P²: A Baseline Approach to Planning with Control Structures and Programs

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
rpetrick@inf.ed.ac.uk

Abstract

Many planners model planning domains with “primitive actions,” where action preconditions are represented by sets of simple tests about the state of domain fluents, and action effects are described as updates to these fluents. Queries and updates are typically combined in only very limited ways, for instance using logical operators and quantification. By comparison, formalisms like Golog permit “complex actions,” with control structures like if-else blocks and while loops, and view actions as programs. In this paper we explore the idea of planning directly with complex actions and programs. We describe the structure of a simple planner based on undirected search, that generates plans by simulating the execution of action programs before they are added to a plan. An initial evaluation compares this approach against a classical heuristic planner using a domain whose program structures have been compiled into ordinary PDDL actions. Initial results illustrate that in certain domains, planning directly with programs can lead to a significant performance improvement. This work offers a baseline planner to compare against alternate approaches to planning with programs.

Introduction and Motivation

A recent trend in modern planning research has focused on the problem of planning with complex expressions, control structures, and programs—representations that are more complicated compared with traditional formalisms based on PDDL (McDermott 1998), the standard language for modelling planning domains. While recent additions to PDDL (e.g., constraints, preferences, durative actions, and numerical fluents) have extended its expressiveness, PDDL remains inherently STRIPS-like (Fikes and Nilsson 1971) in its structure. Primitive actions form the basis of a domain specification: action preconditions are defined by simple tests about the state of domain fluents, and action effects capture the (conditional) changes made to these fluents. Fluent tests and updates are often combined in very limited ways, using standard logical connectives and quantification.

By comparison, attempts to plan with complex actions admit actions with control flow blocks (e.g., sequence, iteration, and conditionals) and other procedural operators inspired by imperative programming languages. In practice, complex actions operate more like programs and are often distinct from primitive actions, with the latter defining the fluent-level state changes and the former acting as a wrapper around sets of primitive actions. While complex actions add more flexibility to the expressiveness of the representation language, most planners cannot directly construct plans with such actions. In this paper we present a simple planner that is capable of manipulating such structures.

The idea of mixing procedural constructs with planning is not new. For instance, much work has addressed the problem of automatically constructing macro operators, which combine useful sequences of actions in an attempt to improve plan generation efficiency (e.g., (Botea, Müller, and Schaeffer 2007; Coles and Smith 2007)). HTN planning (e.g., (Sacerdoti 1975; Nau et al. 2003)) also has a procedural flavour: HTN domains abstract the action space into high-level tasks and methods for decomposing those tasks into more primitive subtasks, with the lowest-level subtasks corresponding to ordinary planning operators. More formally, Levesque (1996) generalizes the planning problem in terms of a universal programming language $R$, which includes sequence, branch, and loop constructs operating over actions. Levesque (2005) also uses a variant of $R$ to investigate the problem of automatically generating plans with loops.

More closely related to the focus of this paper, one of the most popular approaches to planning with programs has been to compile complex actions into primitive actions, written in ordinary PDDL, which can then be used in conjunction with ordinary off-the-shelf planners. For instance, McIlraith and Fadel (2002) formalize an approach that transforms certain classes of programs written in Golog (Levesque et al. 1997)—a high-level programming language based on the situation calculus (McCarthy and Hayes 1969; Reiter 2001)—into PDDL. These programs allow procedural structures like action sequencing, if-else blocks, and a bounded while loop, among others. Baier and McIlraith (2006) build on this work by considering Golog programs with sensing actions (i.e., knowledge-producing actions that observe the state of the world without necessarily changing it) and translate these domains into a form usable by planners that support sensing actions, but not complex actions. Similarly, Baier, Fritz, and McIlraith (2007) compile procedural domain control knowledge into PDDL domains, modelled in a language based on Golog.

There are two potential drawbacks of the compilation approaches. First, new fluents and actions are generally intro-
duced into the resulting planning domain as a consequence of the compilation process, thereby increasing the size of the state space. Second, the rich control knowledge explicitly represented in structures like loops is discarded during compilation. Instead, the behaviour of such structures must be “rediscovered” through search, by appropriately guiding the planner’s search through the resulting primitive actions, to mimic the effects of the original complex actions. While modern planners can often cope with the first drawback, the second is more problematic. For instance, the number of states a planner must visit can quickly become large when loops are permitted. As we will see, even the best heuristic planners do not always work well with compiled domains.

As an alternative to the compilation approaches, we explore the notion of planning directly with complex actions and programs, by simulating their execution within action blocks. We describe the implementation of a simple planner that supports a set of procedural constructs, including if-else blocks and unbounded while loops. During plan construction our planner simulates the application of an action by “running” its precondition or effects program, in a manner not unlike Golog. While we do not aim to be competitive with off-the-shelf planners in terms of speed (e.g., our initial implementation uses blind search), our planner nevertheless shows good performance compared against Metric-FF (Hoffmann 2003) on a toy domain, and provides a useful baseline to compare against alternative approaches. Overall, this work is a first step in a research agenda aimed at designing new planners that can search and plan directly with procedural control structures.

Example: Compiling while Loops into PDDL

As motivation for this work, consider the toy action in Figure 1. This action is similar in form to a primitive action, but includes a while loop. The intent here is to “loop while i is less than or equal to the value of the function size(?d),” adding i to the value of count and 1 to each time through the loop. Although PDDL does not directly support actions with while loops, we can transform this action into a valid PDDL form that achieves a similar effect.1

Figure 2 shows three PDDL actions that encode the behaviour of the action in Figure 1: processDataset models the effects of the original action up to the start of the while loop, processDataset-inLoop simulates one iteration through the loop, and processDataset-endLoop encodes the effects following the loop. The first action contains the preconditions of the original action. The new predicate context-loop acts as a guard, controlling access to the body of the compiled loop. A second new predicate, context-loop-params, tracks the parameters of the original action. (If the domain contained additional actions their preconditions would also be updated with references to context-loop to prohibit their application during the execution of this loop.)

In this case, the correct behaviour of the compiled actions results from the planner’s ability to order these actions appropriately during its search. For instance, once processDataset has been applied, the only action subsequently permitted according to its preconditions is processDataset-inLoop, which can be continually applied until the loop conditions are false. At this point the only permissible action is processDataset-endLoop, which completes the execution of the original action.

Although this example is extremely simple, we note two potential drawbacks. First, two actions and two predicates are added to the domain, increasing the size of the state space. Second, and more worrying, is the prospect that each iteration of the while loop is now an action instance. Thus, a loop with 100 iterations requires a sequence of 102 actions, and the rich control knowledge explicitly represented by the original while loop must be implicitly rediscovered by the backend PDDL planner during preprocessing and search.

Representing Actions as Programs

As an alternative to the compilation approach, we describe the structure of a simple planner called ProgPlan (abbreviated P2), which supports actions with program constructs, and simulates their execution during plan search.

Symbols We assume a planning scenario whose symbols are defined as in an ordinary PDDL planning problem. Thus, we include a set of fluent symbols representing the properties of the domain that can change as a result of action, including both predicates and functions. (We also allow equality and standard numerical relations like <.) A set of constants denoting the objects in the domain is also defined.

The representation language used by P2 is built around the notion of an expression and a program.

Expressions An expression in our representation is similar to the form of the preconditions used by ordinary classical, deterministic planners (e.g., the preconditions in Figure 2). Expressions can use the connectives and, or, not, exists, and forall, plus arithmetic expressions and fluent tests about the value of relations and functions.

We define a complex expression as follows:

Example: Compiling while Loops into PDDL

As motivation for this work, consider the toy action in Figure 1. This action is similar in form to a primitive action, but includes a while loop. The intent here is to “loop while i is less than or equal to the value of the function size(?d),” adding i to the value of count and 1 to each time through the loop. Although PDDL does not directly support actions with while loops, we can transform this action into a valid PDDL form that achieves a similar effect.1

Figure 2 shows three PDDL actions that encode the behaviour of the action in Figure 1: processDataset models the effects of the original action up to the start of the while loop, processDataset-inLoop simulates one iteration through the loop, and processDataset-endLoop encodes the effects following the loop. The first action contains the preconditions of the original action. The new predicate context-loop acts as a guard, controlling access to the body of the compiled loop. A second new predicate, context-loop-params, tracks the parameters of the original action. (If the domain contained additional actions their preconditions would also be updated with references to context-loop to prohibit their application during the execution of this loop.)

In this case, the correct behaviour of the compiled actions results from the planner’s ability to order these actions appropriately during its search. For instance, once processDataset has been applied, the only action subsequently permitted according to its preconditions is processDataset-inLoop, which can be continually applied until the loop conditions are false. At this point the only permissible action is processDataset-endLoop, which completes the execution of the original action.

Although this example is extremely simple, we note two potential drawbacks. First, two actions and two predicates are added to the domain, increasing the size of the state space. Second, and more worrying, is the prospect that each iteration of the while loop is now an action instance. Thus, a loop with 100 iterations requires a sequence of 102 actions, and the rich control knowledge explicitly represented by the original while loop must be implicitly rediscovered by the backend PDDL planner during preprocessing and search.

Representing Actions as Programs

As an alternative to the compilation approach, we describe the structure of a simple planner called ProgPlan (abbreviated P2), which supports actions with program constructs, and simulates their execution during plan search.

Symbols We assume a planning scenario whose symbols are defined as in an ordinary PDDL planning problem. Thus, we include a set of fluent symbols representing the properties of the domain that can change as a result of action, including both predicates and functions. (We also allow equality and standard numerical relations like <.) A set of constants denoting the objects in the domain is also defined.

The representation language used by P2 is built around the notion of an expression and a program.

Expressions An expression in our representation is similar to the form of the preconditions used by ordinary classical, deterministic planners (e.g., the preconditions in Figure 2). Expressions can use the connectives and, or, not, exists, and forall, plus arithmetic expressions and fluent tests about the value of relations and functions.

We define a complex expression as follows:

---

1We have implemented a compiler for transforming actions with simple program structures into PDDL, in order to compare our approach against such compilation methods. The example in Figure 2 was generated by our compiler and is characteristic of the kinds of actions we can produce. In general, we use the ADL subset of PDDL but our example here also requires numerical fluents.
Figure 2: Compiled PDDL actions simulating a while loop

expression ::= expression and expression | expression or expression | not (expression) | (expression) | forall (parameters) expression | exists (parameters) expression | arithmetic-expression | fluent-test.

We note that expressions “ground out” with ordinary arithmetic expressions (which include a large set of expressions from the C programming language) and fluent queries.

Programs A program is a set of control structures which operate over fluent updates and expressions.

program ::= program ; program | if expression then program else program endIF | while expression do program endwhile | forall(parameters) program endForall | exists(parameters) expression then program else program endExists | arithmetic-assignment | fluent-update | nil.

We follow ordinary program syntax in using ; as the standard sequence operator for chaining program statements together. The if-else block is a standard conditional test which allows a choice as to which program should be executed, depending on the outcome of the test (the first program on success, the second on failure). Similarly, while is a standard while loop that repeats the execution of a program as long as the test expression is true. The forall and exists control structures introduce a special type of “quantified” program statement. forall is a loop that repeatedly executes a program; each time through the loop a new binding from the set of domain constants is chosen and assigned to the specified parameters. exists is a conditional nondeterministic choice statement that attempts to find a binding for the specified parameters so that the test expression evaluates as true. If found, the first program block is executed; otherwise, the second program block is executed. In both types of quantified structures, the “bound” parameters may be used in the body of the control block. Finally, a program can also be an empty program nil, an ordinary fluent update, or an arithmetic assignment statement. For arithmetic assignments, we not only allow simple calculations whose results are assigned to functions but also a rich selection of C-style numerical expressions.

Actions Actions are structured in a similar way to ordinary actions, with names, parameters, preconditions, and effects. Parameters are ordinary action variables which are bound to produce action instances. (Such variables may occur in an action’s preconditions or effects.) In our case, preconditions are defined to be expressions and effects are programs, i.e.,

```
(action A (parameters)
  preconditions: expression
  effects: program
endAction)
```

Action preconditions and effects have the same intuitive meaning as ordinary planning actions: during plan construction an action’s preconditions must be true before it’s effects can be applied. In particular, we do not distinguish between “primitive” and “complex” actions in our representation.2

For instance, Table 3 shows a set of actions taken from an e-mail application domain, which give a flavour of the types of actions we can model with our representation. The read(m) action marks a particular message m as “read”, provided it is in the user’s inbox. In this case there are two effects: a fluent update marking m as read, and a second update increasing the count of the function numread which tracks the number of messages marked as read. The markAllRead action has the effect of marking all known messages in the user’s inbox as read. In this case, the effects are modelled with an outer forall block and an inner if-then block, which tests each message and ensures only those messages in the user’s inbox are appropriately marked. The functions numread and numunread denote the number of read and unread messages, respectively. The findUnread action uses the exists structure to find a message in the user’s inbox which has not been read and sets the function current as this message. In the case no such message exists, current is set to a special constant none. Finally, the

---

2We are currently adding a “procedure call” to our representation, allowing one action to execute another action. This construct will let us specify actions with more complex control flow.
action markRead(?m)
  precondition: inbox(?m)
  effect:
    read(?m); numread = numread + 1
endAction

action markAllRead
  precondition: true
  effect:
    numread = 0;
    numunread = 0;
    forall(?m)
      if inbox(?m) then
        read(?m);
        numread = numread + 1
      endIf
    endForall
endAction

action findUnread
  precondition: true
  effect:
    exists(?m)
      (inbox(?m) and not(read(?m)))
    then current = ?m
    else current = none
  endExists
endAction

action countRange
  precondition: from <= to
  effect:
    count = 0;
    skipped = 0;
    i = from;
    while i <= to
      if read(msg(i)) then
        count = count + 1
      else
        skipped = skipped + 1
      endIf
      i = i + 1
    endWhile
endAction

Figure 3: Actions from an e-mail application domain

countRange action is used to count the number of messages in a particular range that are marked as read. The function msg(i) maps a message number i to a particular message. The while loop ensures we only consider the range defined by the functions from and to. The function count tracks the number of messages that are counted in the range, while skipped denotes the number of messages in the range that we ignore. The expression read(msg(i)) illustrates a permissible fluent test, with a nested function as an argument.

Planning by Simulating Program Execution

We now turn our attention to evaluating expressions and programs with respect to our representation.

Expression evaluation A state is a snapshot of the values of all fluents defined in a domain. For expressions, we define a procedure EvalExpr(e, S) which evaluates whether a compound expression e is true at a state S by recursively unwinding the expression down to its component parts (i.e., fluent tests), which are then evaluated at S. A special function EvalArithExpr(e, S) evaluates arithmetic expressions by reducing all arithmetic expressions (which may contain functions) to a number. Following C programming style, an arithmetic expression is “true” if it evaluates to a non-zero value. We have the following evaluation function.

Definition 1 Let S be a state let e, e1, and e2 be expressions.

\[
\text{EvalExpr}(e, S) = \begin{cases} 
true & \text{if } e \text{ has the form } “e_1 \text{ and } e_2” \text{ and } \text{EvalExpr}(e_1, S) = true \text{ and } \text{EvalExpr}(e_2, S) = true, \\
true & \text{if } e \text{ has the form } “e_1 \text{ or } e_2” \text{ and } \text{EvalExpr}(e_1, S) = true \text{ or } \text{EvalExpr}(e_2, S) = true, \\
false & \text{if } e \text{ has the form } “\neg e_1” \text{ and } \text{EvalExpr}(e_1, S) = false, \\
true & \text{if } e \text{ has the form } “\exists \vec{x}/\vec{c} \ e_1” \text{ and } \text{EvalExpr}(e_1(\vec{x}/\vec{c}), S) = true \text{ for every substitution } \vec{c} \text{ of } \vec{x} \text{ in } e_1, \\
true & \text{if } e \text{ has the form } “\forall \vec{x} \ e_1” \text{ and } \text{EvalExpr}(e_1(\vec{x}/\vec{c}), S) = true \text{ for some substitution } \vec{c} \text{ of } \vec{x} \text{ in } e_1, \\
true & \text{if } e \text{ is an arithmetic expression and } \text{EvalArithExpr}(e, S) \neq 0, \\
true & \text{if } e \text{ is a fluent query and } \text{IA}(e, S) = true. \\
\text{false} & \text{otherwise.}
\end{cases}
\]

EvalExpr recursively deconstructs a complex expression into simpler components. In (1) – (4), the standard and, or, and not connectives, plus expression precedence, are evaluated in a straightforward way. In (5) and (6), EvalExpr considers possible substitutions of the quantified parameters. The notation e1(\vec{x}/\vec{c}) indicates that all occurrences of \vec{x} in e1 should be syntactically replaced with \vec{c}, where \vec{c} is taken from the set of defined constants. (I.e., the expression is rewritten before it is recursively evaluated.) In (7), the special function EvalArithExpr evaluates an arithmetic expression against a state S, by attempting to reduce the expression to a number. (Space prohibits us from describing this process in detail.) We follow C programming style here and consider an arithmetic expression to be “true” at S if it evaluates to a non-zero value. In (8), the truth of a fluent query e is determined by a function called IA which checks the fluent’s value in state S. IA is also responsible for evaluating queries with references to nested functional fluents.

Program simulation In traditional planning, a set of ordinary fluent updates, when applied to a state S, transforms S to produce a new state S’. We extend this notion to programs by simulating the run of a given program at a state S. All fluent updates that arise during program execution are applied to the current state, generating a sequence of new states. (Each fluent update could produce a new state.) Upon program termination, we disregard any “intermediate” states and return the final resulting state S’.

A procedure called RunProg(p, S) simulates the execution of a program p starting in a state S, and returns a state S’ on completion of the program run. In general,
proc ProgPlan($S, G, A, P$)
  if EvalExpr($G, S$) = true then return $P$
  else if
    choose($a \in A$) : EvalExpr(pre($a$), $S$) = true then
      $S' = \text{RunProg(eff($a$), $S$)}$
    return ProgPlan($S'$, $G, A, P + a$)
  else return fail
endIf
endProc

Figure 4: Pseudocode for the $P^2$ planning algorithm

RunProg operates as a program interpreter, stepping its way through a given program. A program counter tracks the current program statement being executed, which is updated after its completion. Depending on the type of procedural construct under evaluation, the interpreter runs a small control program to evaluate its outcome. For instance, evaluating a sequence construct involves running two programs in turn, with the second program executing from the state resulting from the execution of the first program, i.e., RunProg($p_1$ and $p_2$, $S$) := RunProg($p_2$, RunProg($p_1$, $S$)). For a while loop, the interpreter runs the control program

$\text{RunProg(while } e \text{ do } p \text{ endwhile, } S):=\$
$\text{while } \text{EvalExpr}(e, S) = \text{true do}$
$\quad S = \text{RunProg}(p, S)$
$\text{endWhile ; return } S$.

Here, EvalExpr evaluates the truth of expression $e$ in each iteration of the loop. (The underlined control structures are part of the interpreter’s control program for simulating the while loop.) $S$ is updated each time through the loop and the final $S$ is returned on completion. One important danger of this approach is that programs aren’t guaranteed to terminate: since we simulate actual programs, we also inherit the problems of ordinary program design, including the possibility of infinite loops. Similar control programs are defined for the other control structures in our representation language. When RunProg encounters a fluent update, it applies it to the existing state an an ordinary update.

Planning A planning problem is specified by a set of actions $\mathcal{A}$, an initial state $S$, and a set of goal conditions $G$. The initial state can be any state (as in ordinary PDDL) and a goal is any expression. Figure 4 shows the pseudocode describing the main operation of our program planner, $P^2$. Plans are built in a simple forward-chaining manner, starting from the initial state. The planning algorithm attempts to grow a plan by searching over the space of applicable actions and choosing a ground action instance $a$ whose preconditions $\text{pre}(a)$ (an expression) are satisfied in the current state $S$ according to EvalExpr. If such an action exists, its effects $\text{eff}(a)$ (a program) are applied to $S$ by RunProg to produce a new state $S'$. Action $a$ is concatenated to the end of the current plan and planning continues until a state is reached where the goal is satisfied, or the plan cannot be extended.

Initial Evaluation

We have implemented an initial version of our planner in C++ as a simple forward-chaining planner using undirected

### Table 1: Running time in seconds on the example domain

<table>
<thead>
<tr>
<th>$n$</th>
<th>Test-1</th>
<th>Test-2</th>
<th>Test-3</th>
<th>Test-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>1000</td>
<td>0.07</td>
<td>0.11</td>
<td>0.14</td>
<td>0.21</td>
</tr>
<tr>
<td>10000</td>
<td>0.71</td>
<td>0.90</td>
<td>1.20</td>
<td>1.94</td>
</tr>
<tr>
<td>100000</td>
<td>7.02</td>
<td>8.82</td>
<td>13.08</td>
<td>19.30</td>
</tr>
</tbody>
</table>

### Table 2: Running time in seconds of benchmark tests on while loop programs of $n$ iterations and length 100 plans

<table>
<thead>
<tr>
<th>$n$</th>
<th>Test-1</th>
<th>Test-2</th>
<th>Test-3</th>
<th>Test-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>1000</td>
<td>0.07</td>
<td>0.11</td>
<td>0.14</td>
<td>0.21</td>
</tr>
<tr>
<td>10000</td>
<td>0.71</td>
<td>0.90</td>
<td>1.20</td>
<td>1.94</td>
</tr>
<tr>
<td>100000</td>
<td>7.02</td>
<td>8.82</td>
<td>13.08</td>
<td>19.30</td>
</tr>
</tbody>
</table>

Table 1: Running time in seconds on the example domain

Table 2: Running time in seconds of benchmark tests on while loop programs of $n$ iterations and length 100 plans depth-first search and breadth-first search.

Table 3: Running time in seconds of benchmark tests on while loop programs of $n$ iterations and length 100 plans

Table 4: Running time in seconds of benchmark tests on while loop programs of $n$ iterations and length 100 plans depth-first search and breadth-first search.

3 The source code for $P^2$ is available from http://homepages.inf.ed.ac.uk/rpetrick/research/p2.
Our current planner is not meant to be competitive with current off-the-shelf planners. Instead, it is a first step in an ongoing research programme aimed at developing practical planners that can operate in more complex state spaces. As such, we offer our present planner to the community as a baseline tool for evaluating alternative approaches and advancing research into planning with programs.

Acknowledgements
This work was partly funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

References