Computing Plans with Control Flow and Procedures Using a Classical Planner

Sergio Jiménez and **Anders Jonsson**
Dept. Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138
08018 Barcelona, Spain
{sergio.jimenez,anders.jonsson}@upf.edu

Abstract

We propose a compilation that enhances a given classical planning task to compute plans that contain control flow and procedure calls. Control flow instructions and procedures allow us to generate compact and general solutions able to solve planning tasks for which multiple unit tests are defined. The paper analyzes the relation between classical planning and structured programming with unit tests and shows how to exploit this relation in a classical planning compilation. In experiments, we evaluate the empirical performance of the compilation using an off-the-shelf classical planner and show that we can compress classical planning solutions and that these compressed solutions can solve planning tasks with multiple tests.

Introduction

As shown in generalized planning (Hu and De Giacomo 2011) and in learning-based planning (Zimmerman and Kambhampati 2003; Jiménez et al. 2012), some classical planning domains admit general solutions that are not tied to a specific instance. An example is the *gripper* domain where the following general strategy solves any instance:

(1) while there are still balls to be moved:
(2) pick up a ball,
(3) move to the goal location,
(4) put down the ball,
(5) return to the initial location.

A key issue is to identify compact representations that can express general solution strategies of this type. One such example comes from structured programming, which reduces the size (and development time) of computer programs by exploiting *selection* and *repetition* constructs (i.e. control flow) as well as procedures (DeMillo, Eisenstat, and Lipton 1980).

In this work we aim to compute general solution strategies for planning in the form of programs that include control flow and procedures. Although the idea of representing plans as programs is not new (Lang and Zanuttini 2012), we are not aware of any previous work that computes plans in the form of programs starting from a basic description of the domain.

We propose a novel compilation that enhances a given classical planning task with two mechanisms from programming: *goto* instructions and parameter-free procedures. Given that these programming constructs make it possible to generate compact and general solutions, we also study a special case of the compilation, namely planning tasks that model multiple instances at once. Incidentally, the compilation enables us to express simple programming tasks as planning tasks.

Central to the compilation are the notions of *program lines* and *instructions*, where the latter include the actions of the original planning task. To solve the compiled task, a planner has to both *program* the solution, i.e. decide which instructions to include on program lines, and *execute* the program to verify that it actually solves each instance.

The paper is organized as follows. First, we review classical planning and structured programming, the two models we rely on throughout the paper, and analyze their relationship. Next, we describe a series of compilations which, in turn, include goto instructions, parameter-free procedures and multiple instances. Finally, we present an empirical evaluation of the various compilations, discuss related work and finish with a conclusion.

Background

In this section we review classical planning and describe a simplified representation of an automatic programming task.

Classical Planning

We consider the classical planning formalism that includes conditional effects and costs, and use sets of literals to describe conditions, effects, and states. Given a set of fluents F, a set of literals L is a partial assignment of values to fluents, represented by literals f or $\neg f$. Given L, let $F(L) = \{f : f \in L \vee \neg f \in L\}$ be the subset of fluents in the assignment and let $\neg L = \{\neg l : l \in L\}$ be the complement of L. A *state* s is a set of literals such that $F(s) = F$.

A classical planning task is a tuple $P = \langle F, A, I, G\rangle$ with F a set of fluents, A a set of actions, I an initial state and G a goal condition, i.e. a set of literals. Each action $a \in A$ has a set of literals $\mathsf{pre}(a)$ called the *precondition*, and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect

$C \rhd E \in \mathsf{cond}(a)$ is composed of sets of literals C (the condition) and E (the effect).

Action a is applicable in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s,a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is a new state $\theta(s,a) = (s \setminus \neg\mathsf{eff}(s,a)) \cup \mathsf{eff}(s,a)$.

A plan solving a planning task P is a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$, inducing a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$, $G \subseteq s_n$ and the application of each action a_i, $1 \leq i \leq n$, at state s_{i-1} generates the successor state $s_i = \theta(s_{i-1}, a_i)$. We define a cost function $c : A \to \mathbb{N}$ such that the cost of a plan π is the sum of its action costs, $c(\pi) = \sum_i c(a_i)$, and π is optimal if $c(\pi)$ is minimized.

Automatic Programming

Our definition of automatic programming is loosely based on linear genetic programming (Brameier 2004). We define an automatic programming task as a tuple $\mathcal{S} = \langle R, R_o, \mathcal{I}, \mathcal{T}, n \rangle$ with R a set of registers, $R_o \subseteq R$ a set of output registers, $\mathcal{I}$ a set of instructions, $\mathcal{T}$ a set of *unit tests*, each modelling a concrete instance of the problem, and n a fixed number of program lines. Each register $r \in R$ has finite domain D.

Each instruction $i \in \mathcal{I}$ has a subset of registers $R_i \subseteq R$ and is either *sequential* or *control flow*. In most programming languages, $|R_i| \leq 3$. A sequential instruction i has a mapping $\phi_i : D^{|R_i|} \to D^{|R_i|}$, and given a joint value $d \in D^{|R_i|}$, executing i on line k modifies the registers in R_i as $\phi_i(d)$ and increments k. A control flow instruction i has a mapping $\varphi_i : D^{|R_i|} \to \{0, \ldots, n, \bot\}$, and given $d \in D^{|R_i|}$, executing i on line k leaves registers unchanged but sets k to $\varphi_i(d)$ if $\varphi_i(d) \neq \bot$, else increments k. Such instructions can simulate both conditional statements and loops. Each unit test $t \in \mathcal{T}$ assigns an initial value $t_i(r) \in D$ to each register $r \in R$ and a desired value $t_d(r) \in D$ to each output register $r \in R_o$.

A program $\Pi = \langle i_0, \ldots, i_m \rangle, m < n$, assigns instructions to program lines. To execute Π on a unit test $t \in \mathcal{T}$ we initialize each register $r \in R$ to $t_i(r)$, initialize a program counter k to 0 and repeatedly execute the instruction i_k on line k until $k > m$ (we later address the issue of infinite loops). Π solves $\mathcal{S}$ if and only if, for each unit test $t \in \mathcal{T}$, the value of each output register $r \in R_o$ is $t_d(r)$ after executing Π on t.

As an example, consider the task of computing the summatory $\sum_{j=1}^{m} j$ of a number $m > 0$. We can define this as an automatic programming task $\mathcal{S} = \langle R, R_o, \mathcal{I}, \mathcal{T}, 3 \rangle$ with $R = \{x, y, z\}$, $R_o = \{z\}$, $D = \{0, \ldots, M\}$ for some $M > 0$ and three types of instructions in $\mathcal{I}$:

Instruction	R_i	$\phi_i(d)/\varphi_i(d)$
$inc(r_1)$	$\{r_1\}$	$\phi(d_1) = (d_1 + 1)$
$add(r_1, r_2)$	$\{r_1, r_2\}$	$\phi(d_1, d_2) = (d_1 + d_2, d_2)$
$goto(k', r_1 \mathrel{!}= r_2)$	$\{r_1, r_2\}$	$\begin{cases} \varphi(d_1, d_2) = k', & d_1 \neq d_2 \\ \varphi(d_1, d_2) = \bot, & d_1 = d_2 \end{cases}$

Sequential instruction inc increments the value of a register and add adds the value of a register to that of another. Control flow instruction $goto$ jumps to program line k' if r_1 and r_2 have different values, else increments k. Each unit test $t \in \mathcal{T}$ sets initial values $t_i(x) = m$ and $t_i(y) = t_i(z) = 0$ and desired value $t_d(z) = \sum_{j=1}^{m} j \leq M$ for some m. Figure 1 shows an example of a 3-line program for solving this task.

```
0. inc(y)
1. add(z,y)
2. goto(0,y!=x)
```

Figure 1: Example program for computing the summatory.

A common extension to automatic programming is *procedures*, i.e. subsequences of instructions that can be called repeatedly. An automatic programming task with procedures is a tuple $\mathcal{S} = \langle R, R_o, \mathcal{I}, \mathcal{T}, n, \mathcal{P} \rangle$ with $\mathcal{P}$ the set of procedures, each with n program lines. We consider parameterfree procedures that operate on the same set of registers R. The set $\mathcal{I}$ includes a third type of instruction called *procedure call*, that does not modify registers but instead executes a procedure.

A program Π assigns instructions to the program lines of each procedure in $\mathcal{P}$. In general, to execute a program with procedures it is necessary to extend the execution model with a call stack that remembers from where each procedure was called. In this paper, however, we do not allow nested procedure calls, in which case execution is easier to manage. Figure 2 shows an example program for computing the desired value $t_d(z) = 2^{2m} - 1$ given initial values $t_i(x) = m$ and $t_i(y) = t_i(z) = 0$.

```
main:  0. p1          p1:  0. add(z,z)
       1. p1               1. inc(z)
       2. inc(y)
       3. goto(0,y!=x)
```

Figure 2: Example program for computing $z = 2^{2m} - 1$.

Automatic Programming and Planning

In this section we show that planning and our restricted form of automatic programming are *equivalent* in the sense that one can be reduced to the other. Although our main motivation is to exploit mechanisms from programming (procedures and gotos) to produce compact solutions to classical planning tasks, the equivalence also means that we can model simple programming tasks as planning problems.

We first define a class PF of planning tasks that we call *precondition-free*, i.e. $\mathsf{pre}(a) = \emptyset$ for each $a \in A$. Although this appears restrictive, note that any action $a = \langle \mathsf{pre}(a), \mathsf{cond}(a) \rangle$ can be compiled into a precondition-free action $a' = \langle \emptyset, \{(\mathsf{pre}(a) \cup C) \rhd E : C \rhd E \in \mathsf{cond}(a)\} \rangle$; the only difference between a and a' is that a' is applicable when $\mathsf{pre}(a)$ does not hold, but has no effect in this case.

Let BPE(PF) be the decision problem of bounded plan existence for precondition-free planning tasks, i.e. deciding if

an arbitrary precondition-free planning task P has a solution π of length $|\pi| \leq K$. Likewise, let PE(A) be the decision problem of program existence for the automatic programming tasks.

Theorem 1. *BPE(PF) is polynomial-time reducible to PE(A).*

Proof. Given an instance $P = \langle F, A, I, G\rangle$, K of BPE(PF), construct a programming task $\mathcal{S} = \langle F, F(G), \mathcal{I}, \{t\}, K\rangle$. Since registers are binary, sets of literals are partial assignments to registers. Thus I is the set of initial values of the only test t, and G is the set of desired values. For each $a \in A$, $\mathcal{I}$ contains an associated sequential instruction i defined as

$$R_i = \bigcup_{C \rhd E \in \mathsf{cond}(a)} F(C) \cup F(E),$$
$$\phi_i(d) = (d \setminus \neg\mathsf{eff}(d,a)) \cup \mathsf{eff}(d,a).$$

Note that fluents not in R_i are irrelevant for computing the triggered effect of a, so an assignment d to R_i suffices to compute $\mathsf{eff}(d,a)$. Since all instructions in $\mathcal{I}$ are sequential, program $\Pi = \langle i_0, \ldots, i_m\rangle$ solves $\mathcal{S}$ if and only if $m < K$ and plan $\pi = \langle a_0, \ldots, a_m\rangle$ solves P where, for each k, $0 \leq k \leq m$, a_k is the action associated with instruction i_k. □

To show the opposite direction, that programming tasks can be reduced to planning tasks, we proceed in stages. We first define a class ALSS of procedure-less programming tasks with sequential instructions and a single test, and present a reduction from PE(ALSS) to BPE(PF). In the next section we show how to add control flow instructions, procedures and multiple tests directly to planning tasks.

Theorem 2. *PE(ALSS) is reducible to BPE(PF).*

Proof. Let $\mathcal{S} = \langle R, R_o, \mathcal{I}, \{t\}, n\rangle$ be an instance of PE(ALSS). Construct a planning task $P = \langle F, A, I, G\rangle$ with $F = \{r[d] : r \in R, d \in D\}$. The initial state is $I = \{r[t_i(r)] : r \in R\} \cup \{\neg r[d] : r \in R, d \in D - \{t_i(r)\}\}$ and the goal is $G = \{r[t_d(r)] : r \in R_o\}$, where $t_i(r)$ and $t_d(r)$ are the initial and desired values of register r for the only test t.

For each instruction $i \in \mathcal{I}$, A contains an associated action a that simulates i. Let us assume that $R_i = \{r_1, \ldots, r_q\}$. Action a has empty precondition and conditional effects

$$\begin{aligned}\mathsf{cond}(a) = \{&\{r_1[d_1], \ldots, r_q[d_q]\}\\ &\rhd \{\neg r_1[d_1], r_1[d'_1], \ldots, \neg r_q[d_q], r_q[d'_q]\}\\ &: (d_1, \ldots, d_q) \in D^q, \phi_i(d_1, \ldots, d_q) = (d'_1, \ldots, d'_q)\}.\end{aligned}$$

Thus a has one conditional effect $C \rhd E$ per joint value $(d_1, \ldots, d_q) \in D^q$ that simulates the effect $\phi_i(d_1, \ldots, d_q)$ on registers in R_i. To make E well-defined, literals $\neg r_j[d_j]$ and $r_j[d'_j]$ are removed from E for each register $r_j \in R_i$ such that $d_j = d'_j$. Note that the size of $\mathsf{cond}(a)$ is exponential in $q = |R_i|$; however, if we assume that the mapping ϕ_i is explicitly defined in $\mathcal{S}$, the reduction remains polynomial in the size of $\mathcal{S}$. Moreover, from a practical perspective this is usually not an issue since $|R_i| \leq 3$ for any programming language based on binary, or possibly tertiary, instructions.

Clearly, P, n is an instance of BPE(PF). Let $\pi = \langle a_0, \ldots, a_m\rangle$ be a plan for P. Since actions simulate instructions, plan π solves P and satisfies $|\pi| \leq n$ if and only if program $\Pi = \langle i_0, \ldots, i_m\rangle$ solves $\mathcal{S}$, where i_k is the instruction associated with action a_k for each k, $0 \leq k \leq m$. □

When n is small, we can remove the bound n on the plan length by encoding a program counter using fluents $F_n = \{\mathsf{pc}_k : 0 \leq k \leq n\}$. Given action a, let a^k, $0 \leq k < n$, be an action with precondition $\mathsf{pre}(a^k) = \mathsf{pre}(a) \cup \{\mathsf{pc}_k\}$ and conditional effects $\mathsf{cond}(a^k) = \mathsf{cond}(a) \cup \{\emptyset \rhd \{\neg\mathsf{pc}_k, \mathsf{pc}_{k+1}\}\}$.

Given an arbitrary planning task $P = \langle F, A, I, G\rangle$, let $P_n = \{F \cup F_n, A_n, I_n, G\}$ be a modified planning task with $A_n = \{a^k : a \in A, 0 \leq k < n\}$ and $I_n = I \cup \{\mathsf{pc}_0\} \cup \{\neg\mathsf{pc}_k : 1 \leq k \leq n\}$. Since each action in A_n increments the program counter and no actions are applicable when pc_n holds, P_n has a solution if and only if P has a solution of length at most n.

Enhancing Planning Tasks

In this section we show how to enhance planning tasks with mechanisms from programming: goto instructions, parameter-free procedures and multiple unit tests. Since we manipulate planning tasks directly, actions do not have to be precondition-free.

Goto Instructions

Let $P = \langle F, A, I, G\rangle$ be a planning task. The idea is to define another planning task P'_n that models a programming task with n program lines. The set of instructions of the programming task is $A \cup \mathcal{I}_{go}$, i.e. the (sequential) actions of the planning task P enhanced with goto instructions. We incorporate the idea of a program counter from the previous section.

When control flow is not sequential, program lines may be visited multiple times. To ensure that different instructions are not executed on the same line, we define a set of fluents $F_{ins} = \{\mathsf{ins}_{k,i} : 0 \leq k \leq n, i \in A \cup \mathcal{I}_{go} \cup \{\mathsf{nil}, \mathsf{end}\}\}$ that each models the instruction i on line k. Instruction nil denotes an empty line, i.e., a line that has not yet been programmed, while end denotes the end of the program.

Let $i \in A \cup \mathcal{I}_{go} \cup \{\mathsf{end}\}$ be an actual instruction on line k, $0 \leq k \leq n$, and let i^k be the compilation of i that updates the program counter. Since i^k may be executed multiple times, we define two versions: $\mathsf{P}(i^k)$, that simultaneously programs and executes i on line k, and $\mathsf{R}(i^k)$, that repeats the execution of i on line k. Actions $\mathsf{P}(i^k)$ and $\mathsf{R}(i^k)$ are defined as

$$\begin{aligned}\mathsf{pre}(\mathsf{P}(i^k)) &= \mathsf{pre}(i^k) \cup \{\mathsf{ins}_{k,\mathsf{nil}}\},\\ \mathsf{cond}(\mathsf{P}(i^k)) &= \mathsf{cond}(i^k) \cup \{\emptyset \rhd \{\neg\mathsf{ins}_{k,\mathsf{nil}}, \mathsf{ins}_{k,i}\}\},\\ \mathsf{pre}(\mathsf{R}(i^k)) &= \mathsf{pre}(i^k) \cup \{\mathsf{ins}_{k,i}\},\\ \mathsf{cond}(\mathsf{R}(i^k)) &= \mathsf{cond}(i^k).\end{aligned}$$

Hence a programming action (P) is applicable on an empty line and programs an instruction on that line, while a repeat action (R) is applicable when an instruction already appears.

To define goto instructions we designate a subset of fluents $C \subseteq F$ as goto conditions. The set of goto instructions is $\mathcal{I}_{go} = \{\mathsf{goto}_{k',f} : 0 \leq k' \leq n, f \in C\}$, modelling control flow instructions $goto(k', !f)$. The compilation $\mathsf{goto}^k_{k',f}$ does not increment the program counter; rather, it is defined as

$$\mathsf{pre}(\mathsf{goto}^k_{k',f}) = \{\mathsf{pc}_k\},$$
$$\mathsf{cond}(\mathsf{goto}^k_{k',f}) = \{\emptyset \rhd \{\neg\mathsf{pc}_k\}, \{\neg f\} \rhd \{\mathsf{pc}_{k'}\}, \{f\} \rhd \{\mathsf{pc}_{k+1}\}\}.$$

Without loss of generality we assume $k \neq k'$, else the goto instruction may cause an infinite loop. In this work we found it more convenient to model a goto instruction $\mathsf{goto}^k_{k',f}$ such that execution jumps to line k' if fluent f is *false*, else execution continues on line $k + 1$.

We also have to make sure that the execution of the program does not finish on a line with an instruction. For this reason we define a fluent done which models that we are done programming. We also define the action end^k, $1 \leq k \leq n$, as $\mathsf{pre}(\mathsf{end}^k) = \{\mathsf{pc}_k\}$ and $\mathsf{cond}(\mathsf{end}^k) = \{\emptyset \rhd \{\mathsf{done}\}\}$. Implicitly, programming an end instruction on line k prevents other actions from being programmed on the same line, and since end^k does not cause execution to change lines, this causes execution to halt (technically, we can keep repeating the end action but this does not change the current state).

We are now ready to define the planning task $P'_n = \langle F'_n, A'_n, I'_n, G'_n\rangle$. The set of fluents is $F'_n = F \cup F_n \cup F_{ins} \cup \{\mathsf{done}\}$. The initial state is defined as

$$I'_n = I_n \cup \{\mathsf{ins}_{k,\mathsf{nil}} : 0 \leq k \leq n\} \cup \{\neg\mathsf{done}\} \cup \{\neg\mathsf{ins}_{k,i} : 0 \leq k \leq n, i \in A \cup \mathcal{I}_{go} \cup \{\mathsf{end}\}\},$$

and the goal is $G'_n = G \cup \{\mathsf{done}\}$. The set of actions is

$$A'_n = \{\mathsf{P}(a^k), \mathsf{R}(a^k) : a \in A, 0 \leq k < n\}$$
$$\cup \{\mathsf{P}(\mathsf{goto}^k_{k',f}), \mathsf{R}(\mathsf{goto}^k_{k',f}) : \mathsf{goto}_{k',f} \in \mathcal{I}_{go}, 0 \leq k < n\}$$
$$\cup \{\mathsf{P}(\mathsf{end}^k), \mathsf{R}(\mathsf{end}^k) : 1 \leq k \leq n\}.$$

For each line k, $0 \leq k \leq n$, the subset of fluents $\{\mathsf{ins}_{k,i} : i \in A \cup \mathcal{I}_{go} \cup \{\mathsf{nil}, \mathsf{end}\}\}$ is an *invariant*: only $\mathsf{ins}_{k,\mathsf{nil}}$ is true in the initial state and each action that deletes $\mathsf{ins}_{k,\mathsf{nil}}$ adds another fluent in the set. Moreover, no action deletes $\mathsf{ins}_{k,i}$ for $i \in A \cup \mathcal{I}_{go} \cup \{\mathsf{end}\}$. A plan π thus programs instructions on lines and can never delete an instruction once programmed.

Let Π be the program induced by fluents of type $\mathsf{ins}_{k,i}$ at the end of a plan π for P'_n. We show that π solves P'_n if and only if Π solves P. The actions of π simulate an execution of Π, updating fluents in F and the program counter and ending with an action $\mathsf{P}(\mathsf{end}^k)$, needed to add the goal fluent done. Action $\mathsf{P}(\mathsf{end}^k)$ is only applicable on an empty line and effectively halts execution, so for π to solve P'_n, the goal G has to hold when applying $\mathsf{P}(\mathsf{end}^k)$, i.e. after executing Π. Since π verifies that Π solves P, Π cannot contain infinite loops (a planner may spend effort, however, in attempting to verify an incorrect program).

```
prog_{0,inc(v1)} [1001]    main: 0.  inc(v1)
prog_{1,add(v0,v1)} [1001]       1.  add(v0,v1)
pgoto_{2,0,v1=5} [1001]          2.  goto(0,v1!=5)
repe_{0,inc(v1)} [1]
repe_{1,add(v0,v1)} [1]
rgoto_{2,0,v1=5} [1]
repe_{0,inc(v1)} [1]
repe_{1,add(v0,v1)} [1]
rgoto_{2,0,v1=5} [1]
repe_{0,inc(v1)} [1]
repe_{1,add(v0,v1)} [1]
rgoto_{2,0,v1=5} [1]
repe_{0,inc(v1)} [1]
repe_{1,add(v0,v1)} [1]
rgoto_{2,0,v1=5} [1]
end_3 [1]
```

Figure 3: Example plan generated for computing $\sum_{k=1}^{5} k$ using two variables $v0$ and $v1$, and the corresponding program.

As an example, Figure 3 shows a plan that solves the problem of computing $\sum_{k=1}^{5} k$. There are two variables $v0$ and $v1$, both initially 0, and the goal is having $v0 = 15$ and $v1 = 5$. We include the desired value $v1 = 5$ to allow the planner to use $v1$ as an auxiliary variable. The first three actions in the plan program two sequential instructions and a *goto* instruction. The remaining actions represent instructions that are not programmed but repeated and the *end* instruction. Figure 3 also shows the resulting program (left side of the Figure). The solution plan with 16 steps has a total cost of 3,016 and the program corresponding to this solution plan has 3 lines of code.

Procedures

In this section we show how to add procedures to the planning task P'_n from the previous section. Our current version does not include a call stack; instead, we only allow procedure calls in the main program and control always returns to the main program when a procedure finishes. In the future we plan to model arbitrary procedure calls using a call stack.

Let $P = \langle F, A, I, G\rangle$ be a planning task. We compile P into a planning task $P_{b,n} = \langle F_{b,n}, A_{b,n}, I_{b,n}, G_{b,n}\rangle$ with b the number of procedures and n the number of lines of each procedure. The set of instructions is $A \cup \mathcal{I}_{go} \cup \mathcal{I}_{pc} \cup \{\mathsf{nil}, \mathsf{end}\}$, where $\mathcal{I}_{pc} = \{\mathsf{call}_j : 1 \leq j < b\}$ is a set of procedure calls. We designate procedure 0 as the main program. Since $P_{b,n}$ is similar to the planning task $P'_n = \langle F'_n, A'_n, I'_n, G'_n\rangle$ from the previous section, we only describe the differences below.

The set $F_{b,n}$ contains the same fluents as F'_n with the following modifications:

- Fluents $\mathsf{pc}_{j,k}$ and $\mathsf{ins}_{j,k,i}$ include the procedure j, $0 \leq j < b$, in which a line or instruction appears.
- Fluents $\mathsf{ins}_{j,k,i}$ include instructions in the set $\mathcal{I}_{pc}$.
- There is a new fluent main which models that control is currently with the main program.

Fluent main and program counter $\mathsf{pc}_{0,0}$ are initially true, all procedure lines are empty, and the goal is $G_{b,n} = G \cup \{\mathsf{done}\}$. The actions from P'_n are modified as follows:

- Actions $a^{j,k}$ and $\mathsf{goto}^{j,k}_{k',f}$ include the procedure j, $0 \leq j < b$, and have an extra precondition main for $j = 0$.
- There is a new action $\mathsf{call}^{0,k}_j$ that calls procedure j, $1 \leq j < b$, on line k, $0 \leq k < n$, of the main program, defined as $\mathsf{pre}(\mathsf{call}^{0,k}_j) = \{\mathsf{pc}_{0,k}, \mathsf{main}\}$ and $\mathsf{cond}(\mathsf{call}^{0,k}_j) = \{\emptyset \rhd \{\neg\mathsf{pc}_{0,k}, \neg\mathsf{main}, \mathsf{pc}_{0,k+1}, \mathsf{pc}_{j,0}\}\}$.
- Action $\mathsf{end}^{j,k}$ is defined differently for $j = 0$ and $j > 0$:

$$\begin{aligned}\mathsf{pre}(\mathsf{end}^{0,k}) &= \{\mathsf{pc}_{0,k}, \mathsf{main}\},\\ \mathsf{cond}(\mathsf{end}^{0,k}) &= \{\emptyset \rhd \{\mathsf{done}\}\},\\ \mathsf{pre}(\mathsf{end}^{j,k}) &= \{\mathsf{pc}_{j,k}\}, j > 0,\\ \mathsf{cond}(\mathsf{end}^{j,k}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_{j,k}, \mathsf{main}\}\}, j > 0.\end{aligned}$$

The effect of $\mathsf{end}^{j,k}$, $j > 0$, is to end procedure j on line k and return control to the main. The set $A_{b,n}$ is defined as

$$\begin{aligned}A_{b,n} = &\{\mathsf{P}(a^{j,k}), \mathsf{R}(a^{j,k}) : a \in A, 0 \leq j < b, 0 \leq k < n\}\\ &\cup \{\mathsf{P}(\mathsf{goto}^{j,k}_{k',f}) : \mathsf{goto}_{k',f} \in \mathcal{I}_{go}, 0 \leq j < b, 0 \leq k < n\}\\ &\cup \{\mathsf{R}(\mathsf{goto}^{j,k}_{k',f}) : \mathsf{goto}_{k',f} \in \mathcal{I}_{go}, 0 \leq j < b, 0 \leq k < n\}\\ &\cup \{\mathsf{P}(\mathsf{call}^{0,k}_j), \mathsf{R}(\mathsf{call}^{0,k}_j) : \mathsf{call}_j \in \mathcal{I}_{pc}, 0 \leq k < n\}\\ &\cup \{\mathsf{P}(\mathsf{end}^{j,k}), \mathsf{R}(\mathsf{end}^{j,k}) : 0 \leq j < b, 1 \leq k \leq n\}.\end{aligned}$$

Similarly to the previous section, we show that a plan π solves $P_{b,n}$ if and only if the induced program Π solves P. At the end of π, the fluents $\mathsf{ins}_{j,k,i}$ describe a program, i.e. an assignment of instructions to the program lines of each procedure. To solve $P_{b,n}$, π has to simulate an execution of Π, including calls to procedures from the main program. Note that both calling procedures and ending procedures may be repeated due to actions $\mathsf{R}(\mathsf{call}^{0,k}_j)$ and $\mathsf{R}(\mathsf{end}^{j,k})$, in case a procedure is called multiple times on the same line of the main program. Plan π ends with a call to $\mathsf{P}(\mathsf{end}^{0,k})$ for some $k, 1 \leq k \leq n$.

Multiple Tests

When we include multiple tests, we essentially ask the planner to come up with a single program that solves a series of instances, each of which has a different initial state and goal condition. In many cases, this is impossible without control flow instructions. Consider the example in Figure 1. We cannot generate a single sequence of inc and add actions that computes the summatory for different values of m (if, however, we add instructions for multiplication and division, we can compute the summatory as $m(m+1)/2$).

To model multiple tests we assume that the input is a series of planning tasks $P^1 = \langle F, A, I^1, G^1\rangle, \ldots, P^T = \langle F, A, I^T, G^T\rangle$ that share fluents and actions but have different initial states and goal conditions. We define a planning task $P'_{b,n} = \langle F'_{b,n}, A'_{b,n}, I'_{b,n}, G'_{b,n}\rangle$ that models a programming task for solving $P^1, \ldots, P^T$ using procedures and goto instructions. Since $P'_{b,n}$ is similar to the planning task $P_{b,n}$ from the previous section, we describe the differences below.

The set of fluents is $F'_{b,n} = F_{b,n} \cup F_{test}$, where $F_{test} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ models the active test. Fluent test_1 is initially true while test_t is false for $2 \leq t \leq T$. The initial state on fluents in F is I^1, and the goal is $G'_{b,n} = G^T \cup \{\mathsf{done}\}$. The set $A'_{b,n}$ contains all actions in $A_{b,n}$, but we define action $\mathsf{end}^{0,k}_t$ differently for each test t, $1 \leq t \leq T$:

$$\begin{aligned}\mathsf{pre}(\mathsf{end}^{0,k}_t) &= G^t \cup \{\mathsf{pc}_{0,k}, \mathsf{main}, \mathsf{test}_t\}, t < T,\\ \mathsf{cond}(\mathsf{end}^{0,k}_t) &= \{\emptyset \rhd \{\neg\mathsf{pc}_{0,k}, \mathsf{pc}_{0,0}, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\}\\ &\quad \cup\ \{\{\neg l\} \rhd \{l\} : l \in I^{t+1}\}, t < T,\\ \mathsf{pre}(\mathsf{end}^{0,k}_T) &= \{\mathsf{pc}_{0,k}, \mathsf{main}, \mathsf{test}_T\},\\ \mathsf{cond}(\mathsf{end}^{0,k}_T) &= \{\emptyset \rhd \{\mathsf{done}\}\}.\end{aligned}$$

For $t < T$, action $\mathsf{end}^{0,k}_t$ is applicable when G^t and test_t hold, and the effect is resetting the program counter to $\mathsf{pc}_{0,0}$, incrementing the current test and setting fluents in F to their value in the initial state I^{t+1} of the next test. Action $\mathsf{end}^{0,k}_T$ is defined as before, and is needed to achieve the goal fluent done. As before, we add actions $\mathsf{P}(\mathsf{end}^{0,k}_t)$ and $\mathsf{R}(\mathsf{end}^{0,k}_t)$ to the set $A'_{b,n}$ for each t, $1 \leq t \leq T$, and k, $1 \leq k \leq n$. To solve $P'_{b,n}$, a plan π has to simulate an execution of the induced program Π on each of the T tests; hence π solves $P'_{b,n}$ if and only if Π solves P^t for each t, $1 \leq t \leq T$.

Evaluation

The evaluation is carried out in the following domains: The **summatory** domain models the computation of the summatory of m as described in the working example. Tests for this domain assign values to m in the range $[2, 11]$ and are solved by the 3-line program of Figure 1.

The **find**, **count** and **reverse** domains model programming tasks for list manipulation; respectively, finding the first occurrence of an element x in a list, counting the occurrences of an element x in a list and reversing the elements of a list. The tests for **find** include lists with sizes in the range $[15, 24]$, the element could be found in the list in all the tests so they can be solved using the 2-line program (0. $inc(i)$, 1. $goto(0, list[i]!{=}x)$). The tests for the **count** domain have list lengths $[5, 14]$ and can be solved by the 4-line program (0. $goto(2, list[i]!{=}x)$, 1. $inc(counter)$, 2. $dec(i)$, 3. $goto(0, i!{=}0)$). Tests for the **reverse** domain also range the list length in $[5, 14]$ and can be solved by program (0. $swap(head, tail)$, 1. $inc(head)$, 2. $dec(tail)$, 3. $goto(0, head!{=}tail \& head!{=}tail{+}1)$).

We believe that the above programming tasks are good benchmarks to illustrate the performance of our compilation when addressing planning tasks with multiple tests. However, we acknowledge that modelling programming tasks as classical planning tasks may not be the most efficient approach. In the future we plan to explore our compilation approach using other planning languages that more naturally handle numerical representations such as functional

	count[10]	diagonal[10]	find[10]	grid[10]	gripper[10]	lightbot[4]	maze[20]	reverse[10]	summatory[10]	unstack[10]
baseline	2.94	1.19	2.32	4.85	1.77	2.47	13.59	6.03	3.94	1.19
1-3	5.00	10.00	10.00	5.00	0	1.00	11.00	0	10.00	10.00
1-5	9.00	10.00	10.00	10.00	10.00	1.00	14.00	8.00	8.4	10.00
1-7	8.24	3.00	10.00	10.00	10.00	1.00	15.66	5.26	4.18	9.25
1-10	8.24	1.00	10.00	10.00	6.3	2.67	16.54	3.8	4.81	6.86
2-3	8.2	10.00	10.00	9.00	8.31	1.00	15.29	5.66	8.68	10.00
2-5	9.00	1.00	10.00	10.00	9.56	1.6	15.29	6.8	4.71	10.00
2-7	8.24	1.75	10.00	10.00	4.92	2.00	16.63	4.85	4.3	8.11
2-10	6.57	1.00	10.00	8.8	6.3	1.67	15.51	3.00	3.59	4.37

Table 1: Quality IPC scores obtained by our compilation with parameters b-n bounding the number of procedures $b = \{1, 2\}$ and their sizes $n = \{3, 5, 7, 10\}$. Fast Downward run on the original planning tasks serves as a baseline (with a number of program instructions equal to the plan length). The number of problems per domain is in square brackets.

STRIPS (Francés and Geffner 2015) and that are therefore better at modelling programming tasks.

The remaining domains are adaptations of existing planning domains. The **diagonal** and **grid** domains model navigation tasks in an $n \times n$ grid. In both cases, an agent starts at the bottom left corner, and has to reach the top right corner (diagonal) or an arbitrary location (grid). Tests for **diagonal** include grids with sizes in the range $[10, 19]$ and can be solved by the 3-line program (0. $right$, 1. up, 2. $goto(0, x!{=}x_{goal})$). In **grid** the grid size is in the range $[5, 14]$ and can be solved with the 6-line program (0. $goto(2, true)$, 1. $right$, 2. $goto(1, x!{=}x_{goal})$, 3. $goto(5, true)$ 4. up, 5. $goto(4, y!{=}y_{goal})$).

Finally, the **gripper** domain models the transport of n balls from a source to a destination room and the **unstack** domain models the task of unstacking a tower of n blocks. In **gripper**, the number of balls is in the range $[6, 15]$ and instances are solved by the program (0. $pickup$, 1. $move$, 2. $goto(0, atRobot{!=}roomB)$, 3. $drop$, 4. $goto(1, balls_{roomA}!{=}0)$). In **unstack** the program (0. $putdown$, 1. $unstack$, 2. $goto(0, \neg handempty)$) solves the tests which have a number of blocks in the range $[10, 19]$.

Despite the fact that the number of lines required to generate these programs is small, this does not mean that the explored search space is trivial, especially in the case of multiple tests. In this setting the task of simultaneously generating and validating programs for a set of tests often require plans that include tens of actions.

In all experiments, we run the forward-search classical planner Fast Downward (Helmert 2006) using the SEQ-SAT-LAMA-2011 setting (Richter and Westphal 2010) with time bound of 1,800 seconds and memory bound of 4GB RAM. To generate compact programs in this setting we define a cost structure that assigns high cost to programming an instruction and low cost to repeating the execution of a programmed instruction. More precisely, we let $c(\mathsf{P}(i)) = 1,001$ and $c(\mathsf{R}(i)) = 1$ for each instruction i.

Compacting Solutions to Classical Planning

The first experiment evaluates the capacity of our compilation to compact solutions to classical planning tasks. Benchmark problems in this experiment are single-test tasks where each problem represents one of the ten tests per domain described above. In addition, two extra domains are added, **maze** and **lightbot** that model programming tasks from the *Hour of Code* initiative (csedweek.org and code.org).

The size of a solution to a classical planning task is assessed here by the number of instructions in the program obtained from that solution. To serve as a baseline we ran Fast Downward on the original planning task, i.e., no gotos or procedures are allowed. Table 1 summarizes the results achieved by the baseline and our compilation with parameters $b \in \{1, 2\}$ and $n \in \{3, 5, 7, 10\}$ bounding the number of procedures and their sizes. The table reports the IPC quality score (Linares López, Jiménez, and Helmert 2013) with the quality of a plan given by the number of instructions of the program extracted from it. In the case of the baseline the number of program instructions always equals the plan length.

The difference in the scores of the baseline and the various configurations of our compilation becomes larger for larger problems. An illustrative example is the diagonal domain where any configuration of the compilation compresses solution plans to the 3-instruction program (0. $right$, 1. up, 2. $goto(0, x!{=}n)$) while the baseline plans keep growing with the grid size.

Solving Planning Tasks with Multiple Tests

The second experiment evaluates the performance of our approach on programming tasks with multiple tests. Each domain comprises now 10 problems such that the i^{th} problem includes the first i tests defined above, $1 \leq i \leq 10$. In this respect different tests encode different initial states and different goals. Solving this kind of planning tasks is impossible without handling control flow instructions therefore the baseline results are not included. **Maze** and **lightbot** are skipped here because no general solution solves different problems in these domains.

Table 2 reports the number of problems solved in each domain after compiling them into a planning task $P'_{b,n}$, where $b \in \{1, 2\}$ and $n \in \{3, 5, 7, 10\}$. Each cell in the table shows two numbers: problems solved when goto conditions can be any fluent/only fluents in $C \subseteq F$. The C subset comprises the dynamic fluents of the original planning tasks, i.e. fluents that can be either added or deleted by an original action, and fluents $equal(x, y)$ added to the compiled task in the form of derived predicates (Thiébaux, Hoffmann, and

	count[10]	diagonal[10]	find[10]	grid[10]	gripper[10]	reverse[10]	summatory[10]	unstack[10]
1-3	0/0	10/10	10/10	1/1	0/0	0/0	4/8	7/7
1-5	1/5	9/10	8/10	5/10	10/10	3/4	3/8	3/3
1-7	3/3	1/9	1/10	4/4	9/10	2/2	2/7	2/2
1-10	3/3	1/1	1/10	2/1	4/10	1/1	1/6	2/2
2-3	3/3	10/10	10/10	10/10	1/10	4/4	2/8	8/8
2-5	3/1	4/4	6/10	2/5	10/10	4/3	3/8	2/2
2-7	3/3	1/2	1/10	2/2	4/10	1/2	2/6	2/2
2-10	3/3	1/1	1/10	2/1	2/10	1/1	1/4	0/0

Table 2: Problems solved by our compilation with parameters $b = \{1, 2\}$ and $n = \{3, 5, 7, 10\}$ when goto conditions can be: any fluent/only fluents in $C \subseteq F$. Each domain comprises 10 problems such that the i^{th} problem includes the first i tests defined above.

Nebel 2005) to capture when two variables have the same value. Limiting conditions reduces the possible number of goto instructions and therefore the state space and branching factor of the planning task.

No unique parameter setup consistently works best because the optimal number of procedures and program lines depend on the particular program structure. A small number of lines is insufficient to find some programs, while a large number of lines increases the search space and consequently, the planning effort. Examples of the former are the gripper and reverse domains where configuration 1-3 cannot solve a single problem. Examples of the latter are the diagonal and find domains where configurations *-10 only solve one problem. Likewise procedures are useful in particular domains. In the grid domain the 2-3 configuration solves all problems while 1-3 only solves 1. The generated program is (0. up, 1. $goto(0, y!{=}y_{goal})$, 2. $p1$) where procedure $p1$ is defined as (0. $right$, 1. $goto(0, x!{=}x_{goal})$) . The same happens in the gripper domain where program (0. $pickup$, 1. p1, 2. $goto(0, balls_{roomA}! = 0)$) with p1 defined as (0. $move$, 1. $drop$, 2. $move$) also solves all problems.

Related Work

The contributions of this paper are related to work in different research areas of artificial intelligence.

- Compiling Domain-Specific Knowledge (DSK) into PDDL. Our programs and procedures can be viewed as DSK control knowledge. There is previous work on successful compilations of DSK into PDDL such as compiling HTNs into PDDL (Alford, Kuter, and Nau 2009), compiling LTL formulas (Cresswell and Coddington 2004) or compiling procedural domain control (Baier, Fritz, and McIlraith 2007). However, unlike previous work our approach automatically generates the DSK online.
- There is a large body of work on *learning for planning* (Zimmerman and Kambhampati 2003; Jiménez et al. 2012), with macro-action learning being one of the most successful approaches (Botea et al. 2005; Coles and Smith 2007). In the absence of control flow our procedures can be understood as parameter-free macro-actions. A natural future direction is then to study how to reuse our programs and procedures in similar planning tasks.
- *Generalized planning* also aims at synthesizing general strategies that solve a set of planning problems (Levesque 2005; Hu and De Giacomo 2011). A common approach is planning for the problems in the set and incrementally merging the solutions (Winner and Veloso 2003; Srivastava et al. 2011). A key issue here is handling a compact representation of the solutions that can effectively achieve the generalization. Our work exploits *gotos* and procedure calls to represent such general strategies; in addition, programs are built in a single classical planning episode by assigning different costs for programming and executing instructions and exploiting the cost-bound search of state-of-the-art planners.
- Also related is work on deriving Finite-State Machines (FSMs) (Bonet, Palacios, and Geffner 2010), which uses a compilation that interleaves building a FSM with verifying that the FSM solves different problems. Procedureless programs can be viewed as a special case of FSMs in which branching only occurs as a result of goto instructions. However, a single FSM cannot represent multiple calls to a procedure without a significant increase in the number of machine states. The generation of the FSMs is also different since it starts from a partially observable planning model and uses a conformant into classical planning compilation (Palacios and Geffner 2009).
- *Automatic program generation*. Like in *programming by example* (Gulwani 2012) our approach attempts to generate programs starting from input-output examples. In contrast to earlier work, our approach is compilation-based and exploits the model of the instructions with off-the-shelf planning languages and algorithms.
- Our work is also related to knowledge-based programs for planning (Lang and Zanuttini 2012) which, just like our compilation, incorporate selection and repetition constructs into plans. No procedures are allowed, however, and existing work is purely theoretical, analyzing the computational complexity of decision problems related to knowledge-based programs in planning.

Conclusion

In this paper we introduce a novel compilation for incorporating control flow and procedures into classical planning. We also extend the compilation to include multiple unit tests,

i.e. multiple instances of related, similar problems. The result of the compilation is a single classical planning task that can be solved using an off-the-shelf classical planner to generate compact solutions to the original task in the form of programs. In experiments we show that the compilation often enables us to generate solutions that are much more compact than a simple sequence of actions, as well as generalized solutions that can solve a series of related planning tasks.

As is common in works in automatic programming, the optimal parameter settings depend on the task to be solved. A sufficient program size is key for the performance of the approach and too many procedures with too many lines result in tasks too large to be solved by an off-the-shelf planner. Two ways to address this issue would be to implement the successor rules of our compilation explicitly into a planner or iteratively increment the compilation bounds, like in SAT-based planning (Rintanen 2012).

The scalability of our compilation is also an issue. On the positive side, a small number of tests is often sufficient to achieve a generalized solution. For instance, no configuration solved all the summatory problems but solving a problem that only includes two tests already produces a general program. In the future we plan to explore our compilation approach using a planning language that more naturally handle numerical representations. In particular we plan to use functional STRIPS (Francés and Geffner 2015) to improve the scalability of our approach.

Acknowledgment

Sergio Jiménez is partially supported by the *Juan de la Cierva* program funded by the Spanish government.

References

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *International Joint Conference on Artificial Intelligence*.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *International Conference on Automated Planning and Scheduling*.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic Derivation of Finite-State Machines for Behavior Control. In *AAAI*.

Botea, A.; Enzenberger, M.; Mller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

Brameier, M. 2004. *On Linear Genetic Programming*. Ph.D. Dissertation, Fachbereich Informatik, Universit"at Dortmund, Germany.

Coles, A., and Smith, A. 2007. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research* 28:119–156.

Cresswell, S., and Coddington, A. M. 2004. Compilation of ltl goal formulas into pddl. In *European Conference on Artificial Intelligence*.

DeMillo, R. A.; Eisenstat, S. C.; and Lipton, R. J. 1980. Space-Time Trade-Offs in Structured Programming: An Improved Combinatorial Embedding Theorem. *J. ACM* 27(1):123–127.

Francés, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *International Conference on Automated Planning and Scheduling*.

Gulwani, S. 2012. Synthesis from examples: Interaction models and algorithms. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *International Joint Conference on Artificial Intelligence*, 918–923.

Jiménez, S.; De La Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27(04):433–467.

Lang, J., and Zanuttini, B. 2012. European conference on artificial intelligence. In *ECAI*.

Levesque, H. J. 2005. Planning with loops. In *International Joint Conference on Artificial Intelligence*, 509–515.

Linares López, C.; Jiménez, S.; and Helmert, M. 2013. Automating the evaluation of planning systems. *AI Communications* 26(4):331–354.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* 35:623–675.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence Journal* 193:45–86.

Srivastava, S.; Immerman, N.; Zilberstein, S.; and Zhang, T. 2011. Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of {PDDL} axioms. *Artificial Intelligence Journal* 168(12):38 – 69.

Winner, E., and Veloso, M. 2003. Distill: Towards learning domain-specific planners by example. In *International Conference on Machine Learning*.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* 24(2):73.