From: AAAI Technical Report WS-98-02. Compilation copyright © 1998, AAAI (www.aaai.org). All rights reserved.

Mapping Planning Actions and Partially-Ordered Plans into Execution Knowledge

Manuela M. Veloso
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
mmv@cs.cmu.edu

Paola Rizzo
Center of Cognitive Science, University of Turin, and
Institute of Psychology
National Research Council
Viale Marx 15, 00137 Roma, Italy
paola@di.unito.it

Abstract

In essence, the underlying ultimate purpose of classical deliberative planning systems is to model some world task and generate plans that can then be executed. Execution systems rely on the assumption that there is a built-in library of a variety of plans and primitive actions for performing tasks. Although hard and time consuming, such libraries of execution plans and actions are usually handcoded. Faced with two different systems, a planner and an executor, we analyze in this work the representational map between planning actions and partially-ordered plans and the execution knowledge. We developed algorithms to automatically translate classical planning operators to execution primitive actions, and partially-ordered plans into executable tasks with partially ordered subtasks. We implemented our work using the PRODIGY planner and the RAP execution system. We provide illustrative examples of how partially-ordered plans produced by PRODIGY are translated to Reactive Action Packages suitable for execution by the RAP system.

Introduction

In essence, the underlying ultimate purpose of classical deliberative planning systems is to model some world task and generate plans that can then be executed. Execution systems rely on the assumption that there is a built-in library of a variety of plans and primitive actions for performing tasks. Although hard and time consuming, such libraries of execution plans and actions are usually handcoded.

Here we propose an approach to the automatic building of a library of executable actions by using the knowledge contained in planning operators and plans produced by planners; in particular, we propose an algorithm for translating partial order plans produced by a generative planner, PRODIGY, into lists of subtasks that can be processed by an executor, RAP.

The paper is organized as follows. First an overview of the PRODIGY deliberative planner and of the RAP reactive planner is given. The third section describes how PRODIGY actions are translated into Reactive Action Packages (RAPs), and then how partially-ordered plans are translated into RAPs with some illustrative examples. The final section describes future work and related literature.

The PRODIGY deliberative planner

The PRODIGY planner (Veloso *et al.* 1995) reasons about multiple goals and multiple alternative operators relevant to achieving the goals; it combines state-space search corresponding to a simulation of plan execution with backward-chaining responsible for goal-directed reasoning. The strategies used in PRODIGY for directing its choices in decision points are called *control knowledge*. These strategies include the use of control rules (usually domain-dependent), complete problem solving episodes to be used by analogy, and domain-independent heuristics. Here we focus on explaining control rules.

A *control rule* is a production (*if-then*) rule that tells the system which choices should be made (or avoided) depending on the current state, unachieved preconditions of the operators, and other meta-level information based on previous choices or subgoaling links. All the control rules used by PRODIGY are distinguished into three groups: *select*, *reject*, and *prefer* rules. If several control rules are applicable in the current decision point, PRODIGY will use all of them. *Select* and *reject* rules are used to prune parts of the search space, while *prefer* rules determine in what order to explore the remaining parts.

The RAP reactive planner

RAP (Firby 1989), like other reactive planners, e.g. (Georgeff & Lansky 1986; McDermott 1990) is a system devoted to the execution of plans in a dynamic environment. Usually systems of this family are not able to generate plans by themselves but rather rely on a complex plan specification language to handwrite

plans. These languages have several constructs to differentiate the execution according to environmental conditions, allowing the agent to profit from opportunities, to suspend pursuing a goal and give priority to another that has become more urgent, and to change plan in face of execution failures or modified environmental conditions.

RAP is composed of a library of Reactive Action Packages (or RAPs), [1] a task agenda, and a memory. A Reactive Action Package is the basic execution unit. Each task (i.e., achieving a goal) is mapped on a particular RAP, which specifies the set of methods (i.e., unistantiated plans) for achieving it and their conditions of applicability. The methods may consist of primitive actions (i.e. they can be directly executed) or in subtasks to be executed by using other RAPs.

The initial tasks to be performed are put on the task agenda. The interpreter selects one task at a time from the agenda by considering task selection constraints and heuristics. The RAP associated with the selected task is taken from the RAPs library, and an applicable method is chosen for execution. The choice of a method from those listed in the RAP is based on the contents of the memory, which is RAP's internal model of the world. If the method is a primitive action, it is executed; otherwise all of its subtasks are put on the agenda, and the main task is put on the agenda to be checked for success after all of its subtasks are finished.

Transforming planning operators and plans into executable tasks

Our architecture is composed of a deliberative planner that off-line builds plans that can be executed by the reactive planner. Since PRODIGY's operators and plans and Reactive Action Packages do not have the same syntax, it is necessary to translate the former into the latter.

Here we will see the automatic process through which (1) PRODIGY operators are translated into RAPs consisting of directly executable primitive actions, and the effects of PRODIGY operators are translated into "memory rules" for changing the contents of the RAP memory after the successful execution of a primitive RAP; [2] (2) PRODIGY's (possibly multiple) plans are translated into complex RAPs composed of partially ordered primitive RAPs previously translated from the planning operators.

[1] RAP refers to the whole system, while RAP(s) refer to Reactive Action Package(s).

[2] Currently, in such translations we deal with operators containing neither conditional effects nor quantifications; later work will be devoted to such issues.

From PRODIGY operators to RAPs and memory rules. The translation from operators into RAPs is straightforward: an operator is defined through its name, parameters, preconditions and effects; the name and parameters are translated into the RAP index, while the preconditions are mapped into the RAP preconditions. [3] Since the RAP's effects cannot be represented inside the RAP itself, the operator's effects are mapped into a so-called memory rule in the RAP system; such rule represents how the RAP affects the world after its successful execution. Here follows an example:

```
;;; Prodigy operator

(operator EAT
   (params <food>)
   (preconds (has myself <food>))
   (effects ()
       ((del (hungry myself))
        (del (has myself <food>)))))


;;; RAP translated from the operator

(define-RAP (index (eat ?food))
 (preconditions (has myself ?food))
 (method
   (primitive
     (enable (eat ?food))
     (wait-for (succeeded eat ?result)
                    :succeed ?result)
     (wait-for (failed eat ?result)
                    :fail ?result)
     (disable :above))))


;;; RAP memory rule

(define-memory-rule (eat ?food) :finish
  (match-result (okay
    (rule (t
      (mem-del (hungry myself))
      (mem-del (has myself ?food)))))))
```

From PRODIGY plans to RAPs. We describe the translation from PRODIGY plans to RAPs by referring to a simple problem, solved by PRODIGY by producing 2 alternative solutions. The problem (in which the object declarations are omitted) and solutions are shown below.

```
;;; problem

(setf
  (current-problem)
    (create-problem
```

[3] The object declarations are omitted.

```
    (name have-fun)
    (state
      (and (has other-agt toy)
           (has myself book)
           (willing-to-play other-agt)))
    (goal (have-fun myself))))


;;; solutions

Solution #1:
<play-with-agt other-agt>

Solution #2:
<bargain-obj-with-agt book toy other-agt>
<play-with-obj toy>
```

The Reactive Action Package shown below is composed by both plans: each of them maps into a different method. Notice that the ordering constraints of the solutions (taken from a contingency table not shown here) are represented also in the methods through the use of *for* clauses, that specify which task depends on the successful execution of another. The context for each method refers to a subset of the initial state of the problem that must be true for using that particular method. Finally, notice that RAPs represent a generalization of PRODIGY plans, by transforming literals into predicates with variables.

```
;;; RAP translated from the solutions

(define-RAP (index (have-fun))
  (succeed (entertained myself))
  (method 0
   (context (willing-to-play ?agt))
   (task-net
     (1 play-with-agt ?agt)))
  (method 1
   (context (and (has ?agt ?toy)
                 (has myself ?obj)))
   (task-net
     (1 (bargain-obj-with-agt ?obj ?toy ?agt)
        (for 2))
     (2 (play-with-obj ?toy)))))
```

Final Remarks

If our way of using planning techniques is not strictly conventional, other works exist that are somehow similar to ours. RAP has been coupled with PRODIGY also in (Blythe & Reilly 1993), and with another generative planner (AP) in (Bonasso *et al.* 1995). Both works differ from ours mainly because of their rather "conventional" view of the integration of generative and reactive planning. In fact, the following differences emerge: (a) RAPs are not automatically created by translating the planner's operators and plans, but are written and mapped to the planning operators by hand; (b) the generative planner interacts with RAP online rather than off-line, by passing its plans to RAP for immediate execution, and by monitoring the results. Another work (Wilkins & Myers 1995) has addressed the problem of making a generative planner (SIPE-2) and a reactive planner (PRS-CL) speak a common language. Their solution differs from ours because they have created an interlingua (the ACT formalism) used for handwriting plans that can subsequently be understood and used by both systems; on the contrary, in our work there is a unidirectional translation from PRODIGY to RAP, and only the planning operators must be written by the designer, while plans are produced by PRODIGY and automatically transformed into RAPs.

We are currently using this architecture for building behaviors for believable agents that interact with a user. The architecture has produced a set of plans for each agent specification and for several agents (Rizzo *et al.* in press).

Further developments of our work concern the use of our architecture for realizing agents that interact with each other; to this aim, we are currently working on a multi-agent version of RAP. In addition, we aim at realizing a tighter integration of the deliberative and the reactive planner, in which the former would inspect the task agenda, and build new plans on the fly during interactions among agents; this would allow to take into account the interactions between tasks occurring at execution time.

Acknowledgements

Part of this work was done while the second author was visiting the Computer Science Department at Carnegie Mellon University. Manuela Veloso's research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-97-2-0250. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory (AFRL) or the U.S. Government. Paola Rizzo is currently supported by a scholarship from CNR Committee 12 on Information Technology.

References

Blythe, J., and Reilly, W. 1993. Integrating Reactive and Deliberative Planning for Agents. Technical Report CMU-CS-93-155, Carnegie Mellon University, School of Computer Science.

Bonasso, R. P.; Kortenkamp, D.; Miller, D. P.; and Slack, M. 1995. Experiences with an Architecture for Intelligent, Reactive Agents. In *Proceedings of ATAL-95.*

Firby, R. J. 1989. *Adaptive Execution in Complex Dynamic Domains.* Ph.D. Dissertation, Yale University. Technical Report YALEU/CSD/RR #672.

Georgeff, M., and Lansky, A. L. 1986. Procedural Knowledge. *Proceedings of IEEE* 74(10):1383–1398.

McDermott, D. 1990. Planning Reactive Behavior: A Progress Report. In Sycara, K., ed., *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control.* San Mateo (CA): Morgan Kaufmann.

Rizzo, P.; Veloso, M. M.; Miceli, M.; and Cesta, A. in press. Goal-based personalities and social behaviors in believable agents. *Applied Artificial Intelligence.*

Veloso, M. M.; Carbonell, J.; Perez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7:81–120.

Wilkins, D. E., and Myers, K. L. 1995. A Common Knowledge Representation for Plan Generation and Reactive Execution. *Journal of Logic and Computation.*