1

# Golog-Style Search Control for Planning

JORGE A. BAIER, CHRISTIAN FRITZ, SHEILA A. MCILRAITH

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 the Planning Domain Definition Language (PDDL), specifically PDDL2.1. PDDL is the de facto standard input language for state-of-the-art automated planning systems. Our compilation enables any PDDL2.1-compatible planner to exploit procedural DCK without the need for special-purpose computational machinery. The contributions of this paper are threefold. First, inspired by the logic programming language GOLOG, we propose a PDDL-based semantics for an Algol-like procedural language that can be used to specify procedural DCK as a program. 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.

## Foreword (by Sheila McIlraith)

When Hector, Ray Reiter, and colleagues at the University of Toronto first introduced the GOLOG logic programming language, it was viewed as a means of specifying high level control for robots and software agents, as well as for industrial processes and discrete event simulations. Part of GOLOG's elegance was its situation calculus semantics which enabled GOLOG programs to reason about the complex dynamics of the world, as specified in situation calculus. From an outsider's perspective, GOLOG was inextricably tied to the situation calculus and to the simple interpreter that allowed it to reason over the situation calculus specification of dynamical systems.

Our motivation with this work was to bring the same basic GOLOG philosopy to bear on dynamical systems described in the less expressive Planning Domain Definition Language (PDDL) [McDermott 1998], using a GOLOG-style language to specify programs that either imparted procedural search control on the plan generation process or that specified what could be viewed as a complex, temporally extended plan objective. In contrast to GOLOG, the semantics of this GOLOG-style language was provided in PDDL. More importantly, rather than use a GOLOG interpreter to extract a plan, the mechanism by which we specified our PDDL semantics enabled us to *compile away* our GOLOG-like programs and to exploit highly effective state-of-the-art planning technology to synthesize plans that adhered to the constraints of the GOLOG-like programs.

Our initial work in this vein is reflected in the article that follows, a version of which originally appeared under the title "Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners" in the Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS2007). A follow-on paper at KR2008 examined related issues, extending the expressivity of our GOLOG-like language to include a number of ConGolog constructs.

The article that follows was written for a planning audience and as such extols the virtues and shortcomings of the work from a plan generation perspective. Indeed, an important benefit of our work is that it provides a means for state-of-the-art planners to exploit procedural domain control knowledge (DCK) specified in a GOLOG-like language. This benefit is also shared by GOLOG researchers in so far as the work provides a computationally effective means of synthesizing a class of GOLOG program executions with respect to a restricted class of dynamical systems. However, the benefits of this work do not stop there. The technique used to specify the semantics, here in PDDL, in the KR2008 paper in the situation calculus, is interesting because it avoids the need for reification – something Hector particularly liked about the work. Further, from our understanding of the relationship between PDDL and situation calculus and between GOLOG and other DCK formalisms such as Hierarchical Task Networks (HTNs) (elaborated upon by Alfredo Gabaldon in this volume), this body of work enables not only GOLOG, but also HTN-like DCK, or even hybrid GOLOG-HTN DCK to be compiled away in a similar fashion and for plans and program executions to be synthesized using efficient automated planning technology.

## 1 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 planners that do not exploit DCK. Indeed most planners that exploit DCK, such as TLPLAN [Bacchus and Kabanza 1998] or TALPLANNER [Kvarnström and 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 state-of-the-art planning techniques and existing state-of-the-art 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 2006 International Planning Competition, where the goal is to deliver packages between certain locations using a limited capacity truck with restricted access. Once 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, Cao, Lotem, and Muñoz-Avila 1999] or GOLOG [Levesque, Reiter, Lespérance, Lin, and Scherl 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 and Long [2003].

With a well-defined procedural DCK language in hand, we examine how to use state-of-the-art planning techniques together with DCK. Of course, most state-of-the-art planners are unable to exploit DCK. As such, we present an algorithm that translates a PDDL2.1-specified Action Description Language (ADL) [Pednault 1989] 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 state-of-the-art planners may not exploit techniques that optimally leverage the DCK embedded in the planning instance. As such, we investigate how state-of-the-art 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 2006 International Planning Competition. Our results show that procedural DCK improves the performance of state-of-the-art planners, and that our heuristics are sometimes key to achieving good performance.

## 2 Background

In this section, we review the subset of PDDL2.1 that we use to define the semantics of our GOLOG-like language. PDDL is the de facto standard specification language for input to most state-of-the-art planners, providing a means of specifying planning domains (roughly, predicates and actions) and planning instances (roughly, objects, initial state, and goal specification).

### 2.1 A Subset of PDDL 2.1

In PDDL, 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 [Fox and Long 2003].

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 defined by a tuple $\langle O(\vec{x}), \vec{t}, \mathit{Prec}(\vec{x}), \mathit{Eff}(\vec{x})\rangle$, 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, $\mathit{Prec}(\vec{x})$ is a boolean formula with quantifiers (BFQ) 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 $\varphi$ is a BFQ, then so is $Qx\text{-}t\,\varphi$, for a variable $x$, a type symbol $t$, and $Q$ is either $\exists$ or $\forall$. 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:

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

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

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

where $\varphi$ is a BFQ whose only free variables are among $\vec{x}$ and $\vec{y}$, $R$ is a predicate, $f$ is a function, and $\mathit{obj}$ 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 $f(\vec{c}) = \mathit{obj}$).

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

**Semantics:** Fox and Long [2003] gave 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 $\varphi$ is true in state $s$. Moreover, for a planning instance $I$, we assume there exists a relation $\mathit{Succ}$ such that $\mathit{Succ}(s, a, s')$ 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 $s_0 = Init$, $\mathit{Succ}(s_i, a_{i+1}, s_{i+1})$ for $i \in \{0, \ldots, n-1\}$, and $s_n \models \mathit{Goal}$.

## 3 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 is based on GOLOG [Levesque, Reiter, Lespérance, Lin, and Scherl 1997] and thus 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.

## 3.1 Syntax

Our procedural search control language is based on GOLOG. 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 $\phi$ is a boolean formula with quantifiers on the language of $I$, possibly including terms in the set of program variables $V$. Atomic programs are defined as follows.

1. $nil$: the empty program.
2. $o$: a single operator instance, where $o \in \mathcal{O}$.
3. $\mathbf{any}$: "any action".
4. $\phi?$: a *test action*, that tests for the truth of formula $\phi$.

If $\sigma_1$, $\sigma_2$ and $\sigma$ 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. $\sigma^*$: 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.
- $(\, load(C, P); fly(P, LA) \mid load(C, T); drive(T, LA)\, )$: Either load $C$ on the plane $P$ or on the truck $T$, and perform the right action to move the vehicle to $LA$.

### 3.2 Semantics

The problem of planning for an instance $I$ under the control of program $\sigma$ corresponds to finding a plan for $I$ that is also an execution of $\sigma$ 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 $a_1 \cdots a_n$ corresponds to the execution of a program $\sigma$. The device we use is a nondeterministic finite state automaton with $\varepsilon$-transitions ($\varepsilon$-NFA).

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

An $\varepsilon$-NFA $A_{\sigma,I}$ is defined for each program $\sigma$ 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 $\sigma$ 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, $\sigma$, as well as the current planning state after performing the actions it has read already, $s$.

Formally, $A_{\sigma,I} = (Q, \mathcal{A}, \mathit{Tr}, q_o, F)$, where $Q$ is the set of program configurations, the alphabet $\mathcal{A}$ is a set of domain actions, the transition function is $\mathit{Tr} : Q \times (\mathcal{A} \cup \{\varepsilon\}) \to 2^Q$, $q_0 = [\sigma, \mathit{Init}]$, and $F$ is the set of final states.

Our definition of $\mathit{Tr}$ closely follows the definition of $\mathit{Trans}$ and $\mathit{Final}$ from GOLOG's transition semantics [De Giacomo, Lespérance, and Levesque 2000].

The transition function $\mathit{Tr}$ is defined as follows for atomic programs.

$$\mathit{Tr}([a, s], a) = \{[\mathit{nil}, s']\} \quad \text{iff } \mathit{Succ}(s, a, s'),\ \text{for any } a \in \mathcal{A}, \tag{4}$$

$$\mathit{Tr}([\mathbf{any}, s], a) = \{[\mathit{nil}, s']\} \text{ iff } \mathit{Succ}(s, a, s'),\ \text{for any } a \in \mathcal{A}, \tag{5}$$

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

Equations 4 and 5 dictate that actions in programs change the state according to the $\mathit{Succ}$ relation described in the previous section. Finally, Equation 6 defines transitions for $\phi$? when $\phi$ 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 $\phi$.

Now we define $\mathit{Tr}$ for non-atomic programs. In the definitions below, assume that $a \in \mathcal{A} \cup \{\varepsilon\}$, and that $\sigma_1$ and $\sigma_2$ are subprograms of $\sigma$, where occurring elements in $V$ may have been instantiated by any object in the planning instance $I$.

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

$$\mathit{Tr}([(\mathit{nil};\sigma_2), s], \varepsilon) = \{[\sigma_2, s]\}, \tag{8}$$

$$\mathit{Tr}([\mathbf{if}\,\phi\,\mathbf{then}\,\sigma_1\,\mathbf{else}\,\sigma_2, s], \varepsilon) = \begin{cases} [\sigma_1, s] & \text{if } s \models \phi, \\ [\sigma_2, s] & \text{if } s \not\models \phi, \end{cases} \tag{9}$$

$$\mathit{Tr}([(\sigma_1|\sigma_2), s], \varepsilon) = \{[\sigma_1, s], [\sigma_2, s]\} \tag{10}$$

$$\mathit{Tr}([\mathbf{while}\,\phi\,\mathbf{do}\,\sigma_1, s], \varepsilon) = \begin{cases} \{[\mathit{nil}, s]\} & \text{if } s \not\models \phi, \\ \{[\sigma_1; \mathbf{while}\,\phi\,\mathbf{do}\,\sigma_1, s]\} & \text{if } s \models \phi, \end{cases} \tag{11}$$

$$\mathit{Tr}([\sigma_1^*, s], \varepsilon) = \{[(\sigma_1;\sigma_1^*), s], [\mathit{nil}, s]\}, \tag{12}$$

$$\mathit{Tr}([\pi(x\text{-}t)\,\sigma_1, s], \varepsilon) = \{[\sigma_1|_{x/o}, s] \mid (o, t) \in \tau_D \cup \tau_P\}. \tag{13}$$

where $\sigma_1|_{x/o}$ denotes the program resulting from replacing any occurrence of $x$ in $\sigma_1$ by $o$. We now give some intuitions for the definitions. 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). A transition on a conditional corresponds to a transition in the *then* or *else* part depending on the truth value of the condition (Eq. 9). A transition of the nondeterministic choice leads to the consideration of either of the programs (Eq. 10). A transition of a while-loop corresponds to the $nil$ program if the condition is false, and corresponds to the body followed by the while-loop if the condition is true (Eq. 11). On the other hand, a transition of $\sigma_1^*$ represents two alternatives: executing $\sigma_1$ at least once, or stopping the execution of $\sigma_1^*$, with the remaining program $nil$ (Eq. 12). Finally, a transition of the nondeterministic choice corresponds to a transition of its body when the variable has been replaced by any object of the right type (Eq. 13).

To end the definition of $A_{\sigma,I}$, $Q$ corresponds precisely to the program configurations $[\sigma', s]$ where $\sigma'$ is either $nil$ or a subprogram of $\sigma$ such that program variables may have been replaced by objects in $I$, and $s$ is any possible planning state. Moreover, $Tr$ is assumed empty for elements of its domain not explicitly mentioned above. Finally, the set of accepting states is $F = \{[nil, s] \mid s \text{ is any state over } I\}$, 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 $\sigma$ in $I$ iff $a_1 \cdots a_n$ is accepted by $A_{\sigma,I}$.

We use the symbol $\vdash$ to represent a single computation of the automaton. We say that $q \vdash q'$ iff there exists an $a$ such that $q' \in Tr(q, a)$. The symbol $\vdash^*$ represents the reflexive and transitive closure of $\vdash$. Finally, $q_0 \vdash^k q_k$ iff there are exist states $q_1, \ldots, q_{k-1}$ such that $q_0 \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_{k-1} \vdash q_k$.

Before defining what we mean by planning in the presence of control, we prove a number of results that justify the correctness of our automata-based semantics. The detailed proofs can be found in Baier's Ph.D. thesis [Baier 2010]. The first result proves that the definition of the sequence is intuitively correct, i.e., the execution of $\sigma_1; \sigma_2$ corresponds to the execution of $\sigma_1$ followed by $\sigma_2$.

PROPOSITION 2. *Let $\sigma_1$ and $\sigma_2$ be programs. If*

$$[\sigma_1; \sigma_2, s] \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_{k-1} \vdash q_k = [nil, s'],$$

*then for some $i \in [1, k]$, $q_i = [\sigma_2, s']$ and $[\sigma_1, s] \vdash^* [nil, s']$.*

Our second result establishes that the semantics for the execution of an **if - then - else** is intuitively correct.

PROPOSITION 3. *Let $\phi$ be a BFQ and let $\sigma_1$ and $\sigma_2$ be programs, then the following holds:*

$$[\mathbf{if}\,\phi\,\mathbf{then}\,\sigma_1\,\mathbf{else}\,\sigma_2, s] \vdash^* [nil, s']$$

*iff*

$$s \models \phi \textit{ and } [\sigma_1, s] \vdash^* [nil, s'], \textit{ or } s \not\models \phi \textit{ and } [\sigma_2, s] \vdash^* [nil, s'].$$

The execution of a nondeterministic choice of programs has the intended meaning too, as shown by the following result.

PROPOSITION 4. *Let $\sigma_1$ and $\sigma_2$ be programs, then the following holds:*

$$[(\sigma_1|\sigma_2), s] \vdash^* [nil, s'] \quad \textit{iff} \quad [\sigma_1, s] \vdash^* [nil, s'] \textit{ or } [\sigma_2, s] \vdash^* [nil, s'].$$

Now we prove that the execution of the while loop correspond to a repeated execution of the body of the loop.

PROPOSITION 5. *Let $\phi$ be a BFQ and $\sigma$ be a program. If*

$$[\mathbf{while}\, \phi\, \mathbf{do}\, \sigma, s] \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_k \vdash [nil, s'],$$

*then:*

1. *for all $i \in [1, k]$, $q_i$ is either of the form $q_i = [\sigma_r; \mathbf{while}\, \phi\, \mathbf{do}\, \sigma, r_i]$, or of the form $q_i = [\mathbf{while}\, \phi\, \mathbf{do}\, \sigma, r_i]$.*
2. *for all $i \in [1, k]$, if $q_i$ is of the form $q_i = [\mathbf{while}\, \phi\, \mathbf{do}\, \sigma, r_i]$ then $i < k$ iff $r_i \models \phi$. State $q_k$ is of the form $q_k = [\mathbf{while}\, \phi\, \mathbf{do}\, \sigma, r_k]$*
3. *Finally, let $n$ be the number of states $q_i$ ($i \in [1, k]$) of the form $q_i = [\mathbf{while}\, \phi\, \mathbf{do}\, \sigma, r_i]$. Then, $[\sigma^n, s] \vdash^* [nil, s']$, where $\sigma^n$ represents the sequence that repeats $\sigma$ $n$ times.*

In the GOLOG language [Levesque, Reiter, Lespérance, Lin, and Scherl 1997], the **if - then - else** construct is defined by macro expansion, in terms of test actions and non-deterministic choices. Below we prove that our semantics for the **if - then - else** and for the GOLOG macro expansion of such a construct are equivalent.

PROPOSITION 6. *Let $\phi$ be a BFQ and let $\sigma_1$ and $\sigma_2$ be programs, then the following holds:*

$$[\mathbf{if}\, \phi\, \mathbf{then}\, \sigma_1\, \mathbf{else}\, \sigma_2, s] \vdash [\sigma, s] \quad \textit{iff} \quad [(\phi?; \sigma_1)|(\neg\phi?; \sigma_2), s] \vdash^3 [\sigma, s].$$

Now that we have justified the correctness of the semantics of the control language, we return to planning. We are now ready to define the notion of planning under procedural control.

DEFINITION 7 (Planning under procedural control). A sequence of actions $a_1 a_2 \cdots a_n$ is a *plan for instance $I$ under the control of program $\sigma$* iff $a_1 a_2 \cdots a_n$ is a plan for $I$ and is an execution of $\sigma$ in $I$.

## 4 Compiling Control into the Action Theory

This section describes a translation function that, given a program $\sigma$ in the DCK language defined above together with a PDDL2.1 domain specification $D$, outputs a new PDDL2.1 domain specification $D_\sigma$ and problem specification $P_\sigma$. 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 $\sigma$, i.e. that is such that only action sequences that are executions of $\sigma$ 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 $\sigma$ 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

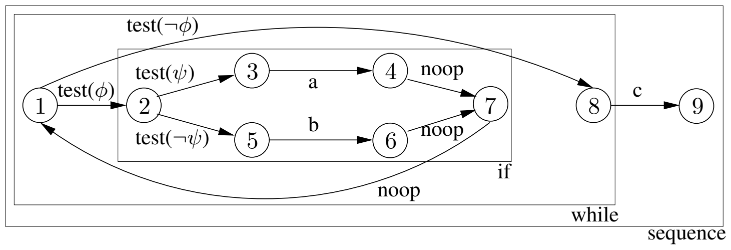


Figure 1: Automaton for $\mathbf{while}\,\phi\,\mathbf{do}\,(\mathbf{if}\,\psi\,\mathbf{then}\,a\,\mathbf{else}\,b); c$.

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.

The translation is defined inductively by a function $C(\sigma, n, E)$ which takes as input a program $\sigma$, 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 $\sigma$. Moreover, $L'$ contains restrictions on the applicability of operators defined in $I$, and $L$ contains additional control operators needed to enforce the search control defined in $\sigma$. Integers $n$ and $n'$ abstractly denote the program state before and after execution of $\sigma$.

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 a similar transition, but conditioned on $\phi$. They are defined as:[1]

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

Function $mentions(\phi, E)$ returns a vector $\vec{e\text{-}t}$ of program variables and types that occur in $\phi$, 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

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

defined as:

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

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

The inductive definitions for other programs $\sigma$ are:

$$\begin{aligned}
&C(\mathit{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},\\
&\quad \vec{r} = r_1, \ldots, r_m,\\
&\quad \mathit{Prec}'(\vec{x}) = (\mathit{state} = s_n \wedge \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\qquad [\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) = (\ [\mathit{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\qquad \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 = [\mathit{Ctest}(\phi, n, n+1, E),\\
&\quad \mathit{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) \text{ with}\\
&\quad (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 \mathit{free\_n_1}(x), t,\ \mathit{state} = s_{n_1},\\
&\qquad\qquad [\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 $\sigma$, the return value $(L, L', n_{\text{final}})$ 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 $D_\sigma$.

$D_\sigma$ 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 $D_\sigma$ 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, $D_\sigma$ contains all operator definitions in $L$. Objects in $D_\sigma$ are the same as those in $D$, plus a few new ones to represent the program variables and the automaton's states $s_i$ ( $0 \leq i \leq n_{\text{final}}$). Finally $D_\sigma$ 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 $P_\sigma$ is defined as follows. Given any predefined problem $P$ over $D$, $P_\sigma$ 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 $\sigma$, except that plans on $I_\sigma$ contain actions that were not part of the original domain definition (*test*, *noop*, and *free*).

THEOREM 8 (Correctness). *Let Filter*$(\alpha, D)$ *denote the sequence that remains when removing from* $\alpha$ *any action not defined in* $D$. *If* $\alpha$ *is a plan for instance* $I_\sigma = (D_\sigma, P_\sigma)$ *then Filter*$(\alpha, D)$ *is a plan for* $I = (D, P)$ *under the control of* $\sigma$. *Conversely, if* $\alpha$ *is a plan for* $I$ *under the control of* $\sigma$, *there exists a plan* $\alpha'$ *for* $I_\sigma$, *such that* $\alpha =$ *Filter*$(\alpha', D)$.

**Proof.** Appears in [Baier, Fritz, and McIlraith 2007]. □

Now we turn our attention to analyzing the size 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 9 (Succinctness). *Let* $\sigma$ *is a program of size* $m$, *and let* $k$ *be the maximal nesting depth of* $\pi(x\text{-}t)$ *statements in* $\sigma$, *then* $|I_\sigma|$ *(the overall size of* $I_\sigma$*) is* $O((k+p)m)$, *where* $p$ *is the size of the largest operator in* $I$.

**Proof.** Appears in [Baier, Fritz, and McIlraith 2007]. □

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.

## 5 Exploiting DCK in State-of-the-Art Heuristic Planners

Our objective in translating procedural DCK to PDDL2.1 was to enable *any* PDDL2.1-compliant state-of-the-art 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 task 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.

## 5.1 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 $\sigma$, and use the resulting instance $I_\sigma$ with any heuristic planner. We call this the *Simple* heuristic.

Unfortunately, when exploiting a relaxed graph to compute heuristics, two issues arise. *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 $\sigma_L \stackrel{\text{def}}{=} \textbf{while}\,\phi\,\textbf{do}\,\pi(c\text{-}crate)\,unload(c,T)$, is intended for a domain where crates can be only unloaded sequentially from a truck. As soon as $c$ is assigned a value, such a value will be considered in all possible iterations of the while loop, which is not what is intended by the program, and has the potential of 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 has an effect on the time required to compute the heuristic.

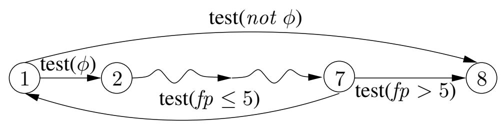


Figure 2: *H-ops* translation of **while** loops. While computing the heuristics, pseudo-fluent *fp* is increased each time no new effect is added into the relaxed state, and it is set to 0 otherwise. The loop can be exited if the last five (7-2) actions performed didn't add any new effect.

## 5.2 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 use heuristics based on the relaxed planning graph. 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 $D_\sigma$ 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.

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 $\sigma_L$ 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 $\phi$ 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 **if** construct whose condition is both true and false at the same time (this can happen because $p$ and $not$-$p$ 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 illustration. An important detail to note is that the loop is not entered when $\phi$ 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 **if**'s, if the condition is both true and false at the same time, the **then** part is processed first, followed by the **else** part. The objective of this is avoidance of state

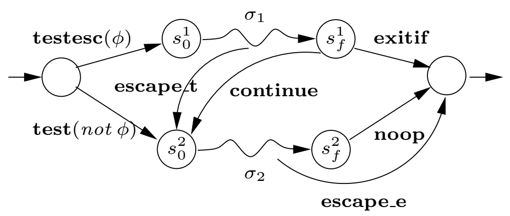


Figure 3: *H-ops* translation for **if - then - else** . Action $\mathbf{testesc}(\phi)$ is possible if condition $\phi$ is true. If condition $\neg\phi$ is also true in the relaxed state, the $\mathbf{testesc}(\phi)$ adds a fact $escape\_active$ that will enable the execution of **continue** and **escape_t** 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.

duplication. However, this new interpretation of the **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 **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.

To solve this problem, *H-ops* 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 "else" part. Figure 3 provides a graphical depiction.

### 5.3 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 $(I_\sigma, Ops)$, where $I_\sigma$ 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. Note that 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.

## 6 Implementation and Experiments

Our implementation 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 details of the GOLOG code used for these examples, see [Baier 2010]. 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, Figure 4 shows times for the different heuristic approaches.

Not surprisingly, our data confirms that DCK helps to improve planner performance, 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.

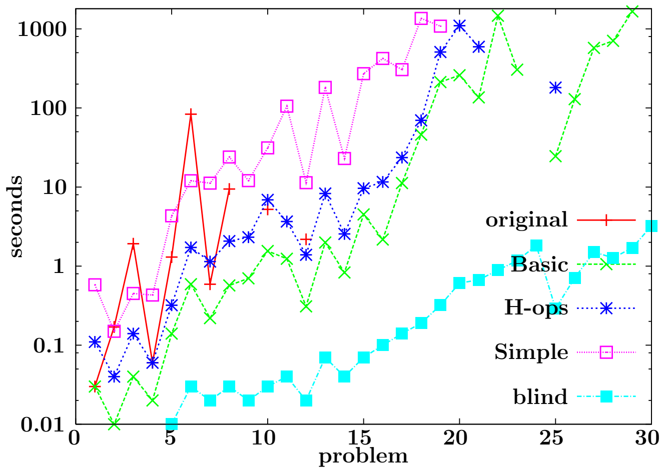
We observe that *H-ops* is the most informative (expands fewer nodes). This fact does not pay off in time in the experiments shown in the table. Nevertheless, it is easy to construct instances where the *H-ops* performs better than *Basic*. This happens when the DCK control restricts the space of valid 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.

## 7 Related Work and Discussion

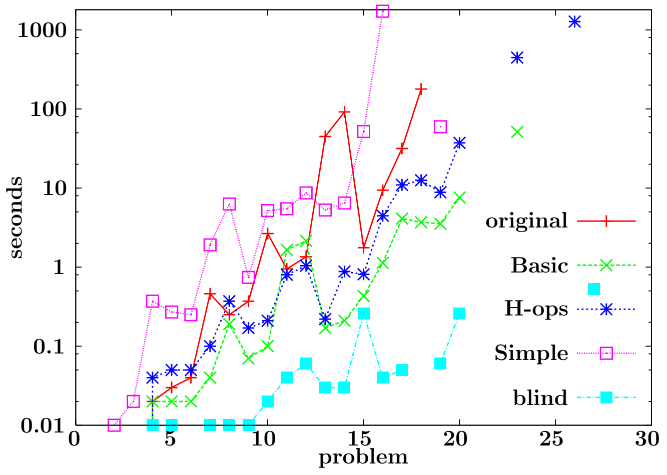
DCK can be used to constrain the set of valid plans and has proven an effective tool in reducing the time required to generate a plan. Nevertheless, many of the planners that exploit it use arguably less natural state-centric DCK specification languages, and their planners use blind search. In this paper we examined the problem of exploiting procedural DCK with state-of-the-art planners. Our goal was to specify rich DCK naturally in the form of a program template and to exploit state-of-the-art planning techniques to actively plan towards the achievement of this DCK. To this end we made three contributions: provision of a GOLOG-like procedural DCK language syntax and PDDL 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 state-of-the-art planner, not just the domain-independent heuristic search planner that we propose. Our experiments show that procedural DCK improves the performance of state-of-the-art 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 and Kabanza 1998] and TALPLANNER [Kvarnström and 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.

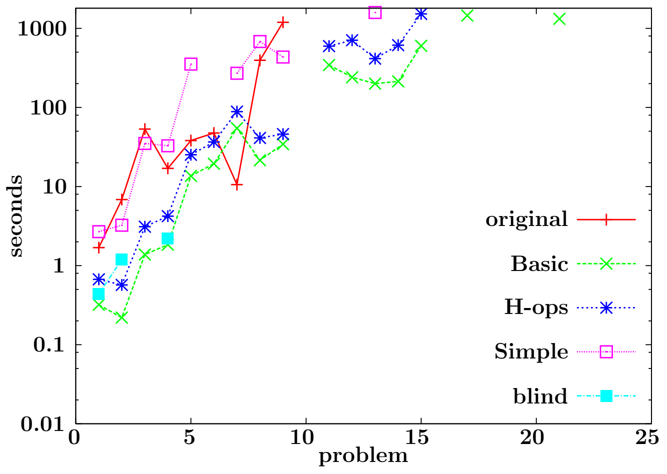
Though not described as DCK specification languages there are a number of languages



(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

| | | *original* | *Simple* | *Basic* | *H-ops* | *blind* |
|---|---|---|---|---|---|---|
| Trucks | #n | 1 | 0.31 | 0.41 | 0.26 | 19.85 |
| | #s | 9 | 9 | 15 | 14 | 3 |
| | $\ell_{\text{min}}$ | 1 | 1 | 1 | 1 | 1 |
| | $\ell_{\text{avg}}$ | 1.1 | 1.03 | 1.02 | 1.04 | 1.04 |
| | $\ell_{\text{max}}$ | 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 |
| | $\ell_{\text{min}}$ | 1 | 1 | 1 | 1 | 1 |
| | $\ell_{\text{avg}}$ | 2.13 | 1.03 | 1.05 | 1.21 | 1.53 |
| | $\ell_{\text{max}}$ | 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 |
| | $\ell_{\text{min}}$ | 1 | 1 | 1 | 1 | 1 |
| | $\ell_{\text{avg}}$ | 4.4 | 1.05 | 1.01 | 1.07 | 1.62 |
| | $\ell_{\text{max}}$ | 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. $\ell_{\text{avg}}$ denotes the average ratio of the plan length to the shortest plan found by any of the approaches (i.e., $\ell_{\text{avg}}$=1.50 means that on average, on each instance, plans where 50% longer than the shortest plan found for that instance). $\ell_{\text{min}}$ and $\ell_{\text{max}}$ are defined analogously.

from the agent programming and/or model-based programming communities that are related to procedural control. We have mentioned the GOLOG logic programming language. EAGLE is a somewhat related goal language designed to also express intentionality [dal Lago, Pistore, and Traverso 2002]. Further, languages such as the Reactive Model-Based Programming Language (RMPL) [Kim, Williams, and 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, HTN specification languages such as those used in SHOP [Nau, Cao, Lotem, and Muñoz-Avila 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 state-of-the-art 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, Eyerich, Lakemeyer, and Nebel [2007] have proposed to integrate an incremental GOLOG interpreter with a state-of-the-art 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 state-of-the-art 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 and McIlraith [2006], Cresswell and 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, Baral, Nam, and McIlraith [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 state-of-the-art planners, nor do they propose any heuristic approaches to planning with them.

## 8 Postlude

The focus of this research, as described here, has been to improve planner performance while also enabling standard planners to plan for a richer class of goals. In addition to this practical achievement, there are a number of interesting theoretical insights in this work. We explore these in more detail in [Fritz, Baier, and McIlraith 2008]. In this later work we go beyond the simplified GOLOG-like language discussed here, and consider full GOLOG with procedures, including its extension for concurrency, ConGolog [De Giacomo, Lespérance, and Levesque 2000].

The main technical contribution of this follow-on work is the definition of a compilation scheme that takes a ConGolog program and a basic action theory of the situation calculus as input and outputs a new basic action theory that represents the program in the context of the original theory. I.e., the resulting basic action theory describes the same tree of situations as the tree of situations induced by the program in *conjunction* with the original basic action theory. This compilation eliminates the need for a ConGolog interpreter. The resulting theory can be interpreted by any implementation of the situation calculus. Further, providing semantics to ConGolog programs in this way eliminates the need for reification in the specification of the semantics.

This *compiled semantics* is useful for a variety of purposes. First and foremost, as discussed in the previous sections, standard PDDL-compliant planners can plan using ConGolog programs to specify search control and/or temporally extended goals. While the target language in [Fritz, Baier, and McIlraith 2008] was that of basic action theories of the situation calculus, in certain cases it is possible to also compile to other representations, such as PDDL. This requires certain restrictions to keep the state space finite, but important features such as concurrency *can* be represented. Intuitively, in the context of the previously described compilation to PDDL, this is achieved by allowing the state machine to be in multiple states at the same time. The type of concurrency achieved is interleaving, just as it is in the original ConGolog.

Secondly, it is possible to represent a significant subset of HTNs [Ghallab, Nau, and Traverso 2004] in ConGolog, which given this work means that it is now possible to reason about such HTNs in the situation calculus as well. Given ConGolog's ability to represent and reason about recursive procedures, these HTNs may similarly make use of recursive method definitions.

Thirdly, since the semantics of programs in the compilation result is captured via fluents, action preconditions and action effects, it is possible to reason about program executions using regression. Loosely speaking, one can "regress programs", i.e., reduce all the constraints that are imposed by a program on the legal executions of that program into constraints about the initial state of the world, when presented with a sequence of actions. This is different from regressing a formula over a program. Rather, since after compilation all constraints that the program imposes on the evolution of the world (the tree of situations described by the underlying basic action theory) are now explicitly expressed as conditions over fluents, these program constraints can be regressed. Just as this was the original motivation for Reiter's use of regression, this use of regression allows applying regular theorem provers that only need to reason about the initial state. Hence, reasoning about the truth values of fluents amounts to mere database look-up. Practically this means that, when given a specific sequence of actions, one can test whether or not it constitutes a legal execution of the regressed program without temporal projection and without the need for program interpretation. This has significant implications with respect to execution monitoring of (Con)Golog programs, building on the unifying perspective of execution monitoring developed in Fritz's Ph.D. thesis [Fritz 2009].

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

## References

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

Baier, J. [2010]. *Effective search techniques for non-classical planning via reformulation*. Ph.D. thesis, University of Toronto, Toronto, Canada.

Baier, J., C. Fritz, and S. McIlraith [2007]. Exploiting procedural domain control knowledge in state-of-the-art planners (extended version). Technical Report CSRG-565, University of Toronto.

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

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

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

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

De Giacomo, G., Y. Lespérance, and H. Levesque [2000]. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence 121*(1-2), 109–169.

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)*, Lake District, UK, pp. 374–377.

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

Fritz, C. [2009]. *Monitoring the Generation and Execution of Optimal Plans*. Ph.D. thesis, University of Toronto, Toronto, Canada.

Fritz, C., J. A. Baier, and S. A. McIlraith [2008]. ConGolog, Sin Trans: Compiling ConGolog into basic action theories for planning and beyond. In *Proc. on the 11th Int'l Conference on Principles of Knowledge Representation and Reasoning (KR-08)*, Sydney, Australia, pp. 600–610.

Ghallab, M., D. Nau, and P. Traverso [2004]. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Kim, P., B. C. Williams, and M. Abramson [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)*, Seattle, USA, pp. 487–493.

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

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

McDermott, D. V. [1998]. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

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

Pednault, E. P. D. [1989]. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of the 1st Int'l Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, Toronto, Canada, pp. 324–332.

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

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