + 1$, if ϕ holds in the initial state, or $test\_n\_n'$, otherwise. In the former case, by Lemma 2, the only action sequence possible will start with a plan for I_{σ',n'',n_1} which, by induction hypothesis, is accepted by the automaton $A_{\sigma',I}$, followed by *noop_n_1_n* which, inductively, implies that it is followed by a plan for $I_{\sigma,n,n'}$. By definition of $A_{\sigma,I}$, in the case where $s \models \phi$, it accepts sequences which begin with sequences accepted by I_{σ',n'',n_1}, followed by any other sequence accepted by $A_{\sigma,I}$. Otherwise, if ϕ does not hold initially, $test\_n\_n'$, which is possible when ϕ doesn't hold, leads to a final state of $I_{\sigma,n,n'}$ and the filtered plan is empty. Analogously $A_{\sigma,I}$ accepts the empty language if ϕ doesn't hold. Thus, $A_{\sigma,I}$ accepts any plan for $I_{\sigma,n,n'}$.

$\sigma = \sigma'^*$**:** From the definition of C for this case and Lemma 1 we know that any plan for I_{σ,n,n_2}, must either consist of *noop*(n, n_2), which after filtering results in the empty plan which is trivially accepted by $A_{\sigma,I}$, or a plan for I_{σ',n,n_1} followed by *noop*(n_1, n) and, recursively, any other plan for $I_{\sigma,n,n'}$. In the latter case, by induction hypothesis, any such plan is accepted by the sequence of automaton $A_{\sigma',I}$ and $A_{\sigma,I}$, which precisely meets the definition of $A_{\sigma,I}$.

□

Now for the case with program variables.

Lemma 5. Let σ be a program, possibly with $\pi(x\text{-}t)$ constructs, $I = (D, P)$ a planning instance, and $\vec{a}$ a plan for planning instance $I_\sigma = (D_\sigma, P_\sigma)$. Then *Filter*$(\vec{a}, D)$ is an execution of σ in I.

Proof: The proof proceeds by induction over the number of $\pi(x\text{-}t)$ constructs in σ.

If σ is program variable free ($\pi(x\text{-}t)$ does not occur), then, trivially by Lemma 4 the proposition holds.

Assume $\sigma = \pi(x\text{-}t)\sigma'$, and let $\vec{a}' = a_0a_1 \cdots a_n$ such that $\vec{a}' \cdot [free_{n_1}(x)]$ is a plan for I_σ. First, we prove that there exists an $o \in Objs$ such that $a_0a_1 \cdots a_n$ is a plan for $I_{\sigma'|x/o}$.

Let us assume that the state trajectory generated when performing $a_0a_1 \cdots a_n$ in $Init$ is $s_0s_1 \cdots s_n$. Observe the actions in the plan cannot delete $map(x)$ or delete $bound(x, o)$. Furthermore, if $bound(x, o)$ is true in a certain state, no action will add $bound(x, o')$ for any o' different from o. Hence, there exists a j ($0 \le j \le n$) such that

- $s_i \not\models map(x)$ and $s_i \not\models bound(x, o)$, for any $o \in Objs$ and any $i < j$, and
- $s_i \models bound(x)$ and $s_i \models map(x, v)$ for all i s.t. $j \le i \le n$ and some $v \in Objs$.

We claim that $a_0a_1 \cdots a_n$ is a plan for $I_{\sigma'|x/v}$. The proof for the claim is split in two parts: (a) we prove that the sequence $a_0a_1 \cdots a_n$ is legally executable in $I_{\sigma'|x/v}$, then (b) we prove that it reaches the goal.

For proving (a), note that the only difference between I_σ and $I_{\sigma'|x/v}$ are the preconditions of some of its operators. For each occurrence of $bound(x) \rightarrow map(x, x_i)$ (for some x_i) in an operator in I_σ there is an occurrence of $x_i = v$ in $I_{\sigma'|x/v}$. It is easy to see that the preconditions of the first $j - 1$ actions of the sequence, $a_0a_1 \cdots a_{j-2}$, are satisfied in $I_{\sigma'|x/v}$. Indeed, note that because $bound(x)$ is not added by these actions in I_σ, by the definition of C, it means that the subformula of the precondition of the operator of I_σ that evaluated to true at that point is identical to that of the respective operator in $I_{\sigma'|x/v}$. Now let's focus on action a_{j-1}. This action ***adds*** $bound(x)$ and $map(x, v)$. By the construction of C this means that the precondition evaluated $bound(x) \rightarrow map(x, x_i)$ to be true in the state were a_{j-1} was performed (this happens because $bound(x)$ is false). Because after performing a_{j-1}, $map(x, v)$ is added, it means that the parameter x_i of the operator took value v, while satisfying all additional preconditions. On the other hand, in $I_{\sigma'|x/v}$, the condition to be checked by the respective operator is instead $x_i = v$, which we know can be made true while satisfying additional preconditions of the operator, because a_{j-1} was executable in I_σ. For the remaining part of the sequence, $a_ja_{j+1} \cdots a_n$ the proof is analogous. When performed in I_σ, some of these actions will evaluate $bound(x) \rightarrow map(x, x_i)$ to true, with the side effect of making the parameter x_i equal to v. On the other hand, in $I_{\sigma'|x/v}$, the same effect is achieved but by the explicit $x_i = v$ in the precondition. Hence, the precondition in $I_{\sigma'|x/v}$ will also be satisfied.

The proof for (b) is straightforward. Since the goal does not mention any bookkeeping predicates, the sequence $\vec{a}'$ produces the same state in $I_{\sigma'|x/v}$ as $\vec{a}' \cdot [free_{n_1}(x)]$ in I_σ.

The proof now follows from Lemma 4. □

Proof of Theorem 1 (continued):
⇐ (Completeness):
Given a plan $\vec{a}$ for I under the control of σ, show that there exists a plan $\vec{a}'$ for I_σ, such that $\vec{a} =$ Filter$(\vec{a}', D)$.

The proof again proceeds by induction over the structure of the program σ, and again we first show the case for programs without $\pi(x\text{-}t)$ constructs, i.e. without program variables.

Lemma 6. Let σ be a program without the $\pi(x\text{-}t)$ construct, $I = (D, P)$ a planning instance, and $\vec{a}$ a plan for I under the control of σ, then there exists a plan $\vec{a}'$ for $I_{\sigma,n,n'}$ such that $\vec{a} =$ *Filter*$(\vec{a}', D)$.

Proof: We will again refer to the compilation result $C(\sigma, n, E) = (L, L', n')$ used to construct $I_{\sigma,n,n'}$, and occasionally also to variables occurring in the particular compilation case considered in the induction proof. Again, since there are no $\pi(x\text{-}t)$ constructs, we can assume that the E argument of C is always empty and can ignore any *bound* and *map* preconditions and effects upon these predicates for now. The program does not contain any program variables.

By assumption we know that $A_{\sigma,I}$ accepts the plan $\vec{a}$. The induction over the structure of σ is as follows:

$\sigma = nil$**:** $A_{\sigma,I}$ only accepts the empty language, since there are no transitions defined for the nil program, but $[nil, s]$ is an accepting state for any state s over I. Thus $\vec{a} = [\,]$. Since both initial an goal state of $I_{\sigma,n,n'}$ only require *state* $= s_n$ on top of the original initial and goal state of I, and $n' = n$, $\vec{a}' = [\,] = \vec{a}$ is also a plan for $I_{\sigma,n,n'}$ and $\vec{a} =$ *Filter*$(\vec{a}', D)$.

$\sigma = a, a \in \mathcal{A}$**:** In this case $\vec{a} = [a]$. Since in the compilation E is empty, the preconditions of the operator corresponding to a in $I_{\sigma,n,n'}$ are the same as those for a in I, except that *state* $= s_n$ has to hold. This condition is easily fulfilled by the fact that the initial state of $I_{\sigma,n,n'}$ states just this. Also, a goal state of $I_{\sigma,n,n'}$ is reached after executing a in $I_{\sigma,n,n'}$, since the new operator, by definition of C has *state* $= s_{n+1}$ as an effect, which, by construction, is the only additional requirement in the goal state of $I_{\sigma,n,n'}$ compared to I. Thus $\vec{a}$ is a plan for $I_{\sigma,n,n'}$, and trivially $\vec{a} = \textit{Filter}(\vec{a}, D)$.

- $\sigma = \phi?$**:** Again, the plan has to be the empty sequence, since this is the only one accepted by $A_{\sigma,I}$. Also, by definition of $A_{\sigma,I}$, the initial state $Init$ of I satisfies ϕ. Let $\vec{a}' = [test\_n\_n']$. This is a plan for $I_{\sigma,n,n'}$, because by its construction in the definition of $Ctest$ its precondition is *state* $= s_n \wedge \phi$. This is satisfied since the initial state of $I_{\sigma,n,n'}$ is like that of I plus the assertion that *state* $= s_n$. Since ϕ cannot mention the new special fluent *state* its truth value does not differ between the initial state of $I_{\sigma,n,n'}$ and that of I itself. Further, $test\_n\_n'$ sets *state* $= s_{n'}$ as its only effect (E is empty), thus satisfying the goal of $I_{\sigma,n,n'}$. Finally, $\vec{a} = [\,] = \textit{Filter}([test\_n\_n'], D)$.

These are the base cases. Now for the induction steps:

- $\sigma = (\sigma_1; \sigma_2)$**:** We start this case by stating an intermediate result where we use $\delta(\sigma, \vec{a})$ to denote the repeated transition of δ over the actions of the sequence $\vec{a}$.

  Claim: If $\vec{a}$ is accepted by $A_{\sigma,I}$, then $\vec{a}$ can be decomposed into two parts $\vec{a}_1$ and $\vec{a}_2$, such that $\vec{a} = \vec{a}_1\vec{a}_2$, and such that $[nil; \sigma_2, s'] \in \delta([\sigma_1; \sigma_2, Init], \vec{a}_1)$, for some s' and such that $[nil, s''] \in \delta([\sigma_2, s'], \vec{a}_2)$. Intuitively, this means that the automaton's state $[nil; \sigma_2, s']$ is part of an accepting path of states for $\vec{a}$. *Proof.* Straightforward (but lengthy) by induction on the structure of σ_1.

  Let us assume that $\vec{a} = \vec{a}_1\vec{a}_2$, for $\vec{a}_1$ and $\vec{a}_2$ as defined above. Furthermore let us define I^1 as an instance just like I except that its goal is to get to state s' (as defined above). Moreover, we define I^2 to be just like I but such that its initial state is s'. Observe now that $\vec{a}_1$ and $\vec{a}_2$ are clearly accepted by A_{σ_1,I^1} and A_{σ_2,I^2}. Indeed, this follows straightforwardly from the claim and the fact that the transition function for A_{σ_1,I^1} and A_{σ_2,I^2} are subsets of the transition function for $A_{\sigma,I}$.

  By induction hypothesis, there are plans $\vec{a}'_1, \vec{a}'_2$ for $I^1_{\sigma_1,n_1,n'_1}$ and $I^2_{\sigma_2,n_2,n'_2}$ for any two integers n_1, n_2, such that $\vec{a}_1 = \textit{Filter}(\vec{a}'_1, D)$ and $\vec{a}_2 = \textit{Filter}(\vec{a}'_2, D)$. Choosing $n_2 = n'_1$ as defined by the compilation of σ_1 with parameter $n = n_1$, we get that the initial state of $I^2_{\sigma_2,n_2,n'_2}$ is a goal state of $I^1_{\sigma_1,n_1,n'_1}$ and thus $\vec{a}' = \vec{a}'_1 \cdot \vec{a}'_2$ is a plan for $I_{\sigma,n,n'}$. Since the concatenation does not introduce any new actions we get $\vec{a} = \textit{Filter}(\vec{a}', D)$.

- $\sigma = (\sigma_1 | \sigma_2)$**:** By definition, $A_{\sigma,I}$ accepts the union of the sets of plans for σ_1 and σ_2, i.e. $\vec{a}$ is accepted by either $A_{\sigma_1,I}$ or $A_{\sigma_2,I}$.

  Assume it is a plan under the control of σ_1 (i.e., it is accepted by $A_{\sigma_1,I}$). By induction hypothesis there is a plan $\vec{a}_1$ for I_{σ_1,n_1,n'_1} for any integer n_1, such that $\vec{a} = \textit{Filter}(\vec{a}'_1, D)$. Then $\vec{a}' = [\textit{noop\_n\_}(n+1)] \cdot \vec{a}'_1 \cdot [\textit{noop\_}n_1\textit{\_}(n_2{+}1)]$ is a plan for I_{σ,n,n_2+1}, where n_2 is defined in the compilation, and since the *noop* actions are filtered again $\vec{a} = \textit{Filter}(\vec{a}', D)$. The case when $\vec{a}$ is a plan under the control of σ_2 is analogous with the plan $\vec{a}' = [\textit{noop\_n\_}(n_1{+}1)] \cdot \vec{a}'_2 \cdot [\textit{noop\_}n_2\textit{\_}(n_2{+}1)]$, n_1, n_2 are defined by the compilation.

- $\sigma = \textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2$**:** Depending on whether or not $Init \models \phi$, $\vec{a}$ is a plan under the control of σ_1 or σ_2, i.e. it is either accepted by $A_{\sigma_1,I}$ or $A_{\sigma_2,I}$. Assume $Init \models \phi$. Then, $\vec{a}_1$ is accepted by $A_{\sigma_1,I}$, and by induction hypothesis, there is a plan $\vec{a}'_1$ for I_{σ_1,n_1,n'_1} for any integer n_1 s.t. $\vec{a} = \textit{Filter}(\vec{a}'_1, D)$. Then $\vec{a}' = [test\_n\_(n+1)] \cdot \vec{a}'_1 \cdot [\textit{noop\_}n_1\textit{\_}n_3]$ is a plan for $I_{\sigma,n,n'}$ and by definition of *Filter* we have $\vec{a} = \textit{Filter}(\vec{a}', D)$. Analogously when $Init \not\models \phi$, $\vec{a}' = [test\_n\_(n_1{+}1)] \cdot \vec{a}'_2 \cdot [\textit{noop\_}n_2\textit{\_}n_3]$ is a plan for $I_{\sigma,n,n'}$ and again $\vec{a} = \textit{Filter}(\vec{a}', D)$.

- $\sigma = \textbf{while}\,\phi\,\textbf{do}\,\sigma'$**:** The induction step for this case is itself by induction. We refer to this induction as "inner induction", and to the other as "outer induction". The inner induction is on the length of the action sequence $\vec{a}$.

  As our inner base case, assume that $Init \not\models \phi$, then $\vec{a} = [\,]$ ($|\vec{a}| = 0$). Then $[test\_n\_n']$ is a plan for $I_{\sigma,n,n'}$ for any integer n, because by construction the precondition for this test action is $\neg\phi \wedge$ *state* $= s_n$, and its effect asserts *state* $= s_{n'}$. Also $[\,] = \textit{Filter}([test\_n\_n'], D)$. This concludes the proof for the inner base case.

  Now, as our inner induction hypothesis, we assume the theorem holds for all sequences of action whose length is strictly less that k. Now assume $|\vec{a}| = k$. In this case, we have that $Init \models \phi$, and then $\vec{a} = \vec{a}_{\sigma'} \cdot \vec{a}''$ is a plan for $I_{\sigma,n,n'}$, where $\vec{a}_{\sigma'}$ is a sequence accepted by $A_{\sigma',I}$, and $\vec{a}''$ is accepted by $A_{\sigma,I'}$, where I' is like I except that the initial state is the state reached after executing $\vec{a}_{\sigma'}$ in $Init$. Then, by outer induction hypothesis there is a plan $\vec{a}'_{\sigma'}$ for I_{σ',n_3,n'_3} for any integer n_3, s.t. $\vec{a}_{\sigma'} = \textit{Filter}(\vec{a}'_{\sigma'}, D)$, and by inner induction hypothesis there is a plan $\vec{a}'''$ for I'_{σ,n_2,n'_2} for any integer n_2 s.t. $\vec{a}'' = \textit{Filter}(\vec{a}''', D)$. Choosing $n_2 = n$ and $n_3 = n + 1$ we get that $\vec{a}' = [test\_n\_(n{+}1)] \cdot \vec{a}'_{\sigma'} \cdot [\textit{noop\_}n_1\textit{\_}n] \cdot \vec{a}'''$ is a plan for $I_{\sigma,n,n'}$, where n_1 is defined by the compilation for σ. Finally, again, $\vec{a} = \textit{Filter}(\vec{a}', D)$.

- $\sigma = \sigma'^*$**:** We again require an inner induction on the length of $\vec{a}$. Assume that $\vec{a} = [\,]$, then $[\textit{noop\_n\_}n']$ is a plan for $I_{\sigma,n,n'}$ and trivially $\vec{a} = \textit{Filter}([\textit{noop\_n\_}n'], D)$. This concludes the proof for the base case of the inner induction. Assume now for the inner induction case that the theorem holds for all sequences of length less than k, where $|\vec{a}| = k$. In this case, $\vec{a} = \vec{a}_1 \cdot \vec{a}_2$ where $\vec{a}_1$ is accepted by $A_{\sigma',I}$ and $\vec{a}_2$ is accepted by $A_{\sigma,I'}$ where I' is like I except that the initial state is the state reached after executing $\vec{a}_{\sigma'}$ in $Init$. Then, by outer induction hypothesis there is a plan $\vec{a}'_1$ for I_{σ',n_3,n'_3} for any integer n_3 s.t. $\vec{a}_1 = \textit{Filter}(\vec{a}'_1, D)$, and by inner induction hypothesis there is a plan $\vec{a}'_2$ for I'_{σ,n_2,n'_2} for any integer n_2 s.t. $\vec{a}_2 = \textit{Filter}(\vec{a}'_2, D)$. Choosing both $n_3 = n$ and $n_2 = n$ we get that $\vec{a}' = \vec{a}'_1 \cdot [\textit{noop\_}n_1\textit{\_}n] \cdot \vec{a}'_2$ is a plan for $I_{\sigma,n,n'}$, where n_1 is defined by the compilation. Again, by the two induction hypotheses and the fact that $noop\_n_1\_n$ is filtered out, $\vec{a} = \mathit{Filter}(\vec{a}', D)$.

□

Now for the case with program variables.

Lemma 7. Let σ be a program over a planning instance $I = (D, P)$ (possibly containing $\pi(x\text{-}t)$ constructs), and $\vec{a}$ a plan for I under the control of σ, then there exists a plan $\vec{a}'$ for $I_{\sigma,n,n'}$ such that $\vec{a} = \mathit{Filter}(\vec{a}', D)$.

Proof: The proof proceeds by induction over the number of $\pi(x\text{-}t)$ constructs occurring in σ. The base case, where this number is zero, is given by Lemma 6.

Otherwise, assume $\sigma = \pi(x\text{-}t, \sigma')$ for some arbitrary other program σ' over I. By the definition of $A_{\sigma,I}$, $\vec{a}$ is accepted by some automaton $A_{\sigma|_{x/o},I}$ where in σ all occurrences of x are replaced by some (but in all occurrences the same) o such that $(o, t) \in \tau_D \cup \tau_P$. We show that (i) $\vec{a}' = \vec{a} \cdot [free\_n_1(x)]$ is a plan for $I_{\sigma,n,n'}$ for any integer n, where n_1 is defined in the compilation of σ using n as the integer parameter. We further need to show that (ii) in a state s' reached after performing $\vec{a}$ in any state s that satisfies $\neg\mathit{bound}(x) \wedge \neg(\exists y).\mathit{map}(x, y)$, we again get $s' \models \neg\mathit{bound}(x) \wedge \neg(\exists y).\mathit{map}(x, y)$. Obviously, the initial state $Init$ has this property for all program variables occurring in σ.

(i) By assumption $\vec{a}$ is accepted by $A_{\sigma|_{x/o},I}$ for some o, i.e. after replacing all occurrences of x in σ with o, and is a plan for I. By induction hypothesis and Lemma 6 there exists a plan $\vec{a}'_1$ for $I_{\sigma|_{x/o},n,n'}$ for any integer n such that $\vec{a} = \mathit{Filter}(\vec{a}'_1, D)$. We show that this is also a plan for $I_{\sigma,n,n'}$ after minor modifications to the occurring test actions, and which in particular do not result in a different result when applying *Filter*. Compile σ as defined using $C(\sigma, n, [\,]) = (L, L', n')$. For any test action occurring in $\vec{a}'_1$ whose corresponding operator definition in L has x as a formal parameter, add o as an additional argument at the position where x appears in the operator definition, creating a new sequence $\vec{a}'_2$. We show that this sequence is a plan for $I_{\sigma,n,n'}$: Let a_1 be the first action in $\vec{a}'_2$ whose corresponding operator definition in L has x as a formal parameter. The corresponding actual parameter is o. Then, since in the initial state s of $I_{\sigma,n,n'}$ we have that $s \models \neg\mathit{bound}(x) \wedge \neg(\exists y).\mathit{map}(x, y)$, s satisfies the preconditions of a_1, because the only preconditions on top of those defined in $I_{\sigma|_{x/o},n,n'}$ are $\mathit{bound}(x) \rightarrow \mathit{map}(x, o)$. The action will further have as an effect $\mathit{bound}(x)$ and $\mathit{map}(x, o)$. Hence, all following actions a_k in $\vec{a}'_2$ whose corresponding operator in L has x as a formal parameter, will also be possible and have the same effects as in $I_{\sigma|_{x/o},n,n'}$ (by construction of $\sigma|_{x/o}$), because also they have o as actual parameter, and since $\vec{a}'_2$ cannot mention any action $free\_n_i(x)$, for any i, we have for all states s'' visited later on during the execution of $\vec{a}'_2$ that $s'' \models \mathit{bound}(x) \wedge \mathit{map}(x, o)$ which entails the preconditions of a_k in $I_{\sigma,n,n'}$. Since further only the truth value of *bound* and *map* are changed compared to the effects in $I_{\sigma|_{x/o},n,n'}$, the goal, which by construction doesn't mention either of these predicates, is reached at the end. Hence, $\vec{a}'_2$ is a plan for $I_{\sigma,n,n'}$. Also $\vec{a} = \mathit{Filter}(\vec{a}'_2 \cdot [free\_n_1(x)], D)$.

(ii) Clearly, since for any n_i, $free\_n_i(x)$ has $\neg\mathit{bound}(x) \wedge (\forall y).\neg\mathit{map}(x, y)$ as an effect, any state s' reached after executing $\vec{a}'_2 \cdot [free\_n_1(x)]$ in any other state satisfies this. □

Theorem 1 then follows directly from Lemmata 5 and 7 for $n = 0$ and n_{final} as defined by the compilation $C(\sigma, 0, [\,]) = (L, L', n_{final})$.

Succinctness (Theorem 2)

Proof of Theorem 2:

The compilation of each programming construct, as defined by C, introduces a constant number of new operators into I_σ or extends the definition of one of the operators of I with a constant number of additional preconditions and effects. In all cases, the size of the new preconditions and effects is bounded by a constant factor in the number of elements of E. From the definition of C for π it follows that the maximal length of E occurring during the compilation of σ is exactly the number of nested π constructs, k. Hence, if the program has size n, then there are no more than n programming constructs. Since also each construct is considered exactly once by C, there can be no more than n operators in I_σ, each of size $O(k)$. Hence, overall I_σ has size $O(k \cdot n)$. □