Appears in the proceedings of AAAI-91 (pp. 640-645).

An Efficient Reactive Planner for Synthesizing Reactive Plans

Patrice Godefroid[*] and **Froduald Kabanza**[†]
Université de Liège
Institut Montefiore, B28
4000 Liège Sart-Tilman, Belgium
Email: {godefroid, kabanza}@montefiore.ulg.ac.be

Abstract

We present a nonlinear forward-search method suitable for planning the reactions of an agent operating in a highly unpredictable environment. We show that this method is more efficient than existing linear methods. We then introduce the notion of safety and liveness rules. This makes possible a sharper exploitation of the information retrieved when exploring the future of the agent.

Introduction

Classically, a plan is a set of actions to guide an agent from its current situation to another situation called the goal. If the result of these actions is not always the expected one, the agent is said to be operating in an unpredictable environment. Under this assumption, the agent may be deviated at any time from the intermediate situations expected in its plan. Whenever there is such a deviation, the agent has to replan from its new current situation. In real-time applications, the agent does not always have the time to replan. This prompted the development of new agent architectures where the agent senses the environment and then executes actions specified in a *reactive plan*. A reactive plan is equivalent to a set of *rules* of the form $S \rightarrow R$ where S is a set of facts describing a *situation* of the environment and where R is a set of actions describing a *reaction* of the agent when placed in situation S.

Unfortunately, in highly unpredictable environments, it is unfeasible to produce and to store reactions for all possible situations. Some recent work propose providing planning capabilities to the agent at execution time [Georgeff and Lansky, 1987, Drummond and Bresina, 1990]. Thus the agent is able to sense the environment and then to compute reactions. This type of planner is said to be a *reactive planner*.

Methods for synthesizing reactive plans can be characterized by how the situations for which they provide reactions are determined. A forward-search method as in [Drummond and Bresina, 1990] starts from the current state of the agent and produces a reaction for each situation leading to the goal. A backward-search method as in [Schoppers, 1987] starts from the goal and produces a reaction for each explored situation since all these situations lead to the goal. The backward-search methods are not really suitable for reactive planning since the current situation of the agent can not be taken into account to control the search. If the agent operates in a highly unpredictable environment and has to react within short time constraints, the forward-search approach enables the search to continue from the current situation if no rule corresponding to this situation is available in the reactive plan currently being constructed. The exploration of the most likely near future of the agent can thus be privileged.

Existing forward-search methods [Drummond, 1989, Drummond and Bresina, 1990] are linear, i.e. they explore possible action *sequences* of the agent. The execution of independent actions (i.e. actions involving disjoint sets of facts) is modeled by all possible interleavings of these actions. The number of interleavings can be very large. This combinatorial explosion limits both the applicability and the efficiency of linear methods.

In this paper, we present a simple forward-search method for exploring the future of an agent without incurring the cost of modeling independent actions by their interleavings. Our method yields results identical to those of methods based on interleaving semantics, it just avoids most of the associated combinatorial explosion. This method can

[*]The work of this author is supported by the European Community ESPRIT BRA project SPEC (3096).

[†]The work of this author is supported by the Belgian National incentive-program for fundamental research in artificial intelligence initiated by the Belgian State - Prime Minister's Office - Science Policy Programming.

be qualified as nonlinear though it differs substantially from classical nonlinear backward-search methods such as NOAH [Cohen and Feigenbaum, 1982], SIPE [Wilkins, 1984], TWEAK [Chapman, 1987].

Our search method is based on recent results in concurrent system verification. In [Godefroid, 1990] it is shown that the global behavior of a set of communicating processes can be represented by an automaton which can be much smaller than the usual global automaton representing all possible interleavings of concurrent actions. The method is justified by using partial-order semantics, namely the concept of Mazurkiewicz's trace [Mazurkiewicz, 1986]. The basic idea is to build an automaton called *trace automaton* that only generates one interleaving of each concurrent execution. A *trace* is defined as an equivalence class of sequences and corresponds to a possible partial ordering of actions the system can perform. Together with an independence relation on actions, a trace automaton fully represents the concurrent executions of the program. The practical benefit is that this automaton can be much smaller than the automaton representing all interleavings: it requires less memory to be stored and can be computed more quickly. It can be used successfully for verification purposes [Godefroid, 1990, Godefroid and Wolper, 1991a, Godefroid and Wolper, 1991b].

In this paper, we build upon this work and develop a new nonlinear forward-search method suitable for reactive planners. We show that this method is more efficient than linear ones. Then we show how to automatically produce reactive plans. We introduce the notion of safety and liveness rules. *Safety* rules are designed to protect the agent from reaching irrecoverable situations, i.e. situations from which the goal is not reachable. Complementarily, *liveness* rules guide the agent towards its goal. Introducing safety rules enables one to exploit more sharply the information retrieved during the search.

Our reactive planner consists of three concurrent processes. One process investigates the future of the agent using our new search method. The two other processes work on the explored part of the future of the agent and produce respectively safety and liveness rules. In the next section we define the representation of actions we use. After describing the search process, the automatic generation of safety and liveness rules is presented. Then the management of these rules is discussed. Finally, we conclude the paper with a comparison between our contributions and related work.

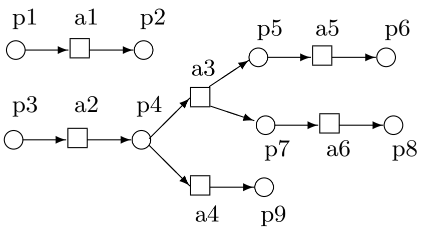

Figure 1: A Plan Net

Action Representation

An action is described by its preconditions and effects (actions can be characterized by global action-schemata called operators). Preconditions and effects are facts about the environment of the agent. Facts can be true or false. A set of actions can be represented by a graph usually called contact-free one-safe Place/Transition-Net [Reisig, 1985] or plan net [Drummond, 1989] for short. Such a graph is built from two types of nodes: actions and facts.

Figure 1 shows a plan net representing a set of actions of an agent. In the figure, squares represent actions and circles represent facts. An arrow from a fact f to an action a means that f is a precondition for a. In contrast, an arrow from an action a to a fact f means that f is a an effect of a. We note $pre(a)$ the set of preconditions of a, and $post(a)$ the set of effects of a. We define a *state* as a set of facts. An action a is *enabled* in a state S iff $pre(a) \subseteq S$. If an action a is enabled in state S, this action a can be *applied* and yields a *successor* state S' such that $S' = (S - pre(a)) + post(a)$, what disables all the preconditions of the action and enables all its effects (note that the set of preconditions and the set of effects of an action may be nondisjoint).

There are other ways to represent actions [Cohen and Feigenbaum, 1982]. Just note that every planning problem expressed in these frameworks can be encoded in the framework presented here. Anyway, the results of this paper could be adapted to other action representations.

For convenience, we assume that the goal is a conjunction of facts describing *totally* the state that the agent has to reach. Let N be the plan net representing the set A of actions of the agent. We define the *dependency* in the plan net N as the relation $D \subseteq A \times A$ such that: $(a_1, a_2) \in D \Leftrightarrow (pre(a_1) + post(a_1)) \cap (pre(a_2) + post(a_2)) \neq \emptyset$. The complement of D is called the *independency* I in N. If $(a_1, a_2) \in I$, a_1 and a_2 are said to be independent actions.

```
search($N$,$S$,$G$) {
  $A$ = select_actions($N$,$S$);
  for $a$ in $A$ {
    $S' = (S - pre(a)) + post(a)$;
    if $S'$ is not in $G$ then {
      $G$ = add_state($S'$,$G$);
      $G$ = add_labeled_arc($S$,$S'$,$a$,$G$);
      $S'.sleep$=sleep_attached_with($a$);
      search($N$,$S'$,$G$);
      }
    else $G$ = add_labeled_arc($S$,$S'$,$a$,$G$);
    }
  }
```

Figure 2: Search Method

The Search Process

The aim of the search process is to investigate the future of the agent from its current state. With linear methods, the future of the agent is represented by all possible action sequences the agent is able to perform. In order to avoid the combinatorial explosion due to the modeling of the application of independent (concurrent) actions by their interleavings, we present a new search method where the future of the agent is described in terms of partial orders rather than sequences. More precisely, we use Mazurkiewicz's traces [Mazurkiewicz, 1986] as semantic model. We briefly recall some basic notions of Mazurkiewicz's trace theory.

Traces are defined as equivalence classes of sequences. A trace represents a set of sequences defined over an alphabet A that only differ by the order of adjacent symbols which are independent according to a dependency relation D. For instance, if a and b are two actions of A which are independent according to D, the trace $[ab]_{(A,D)}$ represents the two sequences ab and ba. A trace corresponds to a partial ordering of actions and represents all linearizations of this partial order.

Given an alphabet and a dependency relation, a trace is fully characterized by only one of its sequences. Thus, *given the set of actions A and the dependency relation D defined as in the previous section, the future of the agent is fully investigated by exploring only one interleaving for each possible partial ordering (trace) of actions the agent is able to perform.*

We present in Figure 2 an algorithm for performing this exploration. The procedure **search(N,S,G)** accepts a plan net N, an initial state S, and a graph G (which initially amounts to the state S) as arguments. The function **add_state(S,G)** returns the graph which results from adding S to G. The function **add_labeled_arc** (S,S',a,G) returns the graph which results from adding an arc in G from S to S', labeled with the action a.

This algorithm looks like a classical exploration of all possible action sequences. The only difference is that, instead of applying systematically *all* actions enabled in a state, we choose only *some* of these actions to be applied in order to explore only one interleaving of enabled independent actions. The function **select_actions** described in Figure 3 performs this selection. For doing so, a "sleep set" is associated with each state reached during the search.

The sleep set associated with a state S (denoted $S.sleep$) is a set of actions that are *enabled* in S and that *will not be applied* from S. The sleep set associated with the initial state is the empty set. As shown in Figure 3, one sleep set is attached to each action selected to be applied: this is performed by the procedure **attach(A,$Sleep$)** which attaches to each action a in the set A the actions of *Sleep* that are not in conflict with a. The sleep set determined for a selected action a will be associated to the state $S' = (S - pre(a)) + post(a)$ reached after the application of a. Sleep sets are introduced to deal with "confusion cases" and their use is described below.

In what follows, actions $a_1, a_2, \ldots, a_n$ are referred to as being *in conflict* iff $(pre(a_1) \cap pre(a_2) \cap \ldots \cap pre(a_n)) \neq \emptyset$. An action which is not in conflict with any other action is said to be *conflict-free.* The function **conflict(a)** returns the actions that are in conflict with a. The function **enabled(N,S)** returns all enabled actions in state S.

The basic idea for selecting among the enabled actions those that have to be applied is the following. Whenever several independent actions are enabled in a given state, we apply only one of these actions. Such a way, we explore only one interleaving of these independent actions. Checking if two enabled actions are independent can be done by checking if they are not in conflict. When enabled actions are in conflict, their occurrences are mutually exclusive and lead to different traces (partial orderings of actions) corresponding to different choices that can be made by the agent.

Let us now describe precisely the function **select_actions** which implements the action selection. If there is an enabled action which is conflict-free, only this action is selected. (Note that if there are several conflict-free enabled actions, only one of these actions is selected, it does not matter which one.) Else, if there is an enabled action that is in conflict exclusively with enabled actions, we select this action and the actions that are in conflict with it in order to explore all possible traces (it corresponds to a branching in the graph).

```
select_actions($N$,$S$) {
  $A$ = enabled($N$,$S$)$-S.sleep$;
  if $\exists a \in A$ :conflict($a$)= $\emptyset$ then
    return(attach($a$,$S.sleep$));
  if $\exists a \in A$ :conflict($a$)$\subseteq$ enabled($N$,$S$) then
    return(attach(($a$+(conflict($a$)$\cap A$)),$S.sleep$));
  newSleep= $S.sleep$;
  result= $\emptyset$;
  while $A \neq \emptyset$ {
      $a$ =one_element_of($A$);
      result=result+attach(($a$+(conflict($a$)$\cap A$)),
                                    newSleep);
      newSleep=newSleep+$a$+(conflict($a$)$\cap A$);
      $A = A - a-$(conflict($a$)$\cap A$);
      }
  return(result);
  }
```

Figure 3: Selection amongst Enabled Actions

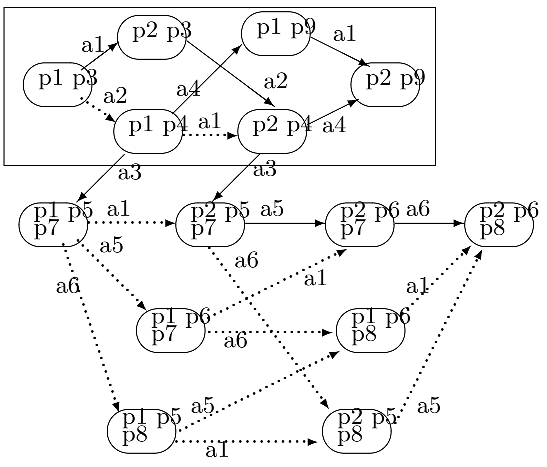

Figure 4: Comparison between G and G'

Finally, if there remain only enabled actions that are in conflict with at least one nonenabled action (this is a situation of *confusion* [Reisig, 1985]), we proceed as follows.

Let a be one of these enabled actions. We know that a is in conflict with at least one action x that is not enabled in the current state. But it is possible that x will become enabled later because of the application of some other actions independent with a. At that time, the application of a could be replaced by the application of x. We have to consider all possible cases: we select a and the actions $a_1, \ldots, a_n$ that are enabled and in conflict with a (if any); and we also have to check if the application of a could be replaced by the application of x after the application of some other enabled actions *independent* with a. This is done by exploring all possible traces the agent can perform from the current state while preventing the application of the actions $a, a_1, \ldots, a_n$ which are *dependent* with a. To prevent the application of $a, a_1, \ldots, a_n$, we include these actions in the current sleep set. Then, we repeat the same procedure with a remaining enabled action and the new sleep set until all enabled actions have been considered. In summary, when all enabled actions that are not in the sleep set are in conflict with at least one nonenabled action, all these actions are selected but are attached with different sleep sets. This ensures that at least one interleaving for each trace is explored. Such a way, we prevent the search from "overshooting" the goal while still avoiding the construction of all possible interleavings of enabled independent transitions. Our method is complete, i.e. always finds the goal if this is possible (due to space constraints, correctness proofs of the algorithms are omitted here; for more details, see [Godefroid and Wolper, 1991b]).

The practical benefit of this search method is that, by construction, the constructed graph G' is a "sub-graph" of the usual graph G representing all possible action sequences of the agent. (By sub-graph, we mean that the states of G' are states of G and the arcs of G' are arcs of G.) Moreover, the time required for constructing G' with our algorithm is linear in the number of constructed arcs as is the case for G with the classical algorithm. Thus *the method presented here never uses more resources and can be much more efficient, both in time and memory, than the classical linear methods* (unless, of course, there is no simultaneous enabled independent actions during the search; then our method becomes equivalent to the linear ones).

Figure 4 shows the graphs G and G' corresponding to the plan net of Figure 1 when the initial state of the agent is $p_1 \wedge p_3$ and the goal to reach is $p_2 \wedge p_9$. The dotted part is not part of G'. One sees clearly that the combinatorial explosion both in the number of states and arcs is avoided using our method.

If the goal had been $p_2 \wedge p_3$, the search process would have missed it ($\{p_2, p_3\}$ is in the dotted part). Actually, before starting the search, we add in the plan net one action whose preconditions are the fact representing the goal and whose effect is a special fact "stop". This simple construction ensures that, if the goal is reachable, then it will be reached during the search.

Note that at any time, if the agent is placed in a state which is not in G', the current search can be delayed and a call to the procedure **search** can be done with the state of the agent as initial state. If this state is in G', the exploration of the traces leaving that state in G' can be privileged. Such a way, the most likely near future of the agent will be investigated first.

Generating Rules

Concurrently to the search process, two other processes compute respectively safety and liveness rules. Safety rules are designed to protect the agent from reaching irrecoverable states, i.e. states from which the goal is not reachable. Complementarily, liveness rules guide the agent towards its goal. This enables one to exploit more sharply information retrieved during the search: if the current trace being investigated by the search process leads to the goal, the search process communicates this trace to the process producing liveness rules; if the current trace being checked leads to an irrecoverable state, the search process communicates this trace to the process producing safety rules. Hence, whatever conclusion the search yields, one is able to produce rules, to refine the reactive plan and thus to help the agent to complete its goal. As for the search, the generation of rules is conducted incrementally as time proceeds. When placed in a state S, the agent looks for rules corresponding to S. If there is a liveness rule, it executes the actions specified by this rule. If not, the agent executes any set of enabled independent actions that are not violating a safety rule corresponding to S.

Whenever the process generating liveness rules receives from the search process a trace $[w]$ i.e. a partial ordering of actions, it expands this trace and computes liveness rules for every state in the expansion. The expansion of a trace is a graph representing all interleavings of the partial order corresponding to this trace. A liveness rule $S' \rightarrow R$ is produced for every state S of this expansion: R is the set of actions leaving S in the expansion (by construction, all actions in R are enabled and independent). S' is a subset of S. Indeed, as pointed out in [Drummond, 1989], not all facts in S are relevant to the success of the given trace. To find just those that are relevant, we scan along the trace, forming a conjunction of facts relevant to the actions encountered during the scan.

For the example of the previous section, the liveness rules produced are: $\{p_1, p_3\} \rightarrow \{a_1, a_2\}$, $\{p_1, p_4\} \rightarrow \{a_1, a_4\}$, $\{p_2, p_3\} \rightarrow \{a_2\}$, $\{p_2, p_4\} \rightarrow \{a_4\}$, $\{p_1, p_9\} \rightarrow \{a_1\}$. The states of the expansion of the trace $[a_2, a_1, a_4]$ appear in a box in Figure 4.

Whenever the process generating safety rules receives a trace $[w]$ leading to an irrecoverable state, it backtracks along the sequence w until there is a branching point in the plan net. This branching point corresponds to a backtracking state S of G'. Then it generates a safety rule $S \rightarrow \neg a$ for every action a that is enabled in S and belongs to w. For the example of the previous section, the only safety rule produced is: $\{p_2, p_4\} \rightarrow \neg a_3$.

Managing Rules

The process generating safety rules runs in parallel with the one generating liveness rules. If there already exists a liveness rule corresponding to a state, no safety rule is generated for it. On the other hand, when a liveness rule is generated for a state, all safety rules corresponding to that state are deleted.

We stated that liveness rules are generated for all states in the expansions of traces leading to the goal. As a matter of fact, rules are not necessary for some of those states.

Definition A *single critical state* is a state from which one can reach an irrecoverable state by the application of only one action. A *concurrent critical state* is a state from which one can reach an irrecoverable states by the application of several independent actions. A *safe state* is a state from which the application of any set of enabled actions leads to a state from which the goal is reachable.

Note that it is not necessary to generate rules for safe states because any execution from them leads to a state from which the goal is reachable. Hereafter, we describe how to distinguish critical states from safe states when the search process has terminated and when all expansions have been computed. Once one is able to make this distinction, it is possible to remove the rules corresponding to safe states.

We first show how to compute single critical states. By definition, a single critical state is one from which the application of one action leads to an irrecoverable state. Observe that the expansion of all traces leading to the goal (this expansion corresponds to the box in Figure 4) contains all states except irrecoverable states. Hence to compute single critical states, one looks for states in the box for which there is an enabled action labeling an arc leading to a state which is out of the box. In the example, the states $\{p_1, p_4\}$ and $\{p_2, p_4\}$ are single critical states. For the need of the computation of concurrent critical states, when we compute single critical states, we construct irrecoverable states that are direct successors of single critical states (i.e. states on the limit of the box). In the example, the state $\{p_1, p_5, p_7\}$ is constructed.

Now, we show how to compute concurrent critical states from single critical states. By definition, a concurrent critical state is a state in the box from which the application of a set of enabled independent actions leads to an irrecoverable state. Such an irrecoverable state is necessarily a successor of a single critical state. From an irrecoverable state just on the limit of the box, we determine all states accessible backwards, on a suffix of length > 1 labeled by independent actions. In the example, there is no such suffix for the irrecoverable state $\{p_1, p_5, p_7\}$. In contrast, for the irrecoverable state $\{p_2, p_5, p_7\}$, the suffix $\{p_1, p_4\} \xrightarrow{a_1} \{p_2, p_4\} \xrightarrow{a_3} \{p_2, p_5, p_7\}$ determines the concurrent critical state $\{p_1, p_4\}$. (It is also a single critical state).

Related Work and Conclusion

Various architectures address the problem of controlling reactive agents [Brooks, 1986, Kaelbling, 1987, Georgeff and Lansky, 1987, Nilsson, 1990]. Brooks' robot control system [Brooks, 1986] is layered into communicating functional units. Each unit implements robot behaviors and corresponds to a level of competence. Behaviors of a level constrain behaviors of lower levels. The behaviors are specified by the user using a Lisp-based specification language. Our rules correspond to a restricted form of behaviors specifiable in that language in assuming that behaviors of our agent are organized into three levels of competence: liveness rules correspond to the highest level, safety rules to the second, and nonplanned reactions to the lowest. Our contribution is that planned behaviors are synthesized automatically.

Kaelbling's hybrid architecture [Kaelbling, 1987] is based on ideas similar to that of Brooks, and uses also a language (Rex) to specify the control system.

Georgeff and Lansky's PRS system [Georgeff and Lansky, 1987] is a reactive planner. The reactivity of the system is driven by rules (Knowledge Area) more general than ours (the consequence of a rule can contain goals that must be decomposed). However, these rules are supplied by the user, and the planner only decomposes goals activated by a rule into primitive actions like a classical planner.

Nilsson [Nilsson, 1990] proposes a similar system controlled by a network (Action Unit) specified in a language (ACTNET) or specified by a Triangle Table synthesized by a planner (see below).

Our work was influenced by that of Drummond [Drummond, 1989]. Noticeable common points are the representation of actions by plan nets, the critical states, and a forward search. We differ essentially on three aspects. One aspect is that we use a nonlinear search. We expand only traces leading to the goal, hence we save time and memory by not exploring the interleavings of traces leading to irrecoverable states. The second aspect is the generation of safety rules which do not appear in [Drummond, 1989]. The third aspect is that our method for computing critical states is more efficient than Drummond's.

Drummond and Bresina [Drummond and Bresina, 1990] use probabilistic information to guide the Drummond's previous planner such that paths are generated in the order of their likelyhood. This technique can certainly also be applied profitably to the planner described herein.

Other linear approaches to the synthesis of reactive plans include the Triangle Table approach [Fikes and Nilsson, 1972], Mitchel's planner [Mitchell, 1990], the Tree Triangle Tables [Nilsson, 1990] and the World-Automata approach [Kabanza, 1990]. These approaches do a linear search, do not generate safety rules, do not generate parallel reactions (i.e. do not detect conflicts among actions at planning time) and make no differentiation between safe and critical states.

Schoppers' Universal Plans [Schoppers, 1987] are reactive plans constructed by a backwards nonlinear planner which is not itself reactive. The planner uses a goal-reduction, assuming parallelism between goals, and detecting at every step conflicts among goals to validate or invalidate their parallelism. When conflicts are detected between goals, the planner generates "confinement rules" that order these goals. The traces we use to represent partial orders of actions differ from the Universal Plan approach and are constructed by a different technique. We construct a trace by a simple depth-first search; if there is a confusion between two actions in the current state, we "fork" another trace (i.e. a backtracking point) to detect if the confusion actually leads to a conflict. We do not need anything like "confinement-rule". While our approach simplifies the handling of conflicts among parallel goals, the Universal Plan representation on the other hand contains states which are only partial descriptions of the environment. This facilitates the direct generation of compact rules, i.e. rules whose antecedent is only a partial description of the environment.

References

[Brooks, 1986] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotic and Automation*, RA-2(1):14–23, March 1986.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Cohen and Feigenbaum, 1982] P. R. Cohen and E. A. Feigenbaum. *Handbook of Artificial Intelligence.* Pitman, London, 1982.

[Drummond and Bresina, 1990] M. Drummond and J. Bresina. Anytime synthetic projection: Maximazing probability of goal satisfaction. In *AAAI-90*, pages 138–144, Boston, August 1990.

[Drummond, 1989] M. Drummond. Situated control rules. In *Proc. of the first international conference* *on Principles of Knowledge Representation and Reasoning*, pages 103–113, Toronto Ontario Canada, May 1989. Morgan Kofmann.

[Georgeff and Lansky, 1987] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of AAAI-87*, pages 677–682, 1987.

[Godefroid and Wolper, 1991a] P. Godefroid and P. Wolper. A partial approach to model checking. To be presented at 6th IEEE Symposium on Logic in Computer Science, Amsterdam, July 1991.

[Godefroid and Wolper, 1991b] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. Technical Report, Université de Liège, January 1991.

[Godefroid, 1990] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. Computer-Aided Verification Workshop*, ACM/AMS DIMACS Series (extended abstract to appear in a volume of Lecture Notes in Computer Science, Springer-Verlag), Rutgers, U.S.A., June 1990.

[Kabanza, 1990] F. Kabanza. Synthesis of reactive plans for multi-path environments. In *AAAI-90*, pages 164–169, August 1990.

[Kaelbling, 1987] L. P. Kaelbling. An architecture for intelligent reactive systems. In M. P. Georgeff and A. Lansky, editors, *Reasonning about Actions and Plans, Proceedings of the 1986 Workshop, Timberline, Oregon*, pages 395–410. Morgan Kaufmann, June-July 1987.

[Mazurkiewicz, 1986] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, 1986.

[Mitchell, 1990] T. M. Mitchell. Becoming increasingly reactive. In *Proc. of AAAI-90*, pages 1051–1058, Boston, August 1990.

[Nilsson, 1990] N. J. Nilsson. Proposal for research on teleo-reactive agents. Draft, May 1990.

[Reisig, 1985] W. Reisig. *Petri Nets: an Introduction.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Schoppers, 1987] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the IJCAI*, pages 1039–1046, Milan, Italy, 1987.

[Wilkins, 1984] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.