A Replanning Algorithm for a Reactive Agent Architecture

Guido Boella and Rossana Damiano

Dipartimento di Informatica
Cso Svizzera 185 Torino ITALY
email: guido@di.unito.it

Keywords: Planning, Agent architectures

Abstract. We present an algorithm for replanning in a reactive agent architecture which incorporates decision-theoretic notions to drive the planning and meta-deliberation process. The deliberation component relies on a refinement planner which produces plans with optimal expected utility. The replanning algorithm we propose exploits the planner's ability to provide an approximate evaluation of partial plans: it starts from a fully refined plan and makes it more partial until it finds a more partial plan which subsumes more promising refinements; at that point, the planning process is restarted from the current partial plan.

1 Introduction

In this paper we present a replanning algorithm developed for a reactive agent architecture which incorporates decision-theoretic notions to determine the agent's commitment. The agent architecture is based on the planning paradigm proposed by [Haddawy and Hanks, 1998], which combines decision-theoretic refinement planning with a sound notion of action abstraction ([Ha and Haddawy,]): given a goal and a state of the world, the planner is invoked on a partial plan (i.e. a plan in which some actions are abstract) and iteratively refines it by returning one or more plans which maximize the expected utility according to the agent's preferences, modelled by a multi-attribute utility function.

The decision-theoretic planning paradigm extends the classical goal satisfaction paradigm by allowing partial goal satisfaction and the trade-off of goal satisfaction against resource consumption. Moreover, it accounts for uncertainty and non determinism, which provide the conceptual instruments for dealing with uncertain world knowledge and actions having non-deterministic effects. These features make decision-theoretic planning especially suitable for modelling agents who are situated in dynamically changing, non deterministic environments, and have incomplete knowledge about the environment.

However, decision-theoretic planning frameworks based on plan refinement ([Haddawy and Hanks, 1998]) do not lend themselves to reactive agent architectures, as they do not include any support for reactive replanning. In this paper, we try to overcome this gap, by proposing an algorithm for replanning for a reactive agent architecture based on decision-theoretic notions.

Since optimal plans are computed with reference to a certain world state, if the world state changes, the selected plan may not be appropriate anymore. Instead of planning an alternative solution from the scratch, by re-starting the planning process from the goal, the agent tries to perform replanning on its current plan.

The replanning algorithm is based on a *partialization process*: it proceeds by making the current solution more partial and then starting the refinement process again. This process is repeated until a new feasible plan is found or the partialization process reaches the topmost action in the plan library (in this case, it coincides with the standard planning process).

We take advantage of the decision-theoretic approach on which the planner is based not only for improving the quality of the replanned solution, but also for guiding the replanning process. In particular, the planner ability to evaluate the expected utility of partial plans provides a way to decide whether to continue the partialization process or to re-start refinement: for each partial plan produced in the partialization step, it is possible to make an approximate estimate of whether and with what utility the primitive plans it subsumes achieve the agent's goal.

Then, the pruning heuristic used during the standard planning process to discard sub-optimal plans can be used in the same way during the replanning process to reduce its complexity.

2 The agent architecture

The architecture is composed of a *deliberation module*, an *execution module*, and a *sensing module*, and relies on a *meta-deliberation* module to evaluate the need for re-deliberation, following [Wooldridge and Parsons, 1999]. The agent is a BDI agent ([Rao and Georgeff, 1991]), in that its internal state of the agent is defined by its beliefs about the current world, its goals, and the intentions (plans) it has formed in order to achieve a subset of these goals . The agent's deliberation and redeliberation are based on decision-theoretic notions: the agent is driven by the overall goal of maximizing its utility based on a set of preferences which are encoded in a utility function.

The agent is situated in a dynamic environment, i.e. the world can change independently from the agent's actions, and actions can have non-deterministic effects, i.e., an action can result in a set of alternative effects. Moreover, there is no perfect correspondence between the environment actual state and the agent's representation of it.

In this architecture, intentions are not static, and can be modified as a result of re-deliberation: if the agent detects a significant mismatch between the initially expected and the currently expected utility brought about by a plan, the agent revises its intentions by performing re-deliberation. As a result, the agent is likely to become committed to different plans along time, each constituted of a different sequence of actions. However, while the intention to execute a certain plan remains the same until it is dropped or satisfied, the commitment to execute single actions evolves continuously as a consequence of both execution and re-deliberation.

In order to represent dynamic intentions, separate structures for representing plan-level commitment and action-level commitment have been introduced in the architecture. So, intentions are stored in two kind of structures: *plans*, representing goal-level commitment, and *action-executions*, representing action-level commitment. New instances of the *plan* structure follow one another in time as a consequence of the agent's re-deliberation; on the contrary, the action-level commitment of an agent is recorded in a unitary instance of the *action-execution* structure, called *execution record*, whose temporal extent coincides with the agent's commitment to a goal and which is updated at every cycle.

The behavior of the agent is controlled by an execution-sensing loop with a meta-level deliberation step (see figure 1). When this loop is first entered, the deliberation module is invoked on the initial goal; the goal is matched against the plan schemata contained in the library, and when a plan schema is found, it is passed to the planner for refinement. This plan becomes the agent's current intention, and the agent starts executing it. After executing each action in the plan, the sensing module monitors the effects of the action execution, and updates the agent's representation of the world. Then, the meta-deliberation module evaluates the updated representation by means of an execution-monitoring function: if the world meets the agent's expectations, there is no need for re-deliberation, and the execution is resumed; otherwise, if the agent's intentions are not adequate anymore to the new environment, then the deliberation module is assigned the task of modifying them.

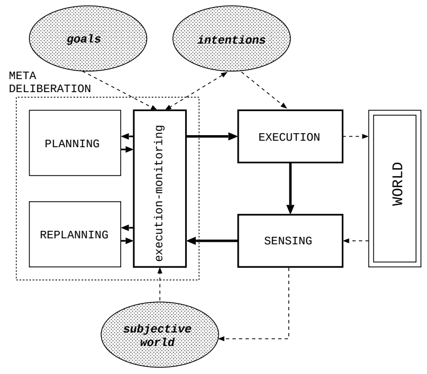

Fig. 1. The structure of the agent architecture. Dashed lines represent data flow, solid lines represent control flow.

Due to the agent's uncertainty about the outcome of the plan, the initial plan is associated to an expected utility interval, but this interval may vary as the execution of the plan proceeds. More specifically, after the execution of a non-deterministic action (or a conditional action, if the agent did not know at deliberation time what conditional effect would apply), the new expected utility interval is either the same as the one that preceded the execution, or a different one. If it is different, the new upper bound of the expected utility can be the same as the previous one, or it can be higher or lower - that is, an effect which is more or less advantageous than expected has taken place.

The execution-monitoring function, which constitutes the core of the meta-deliberation module, relies on the agent's subjective expectations about the utility of a certain plan: this function computes the expected utility of the course of action constituted by the remaining plan steps in the updated representation of the world. The new expected utility is compared to the previously expected one, and the difference is calculated: replanning is performed only if there is a significant difference.

If new deliberation is not necessary, the meta-deliberation module simply updates the execution record and releases the control to the execution module, which executes the next action. On the contrary, if new deliberation is necessary, the deliberation module is given the control and invokes its *replanning component* on the current plan with the task of finding a better plan; the functioning of the replanning component is inspired to the notion of persistence of intentions ([Bratman et al., 1988]), in that it tries to perform the most local replanning which allows the expected utility to be brought back to an acceptable difference with the previously expected one.

3 The planning algorithm

The action library is organised along two *abstraction* hierarchies. The *sequential abstraction* hierarchy is a task decomposition hierarchy: an action type in this hierarchy is a macro-operator which the planner can substitute with a sequence of (primitive or non-primitive) action types. The *specification hierarchy* is composed of abstract action types which subsume more specific ones.
The specification hierarchy is obtained by *inheritance abstraction*, a technique for grouping together conditional probabilistic action operators in abstract classes based on their outcomes which is characterized by the common features of all elements of the class ([Haddawy and Suwandi, 1994]), while the decomposition hierarchy is obtained by sequential abstraction, i.e., by gathering stereotypical sequences of action types into complex action types ([Sacerdoti, 1977]). In the following, for simplicity, we will refer to *sequentially abstract* actions as *complex* actions and to actions in the specification hierarchy as *abstract* actions.

A plan (see section 2) is a sequence of action instances and has associated the goal the plan has been planned to achieve. A plan can be partial both in the sense that some steps are complex actions and in the sense that some are abstract actions. Each plan is associated with the derivation tree (including both abstract and complex actions) which has been built during the planning process and that will be used for driving the replanning phase.

Before refining a partial plan, the agent does not know which plan (or plans) - among those subsumed by that partial plan - is the most advantageous according to its preferences. Hence, the expected utility of the abstract action is *uncertain*: it is expressed as an interval having as upper and lower bounds the expected utility of the best and the worst outcomes produced by substituting in the plan the abstract action with all the more specific actions it subsumes. This property is a key one for the planning process as it makes it possible to compare partial plans which contain abstract actions.

The planning process starts from the topmost action in the hierarchy which achieves the given goal. If there is no time bound, it proceeds refining the current plan(s) by substituting complex actions with the associated decomposition and abstract actions with all the more specific actions they subsume, until it obtains a set of plans which are composed of primitive actions.

At each cycle the planning algorithm re-starts from a less partial plan: at the beginning this plan coincides with the topmost action which achieves the goal, in the subsequent refinement phases it is constituted by a sequence of actions; this feature is relevant for replanning, as it make it possible to use the planner for refining any partial plan, no matter how it has been generated.

At each refinement step, the expected utility of each plan is computed by projecting it from the current world state. Then, a *pruning* heuristic is applied by discarding the plans identified as suboptimal, i.e., plans whose expected utility upper bound is lower

```
procedure plan replan(plan p, world w){
/* find the first action which will fail */
 action a := find-focused-action(p,w);
 mark a; //set a as the FA
 plan p’ := p;
 plan p’’ := p;
/* while a solution or the root are not found */
 while (not(achieve(p’’,w, goal(p’’)))
            and has-father(a)){
/* look for a partial plan with better utility */
  while (not (promising(p’, w, p))
         and has-father(a)){
    p’ := partialize(p’);
    project(p’,w); } //evaluate the action in w
/* restart planning on the partial plan */
 p’’ := refine(p’,w);}
 return p’’;}
```

Fig. 2. The main procedure of the replanning algorithm, *replan*

than the lower bound of some other plan p. The suboptimality of a plan p' with respect to p means that all possible refinements of p have an expected utility which dominates the utility of p', and, as a consequence, dominates the utility of all refinements of p': consequently, suboptimal plans can be discarded without further refining them. On the contrary, plans which have overlapping utilities need further refinement before the agent makes any choice. At each step of refinement the *expected utility interval* of a plan tends to become narrower, since it subsumes a reduced number of plans (in fact, the plan appears deeper in the hierarchy of plans).

4 The replanning algorithm

If a replanning phase is entered, then it means that the current plan does not reach the agent’s goal, or that it reaches it with a very low utility compared with the initial expectations. But it is possible that the current plan is ‘close’ to a similar feasible solution, where closeness is represented by the fact that both the current solution and a new feasible one are subsumed by a common partial plan at some level of the action abstraction hierarchy.

The key idea of the replanning algorithm is then to make the current plan more partial by traversing the abstraction hierarchies in a upsidedown manner, until a more promising abstract plan is found. The abstraction and the decomposition hierarchy play complementary roles in the algorithm: the abstraction hierarchy determines the alternatives for substituting the actions in the plan, while the decomposition hierarchy is exploited to focus the substitution process on a portion of the plan.
A partial plan can be identified as promising by observing its expected utility interval, since this interval includes not only the utility of the (unfeasible) current plan but also the utility of the new solution. So, during the replanning process, it is possible to use this estimate in order to compare the new plan with the expected utility of the more specific plan from which it has been obtained: if it is not promising it is discarded.

```
function plan partialize(plan p){
action a := marked-action(p); /* a is the FA of p */
/* if it is subsumed by a partial action */
if (abstract(father(a))){
   delete(a, p); /* delete a from the tree */
   return p;}
/* no more abstract parents: we are in a decomposition */
 else if (complex(father(a)){
        a1 := find-sibling(a,p);
        if (null(a1)){
/* there is no FA in the decomposition */
            mark(father(a)) //set the FA
      //delete the decomposition
            delete(descendant(father(a)),p);
            return p;}
   else { //change the current FA
            unmark(a);
            mark(a1);}}}
```

Fig. 3. The procedure for making a plan more abstract, *partialize*.

The starting point of the partialization process inside the plan is the first plan step whose *preconditions* do not hold, due to some event which changed the world or to some failure of the preceding actions. In [Haddawy and Suwandi, 1994]'s planning framework the Strips-like precondition/effect relation is not accounted for: instead, an action is described as a set of conditional effects. The representation of an action includes both the action intended effects, which are obtained when its ‘preconditions’ hold, and the effects obtained when its ‘preconditions’ do not hold. For this reason, the notation of the action has been augmented with the information about the action intended effect, which makes it possible to identify its preconditions.[1]

The task of identifying the next action whose preconditions do not hold (the ‘focused action’) is accomplished by the *Find-focused-action* function (see the main procedure in Figure 2); *mark* is the function which sets the current focused action of the plan). Then, starting from the focused action (FA), the replanning algorithm partializes the plan, following the derivation tree associated with the plan (see the *partializes* function in Figure 3).

If the action type of the FA is directly subsumed by an abstract action type in the derivation tree, the focused action is deleted and the abstract action substitutes it in the tree frontier which constitutes the plan. On the contrary, if FA appears in a decomposition (i.e., its father in the derivation tree is a sequentially abstract action) then two cases are possible (see the find-sibling function in 4):

1. There is some action in the plan which is a descendant of a sibling of FA in the decomposition and which has not been examined yet: this descendant of the sibling

[1] Since it is possible that more than one condition-effect branch lead to the goal (maybe with different satisfaction degrees), different sets of preconditions can be identified by selecting the condition associated to successful effects.

```
function action find-sibling(a,p){
/* get the next action  to be refined (in the same decomposition as a) */
 action a0 := right-sibling(a,p);
 action a1 := leftmost(descendant(a0,p));
 while(not (null (a1))){
/* if it can be partialized */
   if (not complex(father(a1))){
     unmark(a); //change FA
     mark(a1)
     return a1;}
/* move to next action */
   a0 := right-sibling(a0,p);
   a1 := leftmost(descendant(a0,p));}
/* do the same on the left side of the plan */
 action a1 := left-sibling(a,p);
 action a1 := rightmost(descendant(a0,p));
 while(not (null (a1))){
   if (not complex(father(a1))){
     unmark(a);
     mark(a1)
     return a1;}
 action a1 := left-sibling(a,p);}
```

Fig. 4. The procedure for finding the new focused action.

becomes the current FA. The order according to which siblings are considered reflects the assumption that it is better to replan non-executed actions, when possible: so, right siblings (from the focused action on) are given priority on left siblings.

2. All siblings in the decomposition have been already refined (i.e., no one has any descendant): all the siblings of FA and FA itself are removed from the derivation tree and replaced in the plan by the complex sequential action, which becomes the current FA (see Figure 4).[2]

As discussed in the Introduction, the pruning process of the planner is applied in the refinement process executed during the replanning phase. In this way, the difficulty of finding a new solution from the current partial plan is alleviated by the fact that suboptimal alternatives are discarded before their refinement.

Beside allowing the pruning heuristic, however, the abstraction mechanism has another advantage. Remember that, by the definition of abstraction discussed in Section 2, it appears that, given a world state, the outcome of an abstract action includes the outcomes of all the actions it subsumes.
Each time a plan p is partialized, the resulting plan p' has an expected utility interval that includes the utility interval of p. However p' subsumes also other plans whose outcomes are possibly different from the outcome of p. At this point, two cases are possible: either the other plans are better than p or not. In the first case, the utility of p' will

[2] Since an action type may occur in multiple decompositions[3], in order to understand which decomposition the action instance appears into, it is not sufficient to use the action type library, but it is necessary to use the derivation tree).

have an higher higher bound with respect to p, since it includes all the outcomes of the subsumed plans. In the second case, the utility of p' will not have a higher upper bound than p. Hence, p' is not more promising than the less partial plan p.
The algorithm exploits this property (see the *promising* condition in the procedure *replan*) to decide when the iteration of the partialization step must be stopped: when a promising partial plan (i.e., a plan which subsumes better alternatives than the previous one) is reached, the partialization process ends and the refinement process is restarted on the current partial plan.

The abstraction hierarchy has also a further role. The assumption underlying our strategy is that a plan failure can often be resolved *locally*, within the subtree the focused action appears into. Not all failures, however, can be resolved locally, but these cases are taken into account by the algorithm as well: after the current subtree has been completely partialized, a wider subtree in the derivation tree will be considered, until the topmost root action is reached: in this case, the root of the derivation tree becomes the FA and the planning process is restarted from scratch.
In case of non-local causal dependencies among actions (i.e., a precondition of the FA is enabled by the effect of an action which does not appear in the local context of FA), the algorithm takes advantage from the fact that the current partial plan is *projected* onto its final state and its expected utility is computed: provided that the definition of abstract action operators is sufficiently accurate to make casual dependencies explicit, it is likely that invalid dependencies will be reflected in the expected utility of the current partial plan, and, as a consequence, it will be pruned during refinement without being further expanded.

Finally, the movement of the FA is a critical point of the algorithm. Here we presented *find-sibling* as a simple process which follows the local structure of the tree. However, some improvements are possible to take advantage from the cases in which non local dependencies are known. Hence, the *find-sibling* procedure should be modified in order to use in deeper way the structure of the plans and, in particular, the implicit enablement links among actions for choosing the next FA.

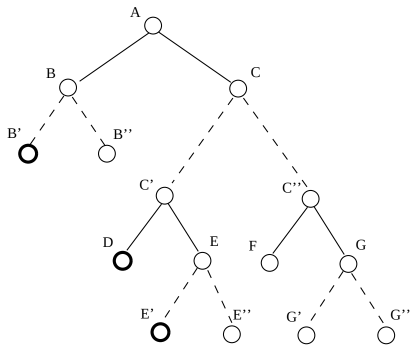

Fig. 5. A generic action hierarchy. Abstraction relations are represented by dashed lines.

For the sake of brevity, in order to illustrate how the replanning algorithm works, we will resort to a generic action hierarchy (see fig. 5), which abstracts out the details of the domains we used to test the implementation.
In the following, we will examine the replanning process that the algorithm would perform, given the initial plan composed of the steps B' - D - E' (see fig. 6).

1. Assume that, at the beginning of the replanning process, the focused step is action D (1). D is examined first, but an alternative instantiation of it cannot be found (as

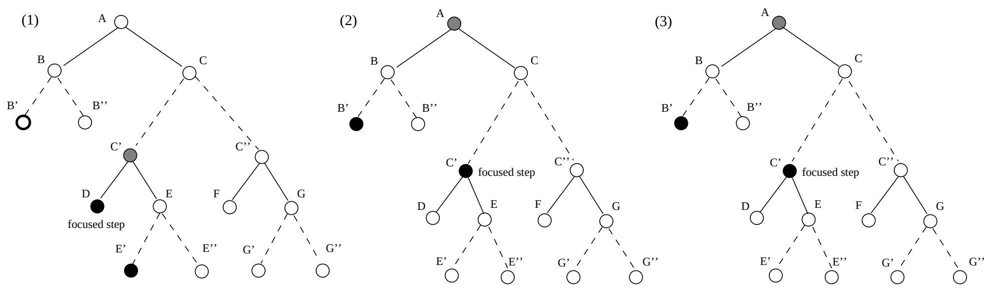

Fig. 6. A graphical representation of the plan replanning process on the generic action library introduced in 5. Black nodes represent the siblings of the focused action node, while the grey nodes represent the local decomposition context. (1)-(2)-(3) represent the phases of the replanning process.

its immediate parent is not an abstract action). The find-siblings function returns the right sibling of D, E.

2. The planner is given as input the partial plan B' - D - E. Assuming that a feasible plan is not found (i.e., B' - D - E'', the only possible alternative to the original plan, does not work), the replanning process is started again after collapsing the sub-plan ($D - E$) on its father, the complex node C' (no siblings left).
3. Given the new input plan B' - C', the focused step now is C' (2). The focused step is examined first, and the more abstract father node C is found; C' is replaced by C in the plan and the planner is invoked on the new partial plan B'-C.
4. Again, assuming that a new feasible plan has not been found by refining B'-C, the replanning process continues by examining B, the only sibling of the focused action C (3). Before the candidate plan is collapsed on its root (A), the replanning process gives the planner as input the plan obtained by substituting the more abstract node B for B' in the current partial plan, obtaining B - C.
5. Finally, if the refinement of the partial plan B - C does not yield a feasible plan, the plan is collapsed on the its father A. If a feasible plan is not found by refining the plan constituted by the root alone, the plan replanning algorithm fails.

In the previous version of the algorithm, the *find-sibling* step proceeds not only from left to right (towards actions yet to be executed), but also in a backward manner: at a certain point it is possible that the focused point is shifted to an already executed actions. In order to overcome this problem, we propose that the projection rule should is changed to include in the projection the actions that must be executed again (possibly in an alternative way). In this case, the FA would be moved incrementally to the left, and would become the reference point for starting the projection of the current partial plan.

5 Related Work and Conclusions

[Hanks and Weld, 1995] has proposed a similar algorithm for an SNLP planner. The algorithm searches for a plan similar to known ones first by retracting refinements: i.e., actions, constraints and causal links. In order to remove the refinements in the right

order, [Hanks and Weld, 1995] add to the plan an history of ‘reasons’ explaining why each new element has been inserted.
In a similar way, our algorithm adapts the failed plan to the new situation by retracting refinements, even if in the sense of more specific actions and decompositions. The same role played by ‘reasons’ is embodied in the derivation tree associated to the plan which explains the structure of the current plan and guides the partialization process.

As it has been remarked on by ([Nebel and Koehler, 1993]), reusing existing plans raises complexity issues. They show that modifying existing plans is advantageous only under some conditions: in particular, when, as in our proposal, it is employed in a replanning context (instead of a general plan-reuse approach to planning) in which it is crucial to retain as many steps as possible of the plan the agent is committed to. Second, when the complexity of generating plans from the scratch is hard, as in the case of our decision-theoretic planner.

For what concerns the complexity issues, it must be noticed that the replanning algorithm works in a similar way as the *iterative deepening* algorithm. At each stage, the height of the tree of the state space examined increases. The difference with the standard search algorithm is that, instead of starting the search from the tree root and stopping at a certain depth, we start from a leaf of the plan space and, at each step, we select a higher tree which rooted by one of the ancestors of the leaf.
In the worst case, the order of complexity of the replanning algorithm is the same as the standard planning algorithm. However, two facts that reduce the actual work performed by the replanning algorithm must be taken into account: first, if the assumption that a feasible solution is “close” to the current plan is true, then the height of the tree which includes both plans is lower than the height of root of the whole state space. Second, the pruning heuristics is used to prevent the refinement of some of the intermediate plans in the search space, reducing the number of refinement runs performed.

Finally, it is worth mentioning that the replanning algorithm we propose is complete, in that it finds the solution if one exists, but it does not necessarily finds the optimal solution: the desirability of an optimal solution, in fact, is subordinated to the notions of resource-boundedness and to the persistence of intentions, which tend to privilege conservative options.

References

[Bratman et al., 1988] Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355.

[Ha and Haddawy,] Ha, V. and Haddawy, P. Theoretical foundations for abstraction-based probabilistic planning. In *Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 291–298, Portland.

[Haddawy and Hanks, 1998] Haddawy, P. and Hanks, S. (1998). Utility models for goal-directed, decision-theoretic planners. *Computational Intelligence*, 14:392–429.

[Haddawy and Suwandi, 1994] Haddawy, P. and Suwandi, M. (1994). Decision-theoretic refinement planning using inheritance abstraction. In *Proc. of 2nd AIPS Int. Conf.*, pages 266–271, Menlo Park, CA.

[Hanks and Weld, 1995] Hanks, S. and Weld, D. S. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2:319–360.

[Nebel and Koehler, 1993] Nebel, B. and Koehler, J. (1993). Plan modification versus plan generation: A complexity-theoretic perspective. In *Proceedings of of the 13th International Joint Conference on Artificial Intelligence*, pages 1436–1441, Chambery, France.

[Rao and Georgeff, 1991] Rao, A. and Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In *Proc. 2th Int. Conf. Principles of Knowledge Representation and Reasoning (KR:91)*, pages 473–484, Cambridge, MA.

[Sacerdoti, 1977] Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. American Elsevier, New York.

[Wooldridge and Parsons, 1999] Wooldridge, M. and Parsons, S. (1999). Intention reconsideration reconsidered. In Müller, J., Singh, M. P., and Rao, A. S., editors, *Proc. of ATAL-98)*, volume 1555, pages 63–80. Springer-Verlag.