P$^2$: A Baseline Approach to Planning with Control Structures and Programs

Ron Petrick
School of Informatics
University of Edinburgh
Edinburgh, Scotland, United Kingdom

ICAPS 2009 Workshop on Generalized Planning: Macros, Loops, Domain Control

Thessaloniki, Greece
20 September 2009
Motivation

• “Traditional” planning is based on generating plans with **primitive actions**:
  – Action preconditions are specified as simple tests about the state of domain fluents.
  – Action effects capture the (conditional) fluent changes.
  – Fluent tests and updates can typically be combined in only very limited ways.

• **Complex actions** admit actions with control flow blocks, such as sequence, iteration, conditionals, and other procedural operators inspired by imperative programming languages.

• Planners that plan directly using complex actions?
Mixing procedure and planning is not new

- Conditional action effects in **ADL** (Pednault 1989).
- **Macro operator** construction  
  (Botea, Müller, & Schaeffer 2007, Coles & Smith 2007).
- **Action abstraction/decomposition in HTN planning**  
  (Sacerdoti 1975, Nau *et al.* 2003).
- **Generalization of planning in a universal programming language** $\mathcal{R}$  
  (sequences, branches, and loops over actions) (Levesque 1996).
- **Automatic generation of plans with loops**  

$\Rightarrow$ **Compilation** of complex actions into ordinary (PDDL) actions usable with standard off-the-shelf planners

- Golog programs (McIlraith & Fadel 2002, Claßen *et al.* 2007)
- Golog + sensing (Baier & McIlraith 2006)
- Procedural control knowledge (Baier, Fritz, & McIlraith 2007).
Overview of the talk

• We describe a simple “baseline” planner that works directly with program structures, by simulating their execution during plan generation, as an alternative to compilation methods.

1. Example: compilation and its drawbacks

2. Representing actions as programs

3. \( P^2 \): Planning by simulating program execution

4. Initial evaluation

5. Discussion and conclusions.
Example: an action with a **while** loop

```
action processDataset(?d)
   precondition:
       dataset(?d) and
       not(processedDataset(?d))
   effect:
       i = 1 ;
       while (i <= size(?d))
           count = count + i ;
           i = i + 1
       endwhile ;
   processedDataset(?d)
endAction
```
Example: compiled PDDL actions

 action processDataset(?d)
  precondition:
    dataset(?d) and
    not(processedDataset(?d))
  effect:
    i = 1 ;
    while (i <= size(?d))
      count = count + i ;
      i = i + 1
    endwhile ;
  endAction
Potential drawbacks of compilation

• New fluents and actions may be introduced into the planning domain
  ⇒ Increases the size of the state space

• Control knowledge in procedural structures is compiled away; the resulting actions mimic the effects of the original complex action.
  ⇒ Control knowledge must be “rediscovered” by search

• Can we build a planner that works directly with procedural structures?
Representing actions as programs

• Planning with primitive actions

• Planning by simulating program execution

⇒ Planning with Programs (P²)
Symbols and states

• Symbols are specified as in ordinary PDDL planning domains.
  – Fluents (predicates and functions) represent properties of the domain.
  – Constants denote domain objects.

• States: snapshots of the values of all fluents.

• Classical planning problem: complete world states.

• Actions are state transforming, but extended to include more complex procedural control structures.

• Built around the idea of expressions and programs.
Expressions

\[ \text{expression} ::= \text{expression and expression} \mid \text{expression or expression} \mid \text{not} (\text{expression}) \mid (\text{expression}) \mid \text{forall}(\text{parameters})\text{ expression} \mid \text{exists}(\text{parameters})\text{ expression} \mid \text{arithmetic-expression} \mid \text{fluent-test}. \]

• Expressions “ground out” with ordinary arithmetic expressions (which can include a large set of C programming language expressions) and fluent queries.
Programs

\[
\text{program ::= program ; program | if expression then program else program endIF | while expression do program endWhile | forall(parameters) program endForall | exists(parameters) expression then program else program endExists | arithmetic-assignment | fluent-update | nil.}
\]

- Programs are control structures which operate over expressions, arithmetic assignments (e.g., \(x = y \times 42\)) and fluent updates.
Actions

\[
\text{action } A \left( \text{parameters} \right) \\
\text{preconditions: expression} \\
\text{effects: program} \\
\text{endAction}
\]

- Action preconditions and effects have the same intuitive meaning as ordinary planning action: an action’s preconditions must evaluate to true before it’s effects can be applied.
- We do not distinguish between “primitive” and “complex” actions.
Example

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

An expression $e$ is true at a state $S$, i.e., $\text{EvalExpr}(e, S) = \text{true}$, if

1. $e$ has the form “$e_1 \text{ and } e_2$” and $\text{EvalExpr}(e_1, S) = \text{true}$ and $\text{EvalExpr}(e_2, S) = \text{true}$,
2. $e$ has the form “$e_1 \text{ or } e_2$” and $\text{EvalExpr}(e_1, S) = \text{true}$ or $\text{EvalExpr}(e_2, S) = \text{true}$,
3. $e$ has the form “$\text{not}(e_1)$” and $\text{EvalExpr}(e_1, S) = \text{false}$,
4. $e$ has the form “$(e_1)$” and $\text{EvalExpr}(e_1, S) = \text{true}$,
5. $e$ has the form “$\text{forall}(\vec{x}) e_1$” and $\text{EvalExpr}(e_1(\vec{x}/\vec{c}), S) = \text{true}$ for every substitution $\vec{c}$ of $\vec{x}$ in $e_1$,
6. $e$ has the form “$\text{exists}(\vec{x}) e_1$” and $\text{EvalExpr}(e_1(\vec{x}/\vec{c}), S) = \text{true}$ for some substitution $\vec{c}$ of $\vec{x}$ in $e_1$,
7. $e$ is an arithmetic expression and $\text{EvalArithExpr}(e, S) \neq 0$,
8. $e$ is a fluent query and $\text{IA}(e, S) = \text{true}$.

Otherwise, $\text{EvalExpr}(e, S) = \text{false}$. 
Program simulation

\( \text{RunProg}(p, S) \): simulate the execution of program \( p \) starting in state \( S \), and return the resulting state on completion.

\[
\text{RunProg}(p_1 \ \text{and} \ p_2, S) :=
\text{RunProg}(p_2, \text{RunProg}(p_1, S))
\]

\[
\cdots
\]

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

\[
\cdots
\]

- Each control structure is interpreted by its own control program.
Program simulation...

• Ordinary fluent updates and arithmetic assignments result in new states.

• We disregard the intermediate states and return the final state.

• We inherit the dangers of ordinary program design, including the possibility of infinite (unbounded) loops.


\textbf{P}^2: Planning with Programs

\begin{verbatim}
proc ProgPlan(S, G, A, P)
  if EvalExpr(G, S) = true then return P
  else if choose(a ∈ A) : EvalExpr(pre(a), S) = true then
    S' = RunProg(\text{eff}(a), S);
    return ProgPlan(S', G, A, P + a)
  else return fail
endIf
endProc
\end{verbatim}

- \textit{PlanProg} attempts to build a plan starting from state \(S\) to achieve goal \(G\) (an expression) using action set \(A\), starting from initial plan \(P\). On success, the resulting sequential plan is returned.
Implementation

• Current version is implemented in C++ as a simple forward-chaining planner using blind depth-first search and breadth-first search.

• A compiler translates the planner input into an intermediate representation. E.g.,

```plaintext
0 | pushmi 0
1 | pushi 5
2 | gotolt 5
3 | pushi 0
4 | goto 6
5 | pushi 1
6 | gotof 14
7 | pushi 0
8 | pushmi 0
9 | pushi 1
10 | add
11 | assign
12 | pop
13 | goto 0
14 | pushi 1
15 | halt
```

while(a < 5)  
a = a + 1  =>
endWhile

• The program simulator executes the intermediate representation on a simple virtual (abstract stack) machine (Aho, Sethi, & Ullman 1985).
Experiment 1

• Single action with a **while** loop running on P² compared against a compiled PDDL domain (3 actions) running on Metric-FF (Hoffmann 2003).

```
action processDataset(?d)
  precondition:
    dataset(?d) and
    not(processedDataset(?d))
  effect:
    i = 1 ;
    while (i <= size(?d))
      count = count + i ;
      i = i + 1
    endWhile ;
    processedDataset(?d)
endAction
```

• Use a single dataset d1 and vary the size of size(d1).

• All tests were performed on a Linux system with a single CPU running at 1.86 GHz and 2 Gb of RAM.
Evaluation 1

The graph shows the relationship between the size of a problem (d1) and the time (s) it takes to solve it for two metrics: Metric-FF and P2. The time increases significantly with the size of the problem, indicating a non-linear increase in computational complexity.
Experiment 2

- Test the speed of the virtual machine.
- A single action with no preconditions. Action effects consist of a while loop of \( n \) iterations. The contents of the while loop vary:
  - Test-1 A single fluent update.
  - Test-2 An if-else statement which conditionally performs a fluent update.
  - Test-3 A forall statement ranging over a domain of 50 objects, performing a fluent update for each instantiation.
  - Test-4 A forall statement ranging over a domain of 100 objects, performing a fluent update for each instantiation.

- Goal is to generate a plan that chains 100 actions together. E.g., when \( n = 10000 \) in Test-1 the program simulator is running 1 million loop iterations and fluent updates.
Evaluation 2

![Graph showing the relationship between time (s) and n for different tests. The graph includes lines for Test-1, Test-2, Test-3, and Test-4, with Test-4 having the steepest slope.]
Discussion

• In Experiment 1, Metric-FF’s performance is not altogether bad, e.g., a 2500 step plan is built in 2 seconds. ($P^2$ always builds a plan of length 1 almost instantly in this experiment.)

• ...on the other hand, the example domain is quite simple and more complex domains should result in a more significant performance hit for Metric-FF.

• In Experiment 2, program simulation using a virtual machine is fast.

• ...however, blind search does not scale well.

• Current work: combine FF style search heuristics together with program simulation.
  – Initially focusing on subsets of our representation for which this approach can be easily applied.
  – Complications: estimates of loop iterations/bounds (e.g., the exit condition might depend on the deletion of a fluent).
Conclusions and future work

• Our approach differs from the complex-to-primitive action compilation methods since we plan directly with program structures.

• There are similarities to Golog (Levesque et al. 1997). As future work our informal procedural semantics could be redefined in terms of Golog programs. We are also identifying sets of Golog programs that can be run directly on P².

• Our planner provides a baseline for comparing against future advances and is not meant to be competitive with current off-the-shelf planners. More work is needed to improve the search strategy.

• P² is (surprisingly) fast on program-intense domains.

• Future work: incomplete information and sensing (cf. PKS), a “procedure call”, an optimised virtual machine, a real compiler?

• Source code for P² will be available at:

  http://homepages.inf.ed.ac.uk/rpetrick/research/p2/


References...(2)


