Embedded Planning

Philippe Morignot

Knowledge Systems Laboratory
Stanford University, 701 Welch Road
Palo Alto, California 94304
morignot@ksl.stanford.edu

Abstract

We [1] present a planner that generates nonlinear plans in real-time to control an autonomous agent. Requirements of plan generation in an embedded system are obtained by extending the action representation of [Chapman 87] and [McAllester & Rosenblitt 91] regarding locations, arithmetic equations and variable's domain. The implementation of this planner exhibits attractive real-time performances. This planner is used to generate plans on the fly for a mobile robot performing various tasks to assist in office work.

1. Introduction

Imagine an agent that has to assist a secretary in an office environment. This agent receives on-the-fly requests from other (human) agents for tasks such as: bring copies of a paper to David today, bring a book to George this afternoon, or warn Ronny immediately that a meeting is postponed. Each task requires a sequence of actions, such as: go to the location of the paper, get the paper, take the paper to the copy room, etc. The agent must plan its actions in real time in order to react effectively to the incoming stream of requested tasks.

Recent reactive approaches argue that fast feedback loops are sufficient to implement an intelligent reactive agent [Schoppers 87] [Agre & Chapman 87] [Maes 90] [Nilsson 92]. Although reactivity is certainly vital for agents in unpredictible environments, we argue that reasoning about perceived and inferred conditions is an equally important, complementary function of intelligent agents.

Generating plans must meet hard constraints. First, it must be done fast enough --- for an agent that receives a stream of asynchronous requests, planning time counts. For example, the agent might integrate new goals into its existing plan and evaluate the required modifications of its future activities before achieving a new or pending goal becomes irrelevant. Thus, it must plan and execute actions to warn Ronny before he leaves for the postponed meeting. Second, actions must be described expressively enough to cover the physical characteristics of the agent. Planned actions are decisions to activate robotics routines that exploit the available sensors and effectors, such as navigation routines (following a wall, a corridor) or paths planners based on metric maps (e.g., potential field [Latombe 91]). The planner must take into account the available routines and capacity limitations. For example, perhaps the robot cannot carry more than 3 books at once.

As Chapman [87] has shown, even with a simple action formalism, generation of correct and complete plans (requiring search) is an NP-complete problem. Thus a planning agent might spend an unbounded amount of time (exponential in the number of goals) deliberating on its future activities, while the outside world may changes and require attention --- the robot actually may be stuck planning actions that already are irrelevant. It also is known though that "typical" planning problems can be solved in polynomial time. Formulating problems using an adequate representation could lead to small search spaces and heuristics for sensibly scanning such spaces [Laurière 78]. Since the first robotics application [Fikes & Nilsson 71], many planners have been built [Sacerdoti 75] [Tate 77] [Wilkins 89]. However, there planners are not transferrabable and do not easily fit into a mobile robot,which must also sense its environment and act upon it. But since [Chapman 87], relatively few precise planning models have been proposed [McAllester & Rosenblitt 91] [Penberthy & Weld 92]. These systems are more general, but they run on toy domains; their action representation is too simple for a robotic agent.

This paper presents a planner adapted to mobile robots in office environments. Our contribution includes extensions of the action model and efficiency concerns. We will first sketch the algorithm of the planner. We then present extensions of the action model about locations (to allow plan interuptibility), arithmetic equations on integers (to describe practical capacity limitations of the robot) and variables' domain (to improve efficiency). We then present

[1] I would like to thank my advisor, Barbara Hayes-Roth, for guidance and support. Thanks to my colleagues on the ABots project for many fruitful discussions. This work has been supported by a Lavoisier fellowship from the Ministère des Affaires Etrangères and by Teknowledge Federal Systems, Contract 71715-1 under ARPA contract DAAA21-92-C-0028.

heuristics and describe experiments with a mobile robot on different scenarios. Finally, we briefly sum up our results.

2. Algorithm

The planner's algorithm takes as input an empty plan (the planning problem) and attempts to produce a solution plan. Using a robotics analogy, on each cycle, the algorithm considers the current plan with its software sensors, makes a decision with its heuristics, and modifies the plan with its software effectors. We describe these three algorithms --- it is assumed that the reader is familiar with [Chapman 87] and [McAllester & Rosenblitt 91].

2.1. Generating Plans

The following algorithm is used to generate a plan composed of actions P, of unification constraints U, of temporal orderings O among actions. Given an initial situation `initial`, a final situation `finish` and a set of actions schemas T, GeneratePlan is initialy called with the arguments ({`initial`,`finish`},∅,{(`initial` < `finish`)},T, depth) --- depth is the maximal number of plan modifications. To preserve the uniformity of representation, `initial` (resp., `final`) constitutes the effects (resp., prerequisites) of a virtual initial (resp., final) action.

`GeneratePlan`(P, U, O, T, d)
1. For every prerequisite g of actions of P, evaluate `mtc`(g, P, U, O).
2. *(Solution-Plan)* If all prerequisites are satisfied, return the plan (P, U, O).
3. For every *unsatisfied* prerequisite g, compute `SatisfyGoal`(g, P, U, O, T).
4. Rate each pair (unsatisfied prerequisite, plan modification)
5. Take the highest rated plan modification and execute it by modifying P, U and/or O to P', U' and/or O' [while keeping the other plan modifications for future backtracking].
6. *(Recursive Call)* If the number of calls to GeneratePlan reaches the depth d, then backtrack, otherwise call `GeneratePlan`(P', U', O', T, d).

The `GeneratePlan` procedure is essentially a depth-first search in the space of plans. It first checks whether a solution-plan is reached by computing the truth value (`mtc` algorithm) of every prerequisite (line 1 and 2). `GeneratePlan` then considers the possible plan modifications (`SatisfyGoal` algorithm) of these unsatisfied goals (line 3), rates them according to its heuristics (line 4) and executes the best plan modification for the best goal (line 5). Then `GeneratePlan` is recursively called if the maximal depth *d* has not been reached.

The plan modifications include action addition (renaming the variables of an action schema of T and adding it to P), (non-)unification addition (adding a constraint (u = v) or (u ≠ v) to U, with u and v two variables of U) and time ordering addition (adding a precedence constraint (a1 < a2) to O, with a1 and a2 two actions of P) --- thus unification, separation, promotion and demotion are represented [Chapman 87].

To preserve completeness (i.e., finding a solution when there is one), the depth of the search is bounded (parameter *d*) and a stack of possible / executed plan modifications is maintained (line 5): backtracking can then occur and lead to search on other branches. The depth is increased if no solution is found within a given depth. The number of calls to `GeneratePlan` (line 6) is measured by the length of this backtrack stack --- empty backtrack frames are allowed in line 5.

2.2. Observing Plans

The `mtc` algorithm checks that a prerequisite g is satisfied in a plan (P, U, O) for all instantiations of variables of U and for all total orders compatible with O. It uses a modal truth criterion: in a plan (P, U, O), a prerequisite g is necessarily true iff it is necessarily established in (P, U, O) and iff each time there is a potential effect c (a *clobberer*) that clobbers g, there is an intervening action w (a *white-knight*) that clobbers c and thus rescues the truth value of g in (P, U, O) --- see [Chapman 87] for the rigorous formulation.

Since the plan is also modified according to this criterion (see next section), this planner is correct --- a generated plan actually is a solution to the initial planning problem. A second interesting property of this criterion is that for each prerequisite, it computes the set of possible / necessary establishers and the set of possible / necessary clobberers (see next section).

Its main drawback (apart from those noted in [Pednault 91]) is its high complexity, $O(n^3)$ with n the number of actions of P --- for example, avoiding this computational cost is one *raison d'être* of SNLP's systematicity. The complexity of the global algorithm can be reduced, though, with the following property:

Property 1: *For a particular prerequisite of an action, the number of times the modal truth criterion needs to be called depends on the number of actions in the (future) solution-plan only.*

Proof (sketch): To see this, first notice that the agent will be interested in executable plans --- fully instantiated and linear plans. This means that the modality of the truth value of prerequisites must be necessity: a prerequisite must be true for all variables' instantiations and for all total orders among actions. Secondly, remember that the plan modification operations are action addition, unification addition, non-unification constraint addition and time ordering addition. If a prerequisite holds for all variables' instantiations, it holds *a fortiori* for the one specified in a plan modification like unification or non-unification addition. Similarly, if a prerequisite holds for all total orders, it holds *a fortiori* for the one specified by a plan modification like an ordering addition. Therefore the only

plan modification that might change a necessary truth value of a prerequisite is action addition ❑

Conversely, effects of a newly introduced action might clobber previously satisfied prerequisites while its own prerequisites might be unsatisfied (the number of unsatisfied prerequisites increases); its effects might not change previously satisfied prerequisites and its prerequisites might be already satisfied (the number of unsatisfied prerequisites decreases). Therefore regressing the mtc on the action schemas themselves is needed to predict some changes made by an action addition to the truth value of prerequisites [Knoblock 90].

Thus, the number of times the mtc must be called does not depend on the number of calls to `GeneratePlan` --- and not, in particular, to promotion, demotion, unification of separation. Thus, line 1 of `GeneratePlan` can be changed, while preserving completeness, to evaluate `mtc`(g, P, U, O) *only if an action addition has been performed in the last plan modification*. This additional condition is checked by considering the label of the topmost frame of the stack of plan modifications (kept in line 5 of GeneratePlan).

2.3. Modifying Plans

The `SatisfyGoal` algorithm returns a list of descriptions of all possible plan modifications that satisfy prerequisite g in a plan defined by (P, U, O) and by using the action schemas T:

`SatisfyGoal`(g, P, U, O, T)
Let E = (E_{poss},E_{nec}) the set of possible and necesssary establishers of g. Let C = (C_{poss}, C_{nec}) the set of possible and necessary clobberers of g.
1. *(Establishment)* If $E_{nec} = \emptyset$
  1.a. If $E_{poss} \neq \emptyset$, for all establisher e in E_{poss}, return the missing constraints (e = g) and/or (e < g).
  1.b. Otherwise, for all possible establisher from the set of action operators T, instantiate it into an action e and return `SatisfyGoal`(g, P ∪ {e}, U, O, T).
2. *(Declobbering)* If $C_{poss} \neq \emptyset$, for all clobberer c of C_{poss}, return the constraints g < c and/or g ≠ c (promotion and/or separation).
3. Otherwise, if $C_{nec} \neq \emptyset$, for all clobberer c of C_{nec}, do:
  3.a. If $E_{poss} \neq \emptyset$, then for all establisher w of E_{poss} such that not(w < c), return unification constraints such that (w = g) in U ∪ {g = c}, and/or ordering constraints such that c < w and w < g.
  3.b. Otherwise, for all possible establisher from the set of templates, instantiate it into an action of P and return `SatisfyGoal`(g, P ∪ {e}, U, O, T).

`SatisfyGoal` looks for an establisher of g, either existing (line 1.a) or new (line 1.b). It also looks for avoidable clobberers (line 3.a), or for existing (line 3.a) or new (line 3.b) white-knights --- white knights are establishers with additional constraints, thus are searched in E_{poss} and E_{nec} (which are already computed by the `mtc` algorithm). This algorithm is called until it returns no plan modification. A plan modification returned by `SatisfyGoal` can be a combination of primitive ones, but is always explicit --- for the condition, derivated from property 1 above, to be triggerable.

3. Action Language

Following [Fikes & Nilsson 71] and [Pednault 86], an action is expressed as prerequisites, effects and *preservation conditions --- conditions that must hold during* the action execution (e.g., (non-)unification constraints). Added and retracted effects are represented uniformly by adding a sign to literals: positive for addition, negative for retraction. The action formalism is extended regarding locations, arithmetic equations and possible objects.

3.1. Locations

In traditional planners, a motion is represented by an operator schema (`MOVE ?point-A ?point-B`); it is essentially composed of one prerequisite, stating that the robot was there, and two effects, one that states that the robot is here now, and another one that states that the robot is not there any longer. Then an executable plan is a sequence of `MOVE` actions and static ones (e.g., take a book from Michelle, move to David's office, give the book to David). Although a motion can be represented this way in our planner, such a `MOVE` action is not *desirable* in a mobile robot for the following reasons.

- Moving constitutes the main activity of a mobile robot. Other components of the agent (e.g., path planners, navigation routines) completely rely on correct locations [Latombe 91] --- giving them a wrong initial location leads to an erratic motion of the robot.
- The current plan of the agent can be interrupted by another plan that achieves a more urgent task (e.g., the phone is ringing, therefore a plan is quickly generated to answer the phone). Thus the execution of the interrupted plan may resume from a different location.
- Using a planner to perform a path search on a topological map is inefficient, when compared to other dedicated search methods [Latombe 91].

To fulfil these requirements, each action of the plan is considered as an *intentional* action that might happen at an *intended* location: our planner does not use any `MOVE` action schema; the location of an action is stored in a particular field of the action (":location") and can be bound to variables of prerequisites. Actions that can be performed at any location (e.g., sending an email) are considered to occur at a location called *anywhere*.

This use of intentional actions relies on the following assumption: *it is always possible for the robot to move to a different place without changing the structure of the plan.* This is true in the indoor environment in our scenarios. But consider an outdoor environment where distance tasks might be required: if I want to ski at lake Tahoe this weekend, which is 200 miles away from San Francisco, I will have to add actions such as filling up the tank of my

car and driving. These two actions only depend on the distance from here to lake Tahoe --- if I lived in a ski resort, I would only have to walk to the lifts, without taking my car. Although the previous assumption holds in our case, we are currently investigating this reasoning with the integer representation (see next section).

To specify a location for each action, a topological map of the environment is stored in the initial situation and a prerequisite of each action matches with that map. For example, the action `(TakeDocument ?doc ?person ?location ?space-left)` contains a prerequisite `(AT ?person ?location)` and the variable `?location` is put into the location field (see below). Since the agent does not move any person at planning time, the topological map is not changed by the plan.

To optimize the path of a (nonlinear) solution-plan, actions involving the same location are regrouped by adding time ordering constraints: if action A1 (at location L1) immediately precedes action A2 (at location L2) and L1 = L2, then consider all actions A3(L3) such that L3 ≠ L2 and non-deterministically add constraint A2 < A3 or A3 < A1. (Choosing which one relies on the global evaluation function of distance minimization.) Since Chapman's declobbering by White Knight clause (see mtc in 2.2.) would be sure to generate longer paths (actually actions A1 and A2 are performed without any motion of the robot), this rule can be considered as an adaptation of SNLP's threat removal operation to our location representation.

3.2. Capacity limitations

Representing numbers is difficult in planning traditional planners, because (1) literals in prerequisite/effect are only (possibly negated) relations among constants or placeholders, and (2) a little increase in the action formalism leads STRIPS-like planners to generate incorrect plans [Lifschitz 86]. But consider, for example, a mobile robot that can carry no more than 3 documents at a time: if it is asked to carry a fourth document, it will have to put one of the others down and then deliver at least one of the three it is holding before coming back later to retrieve the dropped document. In this case, the existence of two actions (putting down a document, coming back later to grab it) only depends on this capacity limitation.

Integers can be represented by iterating a symbol in a literal and by including the occur check clause in the unification algorithm of the planner. The integer *n* is encoded as `(+1 (+1 ... (+1 0) ...))` with `0` a (constant) symbol, `+1` a symbol (positionally used as a function) which is iterated n times over `0`. Capacity limitations can then be represented by using counters: For example, the `TakeDocument` action schema has `(NumDocSpaces (1+ ?space-left))` as a prerequisite, and `(NumDocSpaces ?space-left)` as an effect: thus the counter `NumDocSpaces` is decremented by one when the action is executed [Jacopin 93]. Switching the two expression would increment this counter. As an illustration, the following action schema states that the robot can take a document `?doc` from `?person` at `?location`, if there still at least one space left (`?sl`) on the robot:

```
(deftemplate (TakeDocument ?doc ?person ?location ?sl)
  :prerequisites ((:NOT (hold Robot ?doc))
    (hold ?person ?doc) (at ?person ?location)
    (NumDocSpaces (1+ ?sl)))
  :preservations ((:NOT (:= ?person Robot)))
  :effects ((hold robot ?doc) (:NOT (hold ?person ?doc))
    (:NOT (NumDocSpaces (1+ ?sl))) (NumDocSpaces ?sl))
  :location      ?location)
```

The maximal number of documents that can be carried at once by the robot (3 in our initial example) is specified by an effect in the initial situation: `(NumDocSpaces (1+ (1+ (1+ 0))))`.

Arithmetic equations such as "x = y + constant" and "x = y - constant" are maintained by using unification constraints among composite variables --- a plan modification such as `(= (NumDocSpaces ?x) (NumDocSpaces (1+ (1+ (1+ ?y)))))` encodes the equation "?x = ?y + 3" relating to the `NumDocSpaces` relation. The four basic arithmetic operations, for example, can bencoded this way in our planner.

3.3. Domains

To copy a document on slides, the robot might know in advance the possible locations where slides can be found. We represent this knowledge as a domain of the variables used by the planner. An advantage of this is that passing constraints around might quickly reduce the domain of some variables and thus lead to fast instantiations --- a constraint satisfaction view of search space reduction in the planning process.

A domain of a variable is stated as follows: When a prerequisite is established by several effects of the virtual initial action using a unification constraints, and when the same variable appears in all these constraints, these plan modifications are considered to define a domain for that shared variable. An example of this is *type definition*: the unary relation `SLIDE` might define the available slide pads iff (1) the initial situation includes `(SLIDE S1) (SLIDE S2) (SLIDE S3)` and (2) no action schema creates slides --- no constant creation about this relation. Then the prerequisite `(SLIDE ?S)` can be satisfied by plan modifications. The previous rule then defines `(S1 S2 S3)` as a domain for `?S`. The separation plan modification will then remove values from domains; When there is only one value left in a domain, this value will be taken as the value of the corresponding variable. Practically, it involves sequentially performing two plan modifications (line 5 of `GeneratePlan`): a unification constraint right after a non-unification one.

Two caveats must be stated though: First, contrary to typical Constraint Satisfaction Problems, a variable's' domain might be augmented --- an action might be addedbefore a prerequisite that defines a variable's domain, *after* that domain is defined during the planning process.

Inferring a variable's value with the last value of a domain might then lead to early instantiations. Therefore we assume that no constant is created on the fly by an action schema. In our application, it means knowing in advance all the possible objects the agent might have to deal with --- which is reasonable. Second, since the modal truth criterion does not involve variables' domains, a prerequisite might be considered to be true by considering domains, whereas the `mtc` algorithm would return false --- as noted in [Chapman 87]. It is not clear though how much efficiency would be gained by such an additional reasoning. Our experiments with variables' domains on real cases (see section 5.3.) prove that substantial performances can already be reached.

4. Heuristics

The performance of the planner depends on the order in which unsatisfied prerequisites and plan modifications are selected. The rating of plan modifications (line 4 of `GeneratePlan`) uses the following heuristics:

- *Delay action addition as long as possible*. This heuristic proposes to add an action only when there is no other way to satisfy a prerequisite. Intuitively, all the relevant information should be extracted from a plan before adding new actions. Given property 1, this heuristic also avoids high branching factors.
- *Instantiate a variable when there is one value only.* This heuristic proposes to take a value when there is actually no other one. It is based on the use of previously described domains and assumes that even if that variable is not instantiated, going further into the planning process would not create a second value for this variable.
- *Solve local conflicts first.* This heuristic proposes to sort conflicts by increasing the distance clobberer / prerequisite (measured in the number of actions). Intuitively, local conflicts might interfere with long conflicts, and considering their resolution locally might have fewer possibilities.

5. Real Agent

5.1. Domain

A planner, including the previous features and based on [Morignot 91], has been implemented[1] to control a Nomadics 200 mobile robot [Zhu 92]. The modeled agent must perform some tasks requiring motion and communication in an indoor environment (actually the corridors and offices of the first floor of the K.S.L.). Physically, the robot can rotate its wheels, change their direction, rotate a turret that supports a CCD camera for image recognition, and use a voice synthetizer. Its sensors include 16 sonars, 18 infra-red, 18 bumpers and 1 laser with its associated (second) CCD camera. Sensors and effectors are controled by an on-board PC 486 or by a remote workstation via a radio modem.

In the first scenario, the robot has to survey a level of an office building and respond to alarms (e.g., phone ringing, broken window, open door) according to their nature and urgency. A second set of scenarios involve assisting a secretary with tasks such as delivering documents, copying papers, sending email or verbally warning people.

5.2. Monitoring

The agent monitors the execution of its current plan by moving the virtual initial action ("now") forward in time:

1. According to its actual state and needs, the agent chooses one action to be the next virtual action. If needed, a precedence constraint is put between this chosen action and its siblings (related to the virtual initial action).
2. The action chosen for execution is removed from the plan. Effects of this action are added to those of the virtual initial action, previous effects of the initial action that are mentioned in the chosen action are removed (their truth value changes or is subsumed by one specified by a new effect).
3. The chosen action is sent to robotics routines of the robot for actual control of the sensors and effectors.

The present document focuses on the plan generation capability: For more details about the general role of plans and intentions in the overall agent architecture, see [Hayes-Roth et al. 93].

5.3. Performances

This planner is used in two sets of scenarios (an alarm domain, an office factotum domain) and has been tested in standalone on other domains: blocks domain, towers of Hanoi. Tests were carried out on a SpacStation 10 using Lucid Common-Lisp 4.1 and a source-code compiled in development mode (see table 1).

Domain	Time	Actions	Cycles
Blocks	0.17	3	10
Hanoi	0.70	7	29
Alarm	0.42	5	12
Office	0.37	6	17

Table 1: Performance of the planner.

Although performances get worse as the problem size increases, plan generation in usual cases (i.e., actually happening in our scenarios) takes less than one second in real time.

As a second result, the solution plans are usually close to the root of the search space (less than 30 plan modifications, including less than 10 action additions). Thus depth-first search is appropriate to represent *planning choices*: it has been reported for example that chess players explore a single path at ever greater depth and repeatedly return to the initial state [de Groot 65].

In practice, given that the maximal speed of the robot is roughly 10 inches per second, the world model might be at most 10 inches inaccurate between the start and the end of

[1] The source-code is available by request to the author.

the plan generation process. That's actually half the size of the robot itself, and the navigation routines of the robot already keep a half diameter security interval distance from known objects.

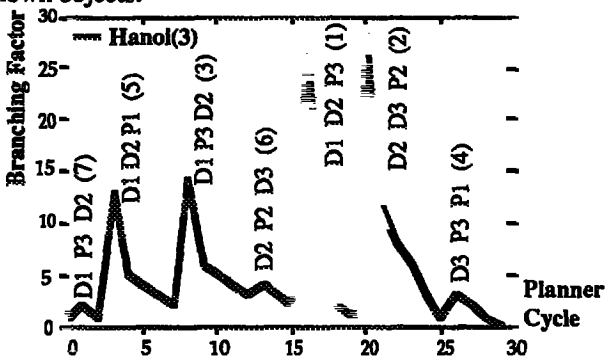

Figure 1: Size of the search space.

To illustrate the way this planner scans the search space, we consider the *pons asinorum* in planning, the towers of Hanoi. To evaluate the shape of the search space, figure 1 shows the branching factor of `GeneratePlan` (i.e., the number of plan modifications) as a function of the number of recursive calls (i.e. the planner's cycle). First, the branching factor is not constant during plan generation (mean = 6.1, variance = 6.8): Each maximum corresponds to an action addition, the prerequisites of which are usually not satisfied (mainly because of fresh variables to bind). Between two action additions, heuristics 2 and 3 add constraints to the plan, which is guaranteed not to increase the branching factor (according to property 1). When these two heuristics cannot be triggered, the plan is either a solution (as in cycle 29) or heuristic 1 is triggered again to generate a new maximum (as in cycle 1, 3, 8, 13, 16, 20, 26). As a result, the example of figure 1 is carried out without backtracking. What is surprising, though, is the fast decrease right after a maximum --- more than one prerequisite is satisfied with one plan modification. Thus previous heuristics choose to execute the plan modifications that satisfy the largest number of prerequisites to keep the branching factor low.

6. Conclusion

A planner that successfully generates plans in real-time to control an autonomous agent has been presented. Handling the practical characteristics of an embedded system has led us to extend the action model to represent arithmetic equations, and to consider intentional actions to represent locations. Efficiency concerns while preserving completeness have led us to represent variables' domains. An implementation of this planner, that exhibits attractive empirical real-time performances, is currently used to generate plans on the fly in an application involving a mobile robot in our laboratory. Our ongoing work focuses on representing collaboration / competition among multiple agents using motivations and intentions.

References

Agre, P., and Chapman, D. 1987. Pengi: An Implementation of a theory of Activity. In Proceedings of AAAI'87, Seattle.

Chapman, D. 1987. Planning for Conjunctive Goals. Artificial Intelligence 32(3):333-377.

de Groot, A. D. 1965. Thought and choice in chess. The Hague: Mouton.

Fikes, R., and Nilsson, N. 1971. Strips: A new Approach to the Application of theorem Proving to Problem Solving. Artificial Intelligence 2(3/4):198-208.

Hayes-Roth, B.; Lalanda, P.; Morignot, P.; Balabanovic, M.; Pfleger, K. 1993. Plans and Behavior in Intelligent Agents. Tech. Rep. KSL-93-42, Stanford Univ.

Jacopin, E. 1993. A Look at Function Symbols in Planning. AAAI Spring Symposium on Planning, Stanford Univ.

Kambhampati, S. 1992. Characterizing Multi-Contributor Causal Structures for Planning. In Proceedings of AIPS'92, College Park.

Knoblock, C. 1990. Learning Abstraction Hierarchies for Problem Solving. In Proceedings of AAAI'90, Boston.

Latombe, J.- C. 1991. Robot Motion Planning. Kluwer Academic Pub., Boston.

Laurière, J.-L. 1978. A Language and a Program for Stating and Solving Combinatorial Problems. Artificial Intelligence 10(1):29-127.

Lifschitz, V. 1986. On the Semantics of STRIPS. In Reasoning about Actions & Plans, Timberline, Georgeff & Lansky eds., Morgan Kaufmann.

Maes, P. 1990. Situated Agents can have Goals. Robotics and Autonomous Systems Journal. Special issue on Designing Autonomous Agents 6 (1&2).

McAllester, D. and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In Proceedings of AAAI'91, Anaheim.

Morignot, P 1991. Truth Criteria in Planning. Ph.D. diss., C. S. Dept., ENST, Paris. (in French)

Nilsson, N. 1992. Towards Agent Programs with Circuit Semantics. Tech. Rep. CS-1412, Stanford Univ.

Pednault, E. 1986. Towards a Mathematical Theory of Plan Synthesis. Ph.D. diss., Elec. Eng. Dept., Stanford Univ.

Pednault, E. 1991. Generalizing Nonlinear Planning to Handle Complex Goals and Actions with Context-Dependent Effects. In Proceedings of IJCAI'91, Sydney.

Penberthy, S., and Weld, D. 1992. In Proceedings of KR'92, Cambridge.

Sacerdoti,, E. 1975. The Nonlinear Nature of Plans. In Proceedings of IJCAI'75, Tbilisi.

Schoppers, M. 1987. Universal Plans for Reactive Robots in Unpredictible Environments. In Proceedings of IJCAI'87, Milan.

Tate, A. 1977. Generating Project Networks. In Proceedings of IJCAI'77, Boston.

Wilkins, D. 1988. Practical Planning: Extending the Classical Planning Paradigm. Morgan Kaufmann.

Zhu, D. 1992. Nomadic Host Software Development Environment. Nomadic Technologies Inc, Mountain View.