A Hierarchical Task-Network Planner based on Symbolic Model Checking[*]

Ugur Kuter and **Dana Nau**
University of Maryland,
Department of Computer Science and
Institute for Systems Research,
College Park, Maryland 20742, USA

Marco Pistore
University of Trento,
Department of Information and
Communication Technology,
Via Sommarive, 14,
Povo di Trento, 38050, Italy

Paolo Traverso
ITC-IRST
Via Sommarive, 18,
Povo di Trento, 38055, Italy

Abstract

Although several approaches have been developed for planning in nondeterministic domains, solving large planning problems is still quite difficult. In this work, we present a novel algorithm, called YoYo, for planning in nondeterministic domains under the assumption of full observability. This algorithm enables us to combine the power of search-control strategies as in *Planning with Hierarchical Task Networks (HTNs)* with techniques from the *Planning via Symbolic Model-Checking (SMC)*. Our experimental evaluation confirms the potentialities of our approach, demonstrating that it combines the advantages of these paradigms.

Introduction

More and more research is addressing the problem of planning in nondeterministic domains. In spite of the recent promising results, the problem is still very hard to solve in practice, even under the simplifying assumption of full observability, i.e., the hypothesis that the state of the world can be completely observed at runtime. Indeed, in the case of nondeterministic domains, the planning algorithm must reason about all possible different execution paths to find a plan that works despite the nondeterminism, and the dimension of the generated conditional plan may grow exponentially.

Among others, planning based on *Symbolic Model Checking* (Cimatti *et al.* 2003; Rintanen 2002; Jensen & Veloso 2000; Cimatti, Roveri, & Traverso 1998) is one of the most promising approaches for planning under conditions of nondeterminism. This technique relies on the usage of propositional formulas for a compact representation of sets of states, and of transformations over such formulas for efficient exploration in the search space. The most common implementations of planning based on symbolic model checking have been realized with *Binary Decision Diagrams (BDDs)* (Bryant 1992), data structures that are well-suited to compactly represent propositional formulas and to efficiently compute their transformations. In different experimental settings, planning algorithms based on symbolic model checking and BDDs, e.g., those implemented in MBP (Bertoli *et al.* 2001a), have been shown to scale up to rather large-sized problems (Cimatti *et al.* 2003).

Another promising approach to planning with nondeterminism is forward-chaining planning with *Hierarchical Task Networks (HTNs)*, which was originally developed to provide efficient search-control heuristics for classical deterministic domains (Nau *et al.* 2003). (Kuter & Nau 2004) describes a way to generalize this approach to work in the nondeterministic case, along with a class of other forward-chaining planning techniques. The ND-SHOP2 planner, a nondeterminization of SHOP2 (Nau *et al.* 2003), is a forward-chaining HTN planner developed based on this technique. Like its predecessor, ND-SHOP2 has the ability to exploit expressive domain-specific search-control heuristics to guide its search for solutions. It has been demonstrated in (Kuter & Nau 2004) that ND-SHOP2 can be very effective in pruning the search space, and in some experiments ND-SHOP2 outperforms MBP. Unfortunately, ND-SHOP2 cannot efficiently solve problems where strategies cannot cut down the search space, since it does not have MBP's ability to work with symbolic representations of abstract collections of states.

In this paper, we have devised a formalism and developed a novel algorithm, called YoYo, that enables us to combine the power of the HTN-based search-control strategies with BDD-based symbolic model checking techniques. YoYo implements an HTN-based forward-chaining search as in ND-SHOP2, built on top of symbolic model-checking primitives based on BDDs as in MBP. This combination has required a complete rethinking of the ND-SHOP2 algorithm, in order to take advantage of situations where the BDD representation will allow it to avoid enumerating states explicitly.

We have performed an experimental comparison of YoYo with MBP and ND-SHOP2. The results confirm the advantage of combining search-control heuristics with symbolic model checking: YoYo's BDD representation enabled it to represent large problems compactly while exploiting HTN search-control strategies to prune large parts of the search space. YoYo easily outperformed both MBP and ND-SHOP2 in all cases, and it

---

[*]This work was performed when the first author was visiting University of Trento.

could deal with problem sizes that neither MBP nor ND-SHOP2 could scale up to.

The paper is organized as follows. We first explain more in detail the reasons why the integration of forward-chaining HTN planning with symbolic model-checking techniques should provide important advantages. We do this with the help of a well-known example, the Hunter-Prey domain (Koenig & Simmons 1995). Then, we present our formal setting, the YoYo planning algorithm, and its implementation using BDDs. Next, we present an experimental analysis of our approach and discuss our results. Finally, we conclude with our future research directions.

Motivations

We have identified two promising approaches for planning in nondeterministic domains; namely, planning with *Symbolic Model Checking (SMC)* and forward-chaining planning with *Hierarchical Task Networks (HTNs)*. These two planning techniques have complementary advantages: the former can exploit very efficient search-control heuristics for pruning the search space, and the latter uses propositional formulas for a compact representation of sets of states, and of transformations over such formulas for efficient exploration in the search space. Thus, it is reasonable to assume that the planning techniques developed using these two planning approaches perform well on different kinds of planning problems and domains. It is not hard to find planning domains that verifies this assumption: one example is the well-known Hunter-Prey domain, which was first introduced in (Koenig & Simmons 1995).

In the Hunter-Prey domain, there is a hunter and a prey in an $n \times n$ grid world. The task of the hunter is to catch the prey in the world. The hunter has five possible actions; namely, north, south, east, west, and catch. The prey has also five actions: it has the same four moves as the hunter, and an action to stay still in the world. The hunter can catch the prey only when the hunter and the prey are at the same location at the same time in the world. The nondeterminism for the hunter is introduced through the actions of the prey: at any time, it can take any of its actions, independent from the hunter's move.

We have experimented with two state-of-the-art planners designed to work in nondeterministic planning domains, namely ND-SHOP2 and MBP. ND-SHOP2 is a forward-chaining HTN-planning algorithm generated by the "nondeterminization" technique for (Kuter & Nau 2004), and MBP, on the other hand, is a planning system designed for exploiting representations and planning techniques based on symbolic model checking and BDDs (Bertoli *et al.* 2001a).

Figure 1 shows the average running times required by MBP and ND-SHOP2, as a function of increasing grid sizes. These results are obtained by running the two planners over 20 randomly-generated problems for each grid size, and then, by averaging the results.
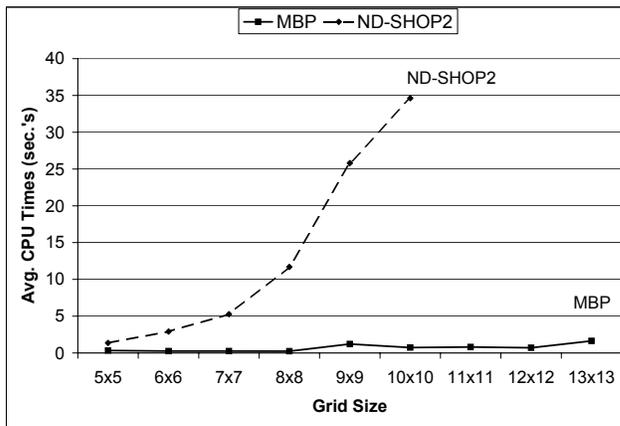

Figure 1: Average running times in sec.'s for MBP and ND-SHOP2 in the Hunter-Prey Domain as a function of the grid size, with one prey. ND-SHOP2 was not able to solve planning problems in grids larger than 10×10 due to memory-overflow problems.
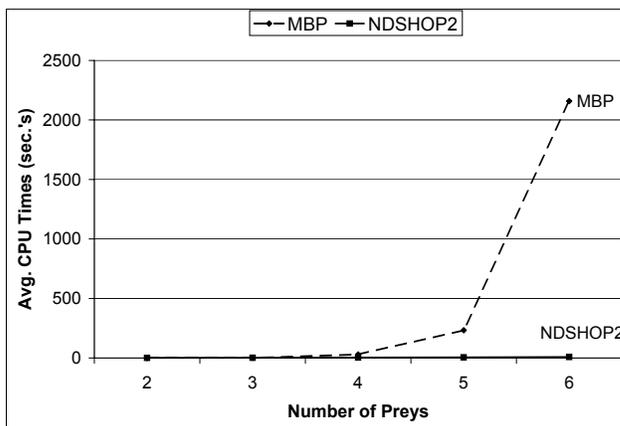

Figure 2: Average running times in sec.'s for MBP and ND-SHOP2 in the Hunter-Prey Domain as a function of the number of preys, with a fixed 3×3 grid.

ND-SHOP2 ran out of memory in the large problems of this domain because of the following: (1) the solutions for the problems in this domain are very large to store using an explicit representation, and (2) the search space does not admit a structure that can be exploited by search-control heuristics. Note that this domain allows only for high-level strategies for the hunter such as "look at the prey and move towards it," since the hunter does not know which actions the prey will take at a particular time. MBP, on the other hand, clearly outperforms ND-SHOP2 in these experiments, demonstrating the advantage of using BDD-based representations over explicit ones.

To test the effectiveness of the search-control heuristics, we have created a variation of the domain in which we may have more than one prey to catch. We made the movements of preys dependent on each other by

assuming that a prey cannot move to a location next to another prey in the world. Figure 2 shows the results in this adapted domain, with the 3×3 grid world: ND-SHOP2 is able to outperform MBP in this domain. The reason for the difference in these results compared to the previous ones is that this adapted domain allows much more powerful strategies for the hunter: e.g., "choose one prey and chase it while ignoring others; when you catch that prey, choose another and chase it, and continue in this way until all of the preys are caught." ND-SHOP2, using this strategy, is able to avoid the combinatorial explosion due the nondeterminism in the world. On the other hand, the BDD-based representations in MBP explode in size since the movements of the preys are dependent to each other, and MBP's backward-chaining breadth-first search techniques apparently cannot compansate for such an explosion.

The two experiments above clearly show that planning using HTN-based search-control heuristics and BDD-based compact representations have complementary advantages, and they demonstrate the improvements in the efficiency of planning that can be achieved when these two techniques are combined in a single planning framework. In this paper, we present one such framework that is built on forward-chaining HTN planning over symbolic model-checking primitives, and a planning algorithm that works in that framework.

Background

We use the usual definitions for nondeterministic planning domains and planning problems in such domains as in (Cimatti *et al.* 2003). A *nondeterministic planning domain* is a tuple of the form $(\mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{P}$ is a finite set of propositions, $\mathcal{S} \subseteq 2^{\mathcal{P}}$ is the set of all possible states, $\mathcal{A}$ is the finite set of all possible actions, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the state-transition relation. The set of successor states generated when an action a is applied in s is $\gamma(s, a) = \{s' \mid (s, a, s') \in \mathcal{R}\}$; we say a is not applicable in s, if $\gamma(s, a) = \emptyset$. The set of states in which an action $a \in \mathcal{A}$ can be applied is $S_a \subseteq \mathcal{S}$.

For instance, consider the Hunter-Prey domain described in the previous section. In this domain, a state $s \in \mathcal{S}$ describe the possible positions of the hunter and the prey on the grid. The actions in $\mathcal{A}$ describe the possible moves of the hunter as well as the act of catching a prey. The actions of the prey are modeled through the effects of the hunter's actions; i.e., they are described in the transtion relation $\mathcal{R}$.

We define a *policy* to be a set $\pi = \{(s, a) \mid s \in \mathcal{S} \text{ and } a \in A(s)\}$, where $A(s) \subseteq \mathcal{A}$ is the set of actions that are applicable in s. The set of states in a policy is $S_\pi = \{s \mid (s, a) \in \pi\}$. An *execution structure* induced by the policy π is a directed graph $\Sigma_\pi = (V_\pi, E_\pi)$: V_π is the set of the nodes of Σ_π, which represent the states that can be generated by executing actions in π. E_π is the set of arcs between the nodes of Σ_π, which represent possible state transitions caused by the actions in π.

A *planning problem* in a nondeterministic planning domain $D = (\mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R})$ is a tuple of the form $P = (D, I, G)$, where $I \subseteq \mathcal{S}$ is a set of initial states, and $G \subseteq \mathcal{S}$ is a set of goal states. In this paper, we focused on only strong and strong-cyclic solutions for planning problems. We summarize their definitions here; for a detailed discussion, see (Cimatti *et al.* 2003):

- A *strong solution* is a policy that is guaranteed to achieve the goals of the problem, despite the nondeterminism in the domain. That is, a policy π is a strong solution if (1) every finite path in the execution structure Σ_π reaches to a final node that satisfies the goals, and (2) there are no infinite paths in Σ_π — i.e., Σ_π is acyclic.

- A *strong-cyclic solution* is a policy that is guaranteed to reach the goals under a "fairness assumption;" i.e., the assumption that the execution of a strong-cyclic solution will eventually exit the loops. More specifically, in a strong-cyclic solution π, every partial path in the execution structure Σ_π can be extended to a finite execution path that reaches to a goal.

We use the usual definitions for primitive tasks, non-primitive tasks, task networks, and methods as in (Nau *et al.* 2003), except that we restrict ourselves to ground instances of these constructs in this paper. We assume the existence of a finite set of symbols that denote the *tasks* to be performed in a planning domain $D = (\mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R})$. Every action in $\mathcal{A}$ is a task symbol, and there are some additional task symbols called *non-primitive tasks*. A *task network* is a partially-ordered set of tasks.

In this paper, we adopt the notion of *ordered task decomposition* (Nau *et al.* 2003); that is, the tasks in a task network are decomposed into subtasks in the order they are supposed to be performed in the world. A *method* describes a possible way of decomposing the tasks in a task network into smaller and smaller tasks. More formally, a method is an expression of the form $m = (t \; C \; w)$ such that t is a nonprimitive task, C is a conjunction of literals, and w is a task network that denotes the subtasks generated by decomposing t by m. The set of states in which m can be applied is $S_m = \{s \mid s \in \mathcal{S} \text{ and } C \text{ holds in } s\}$.

Methods describe the search-control strategies to be used in a domain. As an example, suppose we have a task chase_prey in the Hunter-Prey domain. A method for this task can be defined as follows:

```
(:method (:task (chase_prey))
  (:conditions (prey_not_caught)
               (prey_to_the_north hunter))
  (:subtasks (move_north hunter)(chase_prey)))
```

This method is applicable only in the states in which the prey has not been caught yet, and it is currently at a location to the north of the hunter in the world. It specifies the following search-control strategy: if the world is in any of the states in which the method is applicable, then the hunter should move north first and

```
Procedure YoYo(D, I, G, w, M)
  return YoyoAux(D, {(I, w)}, G, M, ∅, {(I, w)})

Procedure YoyoAux(D, X, G, M, π, X_0)
  X ← PruneSituations(X, G, π)
  if there is a situation (S, w = nil) ∈ X such that S ⊄̸ G
    then return(failure)
  if NoGoodPolicy(π, X, G, X_0) then return(failure)
  if X = ∅ then return(π)
  select a situation (S, w) from X and remove it
  F ← ComputeDecomposition(S, w, D, M)
  if F = ∅ then return(failure)
  X' ← ComputeSuccessors(F, X)
  π' ← π ∪ {(s, a) | (S', a, w') ∈ F and s ∈ S'}
  π ← YoyoAux(D, X', G, M, π', X_0)
  if π = failure then return(failure)
  return(π)
```

Figure 3: YoYo, an HTN planning algorithm for generating solutions in nondeterministic domains. In the YoyoAux procedure above, X is the current set of situations and X_0 is the initial set of situations; i.e., $X_0 = \{(I, w)\}$.

```
Procedure PruneSituations(X, G, π)
  X' ← ∅
  for every situation (S, w) ∈ X
    S' ← S \ (G ∪ S_π)
    if S' ≠ ∅ then X' ← X' ∪ {(S', w)}
  return X'
```

Figure 4: The PruneSituations procedure.

continue chasing prey from there.

Let s be a state, w be a task network, and t be a task that has no predecessors in w – i.e., t can be decomposed into smaller tasks by the semantics of ordered task decomposition since there is no task before t that is to be achieved. Then we have two cases:

- t *is a primitive task.* Then, t can be directly executed in s — i.e., t corresponds to an action in $\mathcal{A}$—, if $s \in S_t$. The result of that application is the set of states $\gamma(s, t)$ and the successor task network $w \setminus \{t\}$.

- t *is a nonprimitive task.* Let m be a method for t – i.e., $m = (t, C, w')$. Then, m can be used to decompose t in s if $s \in S_m$. The result of that decomposition is the task network $(w \setminus \{t\}) \cup w'$ such that every partial-ordering constraint in both $w \setminus \{t\}$ and w' is satisfied in $(w \setminus \{t\}) \cup w'$.

The YoYo Planning Algorithm

In this section, we describe YoYo, a forward-chaining HTN planning algorithm, which is designed to combine the ability of exploiting search-control heuristics as in HTN planning with symbolic model-checking techniques in a single planning framework. Figure 3 shows the YoYo planning procedure for finding solutions for planning problems in nondeterministic domains.

The input for the planning procedure YoYo consists of planning problem (D, I, G) in a nondeterministic planning domain $D = (\mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R})$, an initial task network w, and a set of HTN methods M for the domain D. The algorithm exploits tuples of the form (S, w), called *situations*, which are resolved by accomplishing the task network w in the states of S.

Starting with the initial situation (I, w), YoYo recursively generates successive sets of situations until a solution for the given planning problem is generated. At each iteration of the planning process, YoYo first checks the set X of current situations for cycles and goal states by using the PruneSituations procedure shown in Figure 4. For every situation $(S, w) \in X$, PruneSituations checks the set of states S and removes any state that either appears already in the policy (and therefore, an action has already been planned for it), or appears in the set of goal states G (and therefore, no action should be planned for it). As a result, the set of situations returned by the PruneSituations procedure are truly the situations that needs to be explored and progressed into successor ones in the search. We call such situations as the *open situations* of the current search trace.

After generating the open situations to be explored, YoYo checks if there is an open situation (S, w) such that there are no more tasks to be performed in w, but the goal has not been reached yet (i.e., $S \not\subseteq G$). In this case, we have a failure in the search, and therefore, YoYo returns failure from the current search trace. Otherwise, YoYo uses the routine NoGoodPolicy to further check if the current partial policy conforms to the requirements of the kinds of solutions it is looking for. The formal definition of this routine depends on whether we are looking for strong or strong-cyclic solutions, so we leave the discussion on this routine to the next section.

If the current partial policy π does not meet with the requirements of being a solution to the input problem, then YoYo returns from the current search trace by failure. Otherwise, π is a solution to the underlying planning problem if there are no open situations to be explored further. This is true since π does not violate the requirements of the input problem, as it passed the NoGoodPolicy in the previous step.

Suppose there are open situations to explore for the planner. Then YoYo selects one of them, say (S, w), and attempts to generate an action for every state in S. The ComputeDecomposition routine, which is basically an HTN-planning engine, is responsible for this operation, as follows. In a situation (S, w), let t be a task that has no predecessors in w. If t is a primitive task then t can be executed directly in the world. Let a be an action that corresponds to t, and a can be applied in each state in S; i.e., we have $S \subseteq S_a$. Note that applying an action a in a set of states S does not generate any new open situations: we require that S must be a subset of S_a because, otherwise, there is at least one state in S for which no action is applicable, and this is a failure point in planning.

```
Procedure ComputeDecomposition(S, w, D, M)
  F ← ∅; X ← {(S, w)}
  loop
    if X = ∅ then return(F)
    select a tuple (S, w) ∈ X and remove it
    select a task t that has no predecessors in w
    if t is a primitive task then
      actions ← {a | a ∈ 𝒜 is an action for t, and S ⊆ Sₐ}
      if actions = ∅ then return ∅
      select an action a from actions
      F ← F ∪ {(S, a, w \ {t})}
    else
      methods ← {m | m is a method in M for t
                       and S ∩ Sₘ ≠ ∅}
      if methods = ∅ then return ∅
      select a method instance m from methods
      X ← X ∪ {(S ∩ Sₘ, (w \ {t}) ∪ w′}
      if S \ Sₘ ≠ ∅ then X ← X ∪ {(S \ Sₘ, w)}
```

Figure 5: The ComputeDecomposition procedure.

```
Procedure ComputeSuccessors(F, X)
  X′ ← X ∪ {(succ(S, a), w) | (S, a, w) ∈ F}
  X′ ← {(Compose(w, X′), w) | (S, w) ∈ X′}
  return X′
```

Figure 6: The ComputeSuccessors procedure.

If t is not primitive, then we successively apply methods to the nonprimitive tasks in w until an action is generated. Suppose we chose to apply a method m to t. This generates two possible situations: (1) the situation that arises from decomposing t by m in the states $S \cap S_m$ in which m is applicable, and (2) the situation that specifies the states in which m is not applicable – i.e., the situation $(S \setminus S_m, w)$. In the former case, we proceed with decomposing the subtasks of t as specified in m. In the latter case, on the other hand, other methods for t must be used. Note that if there are no other methods for t to be used in situations like $(S \setminus S_m, w)$, then ComposeDecomposition returns the empty set, forcing YoYo to correctly report a failure.

The ComputeDecomposition returns a set F of the form $\{(S_i, a_i, w_i)\}_{i=0}^{k}$. If $F = \emptyset$ then this means that the decomposition process has failed since there is a state $s \in S$ such that we cannot generate an action for s by using the methods provided for the underlying planning domain. If $F \neq \emptyset$ then the routine has generated an action a_i for each state in S – i.e., we have $S = \bigcup_i S_i$ —, and a task network w_i to be accomplished after applying that action.

Suppose ComputeDecomposition returned a non-empty set F of tuples of the form (S', a, w'). Then, YoYo proceeds with computing the successor situations to be explored using the ComputeSuccessors routine as follows: for each tuple $(S', a, w') \in F$, it first generates the set of states that arises from applying a in S' by using the function

$$succ(S', a) = \{s'' \mid s' \in S' \text{ and } (s', a, s'') \in \mathcal{R}\},$$

where $\mathcal{R}$ is the state-transition relation for the underlying planning domain. The next situation corresponding this action application is defined as $(succ(S', a), w')$.

Once YoYo generates the all of the next situations be explored, it composes the newly-generated situations with respect to the task networks they specify to be accomplished. More formally, the Compose function of Figure 6 is defined as follows:

$$Compose(w, X) = \{s \mid (S, w) \in X \text{ and } s \in S\}.$$

The composition of a set of situations is an optimization step in the planning process. The progression of open situations may create a set of situations in which more than one situation may specify the same task network. Composing such situations is not required for correctness, but it has the advantage of planning with more compact BDD-based representations.

Strong and Strong-Cyclic Planning using YoYo

The abstract planning procedure YoYo can be used for strong and strong-cyclic planning by using slightly different routines for NoGoodPolicy, which specifies the different conditions required for a policy to be a strong or a strong-cyclic solution for a planning problem. In this section, we discuss the definitions for these routines.

Strong Planning. In strong planning, a policy must induce an execution trace to a goal state from every state that is reachable from the initial states and there should be no cycles in the execution structure induced by that policy. This condition can be checked as follows:

```
Procedure NoGoodPolicy_Strong(π, X, G, X₀)
  S′ ← ∅; S₀ ← StatesOf(X₀); S ← G ∪ StatesOf(X)
  while S′ ≠ S
    S′ ← S
    S ← S ∪ {s′ | (s′, a) ∈ π, and γ(s′, a) ⊆ S}
    π ← π \ {(s, a) | s ∈ S and (s, a) ∈ π}
  if S₀ ⊆ S and π = ∅ then return FALSE
  return TRUE
```

The above routine is built on the strong backward-preimage function of (Cimatti et al. 2003). Starting from the set of states in the open situations and the goal states, it computes the set of states in the policy from which an open state or a goal state is reachable. While doing so, it removes the state-action pairs for those states computed by the strong backward-preimage. At the end of this process, if there is a state-action pair left in the policy, then it means that the policy induces a cycle in the execution structure, and therefore, it can not be a strong solution for a planning problem.

NoGoodPolicy_Strong uses a subroutine called StatesOf, which returns the set of all the states that appear in a given set of situations. More formally,

$$StatesOf(X) = \{s \mid (S, w) \in X \text{ and } s \in S\}$$

Strong-Cyclic Planning. The definition for the No-GoodPolicy check for strong-cyclic planning differs from the strong case only in the way that the backward image is computed. In particular, in the strong-cylic case, we use the weak backward-preimage, instead of the strong one. This way, we can detect only those cycles induced by the input policy that violate the "fairness assumption" of strong-cyclic planning as described before.

The NoGoodPolicy procedure for the strong-cyclic planning is defined as follows:

```
Procedure NoGoodPolicy_StrongCyclic(π, X, G, X₀)
  S' ← ∅; S₀ ← StatesOf(X₀); S ← G ∪ StatesOf(X)
  while S' ≠ S
    S' ← S
    S ← S ∪ {s' | (s', a) ∈ π, and S ∩ γ(s', a) ≠ ∅}
    π ← π \ {(s, a) | s ∈ S and (s, a) ∈ π}
  if S₀ ⊆ S and π = ∅ then return FALSE
  return TRUE
```

Discussion. Note that we are using these procedures to verify the generated policies meet the requirements to be a solution for the underlying planning problems; we are not using them for generating the solution policies themselves as in (Cimatti *et al.* 2003) since that generation is performed by the forward-chaining HTN-based search engine in YoYo.

BDD-based Implementation of Our Algorithms

We have implemented a prototype of the YoYo planning algorithm, described in the previous section. Our current implementation is built on both the ND-SHOP2 and the MBP planning systems. It extends the ND-SHOP2 planning system for (1) planning over sets of states rather than a single state, and (2) implementing the NoGoodPolicy routine as a part of its backtracking search. It uses an interface to MBP for exploiting the machinery of BDDs implemented in it.

In this section, we present a framework that enables us to implement the data structures of the YoYo algorithm and its helper routines using BDD-based symbolic model-checking primitives. In this framework, we use the same machinery to represent the states of a planning domain as in (Cimatti *et al.* 2003). This machinery is based on using propositional formulae to compactly represent sets of states and possible transitions between those states in a planning domain.

We assume a vector $\overline{s}$ of propositions that represents the current state of the world. For example, in the Hunter-Prey world with a 3×3 grid and one prey, $\overline{s}$ is $\{hx = 0, \ldots, hx = 3, hy = 0, \ldots, hy = 3, px = 0, \ldots, px = 3, py = 0, \ldots, py = 3, prey\_caught\}$. A *state* is an assignment of the truth-values {TRUE, FALSE} to each proposition in $\overline{s}$. We denote such an assignment by $s(\overline{s})$.

Based on this formulation, a set of states S corre-

sponds to the formula $S(\overline{s})$ such that

$$S(\overline{s}) = \bigvee_{s \in S} s(\overline{s}).$$

This definition of set of states is the basis of our framework in this paper. It allows us to define YoYo's forward search mechanism over BDD-based representations of sets of states, rather than single states.

We also assume another vector $\overline{s}'$ of propositional variables to represent the next states of the world, respectively. Similarly, we use a vector $\overline{a}$ of action variables, which allows representing a set of actions at the same time. A policy π, which is a set of state-action pairs, can be represented as a formula $\pi(\overline{s}, \overline{a})$ in the variables $\overline{s}$ and $\overline{a}$. We denote a set of states S with a formula $S(\overline{s})$ in the state vector $\overline{s}$ as before. We represent a situation as a pair of the form $(S(\overline{s}), w)$, where w is a task network, as described in the previous section.

The initial situation can be represented by $X_0 = \{(I(\overline{s}), w)\}$, where $I(\overline{s})$ represents the initial set of states and w is the initial task network. Similarly, we represent the set of goal states with the formula $G(\overline{s})$. We assume the existence of a state-transition relation R, which can be represented as $R(\overline{s}, \overline{a}, \overline{s}')$, where $\overline{s}$ denotes the current state vector, $\overline{a}$ denotes the current action vector, and $\overline{s}'$ denotes the next state vector.

The formulations of the inequality of sets, set difference operations, and subset relations constitute the most basic primitives used in conditionals and termination conditions of the loops of our algorithms. These operations can be easily encoded in terms of basic logical operations on the formulas described above.

The result of applying an action a in a set of states S can be represented as the formula:

$$\exists \overline{s}' : S(\overline{s}) \wedge R(\overline{s}, \overline{a}, \overline{s}')[\overline{s}'/\overline{s}],$$

where $[\overline{s}'/\overline{s}]$ is called the *forward-shifting* operation. Note that the above formula represents the $succ(S, a)$ function described in the previous section.

The StatesOf primitive we use for computing the set of all states described by a set of situations can be represented as a set-union operator over the situations we are interested in. In particular, if we want to compute the set of all states of $X = \{x_1, x_2, \ldots, x_n\}$, then this operations corresponds to the formula $S_1(\overline{s}) \vee S_2(\overline{s}) \vee \ldots \vee S_n(\overline{s})$, where we have $x_i = (S_i, w_i)$.

We are now ready to give the formulations for our algorithms. The PruneSituations procedure is built on set-difference and set-union operations, which can be represented as follows: $S(\overline{s}) \wedge \neg(G(\overline{s}) \vee \exists \overline{a} : \pi(\overline{s}, \overline{a}))$.

The NoGoodPolicy procedures for strong and strong-cyclic planning are based on two primitives for computing weak and strong preimages of a particular set of states. These preimage computations correspond to

$$\exists \overline{a} \exists \overline{s}'. \pi(\overline{s}, \overline{a}) \wedge S(\overline{s}') \quad \text{and} \quad \exists \overline{a} \exists \overline{s}'. \pi(\overline{s}, \overline{a}) \implies S(\overline{s}'),$$

respectively.

In the ComputeDecomposition routine, we check whether a method or an action is applicable in a given

set S of states or not. This check corresponds to the following formulas: $S(\overline{s}) \implies S_a(\overline{s})$ and $S(\overline{s}) \wedge S_m(\overline{s})$, where $S(\overline{s})$ represents the set of states in which we are performing these checks, and $S_a(\overline{s})$ and $S_m(\overline{s})$ represents the set of all states in which the action a and the method m is applicable.

Finally, we can represent the update of a policy π by a set of state-action pairs π' as follows: $\pi(\overline{s}, \overline{a}) \vee \pi'(\overline{s}, \overline{a})$.

Experimental Evaluation

We have designed three sets of experiments in the Hunter-Prey Domain, described earlier. For our experiments, we assumed that the domain is fully-observable in the sense that the hunter can always observe the location of the prey. We also assumed that the hunter moves first in the world, and the prey moves afterwards. The nondeterminism for the hunter is introduced through the movements of the prey; the prey may take any of its five actions, independent from the hunter's move.

In our experiments, we have investigated the performances of YoYo, ND-SHOP2, and MBP. For all our experiments, we used a HP Pavilion N5415 Laptop with 256MB memory, running Linux Fedora Core 2. We set the time limit for the planners as 40 minutes. In our experiments, each time ND-SHOP2 and MBP had a memory overflow or they could not solve a problem within out time limit, we ran them again on another problem of the same size. We omitted each data point on which this happened more than five failures, but included the data points where it happened 1 to 4 times.

Experimental Set 1. In these experiments, we aimed to investigate how well YoYo is able to cope with large-sized problems compared to ND-SHOP2 and MBP. To achieve this objective, we designed experiments on hunter-prey problems with increasing grid sizes. For these problems, we assumed there is only one prey in the world in order to keep the amount of nondeterminism for the hunter at a minimum.

Figure 7 shows the results of the experiments for grid sizes $n = 5, 6, \dots, 10$. For each value for n, we have randomly generated 20 problems and run MBP, ND-SHOP2, and YoYo on those problems. In this figure, we report the average running times required by the planners on those problems.

For grids larger than $n = 10$, ND-SHOP2 was not able to solve the planning problems due to memory overflows. This is because the sizes of the solutions in this domain are very large, and therefore, ND-SHOP2 runs out of memory as it tries to store them explicitly. Note that this domin admits only high-level search strategies such as "look at the prey and move towards it." Although this strategy helps the planner prune a portion of the search space, such pruning alone does not compansate for the explosion in the size of the explicit representations of the solutions for the problems.

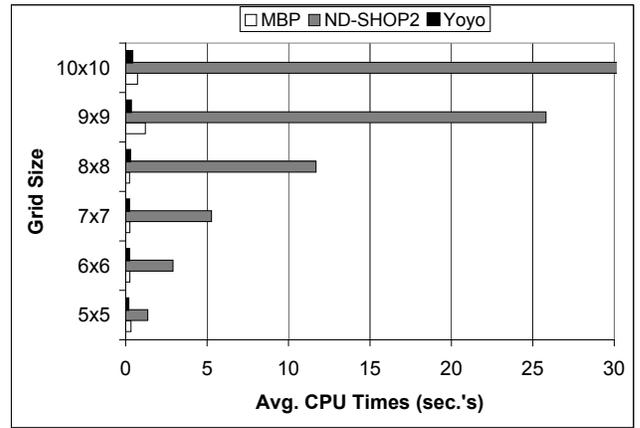On the other hand, both YoYo and MBP was able to solve all of the problems in these experiments.

Figure 7: Average running times (in sec.'s) of YoYo, ND-SHOP2, and MBP in the hunter-prey domain as a function of the grid size, with one prey.
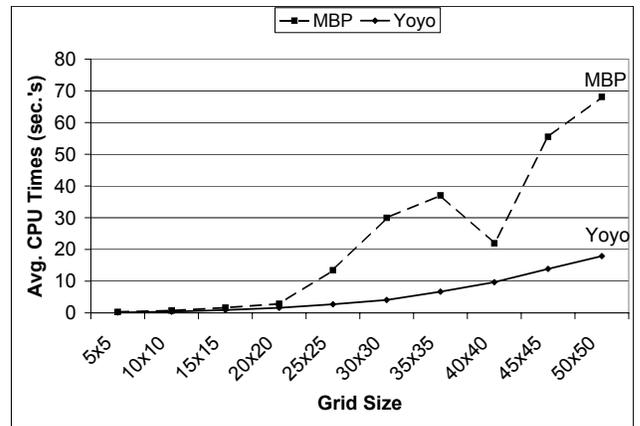

Figure 8: Average running times in sec.'s for YoYo and MBP on some larger problems in the hunter-prey domain as a function of the grid size, with one prey.

The difference between the performances of YoYo and ND-SHOP2 demonstrates the impact of the use of BDD-based representations: YoYo, using the same HTN-based heuristic as ND-SHOP2, was able to scale up as good as MBP since it is able to exploit BDD-based representations of the problems and their solutions.

In order to see how YoYo performs in larger problems compared to MBP, we have also experimented with YoYo and MBP in much larger grids. Figure 8 shows the results of these experiments in which, using the same setup as above, we varied the size of the grids in the planning problems as $n = 5, 10, 15, \dots, 45, 50$.

These results show that YoYo is able to perform better than MBP with the increasing grid size. The running times required by both of the planners increase in larger grids; however, this increase is much slower for YoYo than MBP as shown in Figure 8 due to the following reasons: (1) YoYo is able to combine the advantages of exploiting HTN-based search-control heuristics
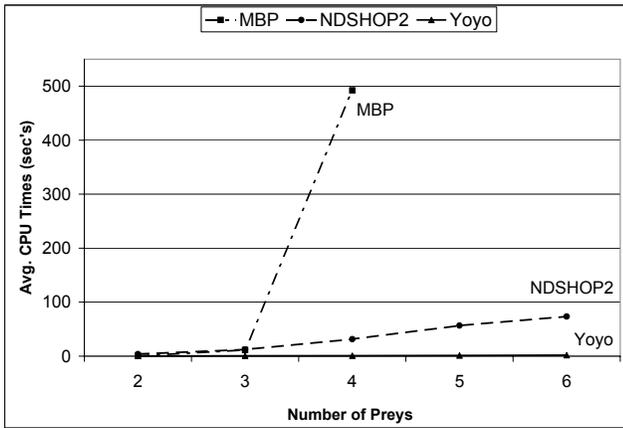
Figure 9: Average running times in sec.'s of ND-SHOP2, YoYo and MBP on problems in the Hunter-Prey domain as a function of the number of preys, with a 4×4 grid. MBP was not able to solve planning problems with 5 and 6 preys within 40 minutes.

with the advantages of using BDD-based representations, whereas MBP cannot exploit HTN-based strategies to complement its BDD-based planning techniques; and (2) YoYo, being a forward planner, considers only those states that are reachable from the initial states of the planning problems, whereas MBP's backward-chaining algorithms explore states that are not reachable from the initial states of the problems at all.

Experimental Set 2. In order to investigate the effect of combining search-control strategies and BDD-based representations in YoYo, we used the following variation of the Hunter-Prey domain. We assumed that we have more than one prey in the world, and the prey i cannot move to any location within the neighbourhood of prey $i + 1$ in the world. In such a setting, the amount of nondeterminism for the hunter after each of its move increases combinatorially with the number of preys in the domain. Furthermore, the BDD-based representations of the underlying planning domain explode in size under these assumptions, mainly because the movements of the preys are dependent to each other.

In this adapted domain, we used a search-control strategy in ND-SHOP2 and YoYo that tells the planners to chase the first prey until it is caught, then the second prey, and so on, until all of the preys are caught. Note that this heuristic allows for abstracting away from the huge state space: when the hunter is chasing a prey, it does not need to know the locations of the other preys in the world, and therefore, it does not need to reason and store information about those locations.

In the experiments, we varied the number of preys from $p = 2, \ldots, 6$ in a 4×4 grid world. We have randomly generated 20 problems for each experiment with different number of preys. Figure 9 shows the results of these experiments with MBP, ND-SHOP2,

and YoYo. These results demonstrate the power of combining HTN-based search-control heuristics with BDD-based representations of states and solutions in our planning problems: YoYo was able to outperform both ND-SHOP2 and MBP. The running times required by MBP grow exponentially faster than those required by YoYo with the increasing size of the preys, since MBP cannot exploit HTN-based heuristics. Note that ND-SHOP2 performs much better than MBP in the presence of good search-control heuristics.

Experimental Set 3. In order to further investigate YoYo's performance compared to that of ND-SHOP2 and MBP, we have also performed an extended set of experiments with multiple preys and with increasing grid sizes. We varied the number of preys as $p = 2, \ldots, 6$ and the grid sizes $n = 3, 4, 5, 6$. As before, we have randomly generated 20 problems for each experiment with different p and n combinations.

Table 1 reports the average running times required by YoYo, MBP, and ND-SHOP2 in these experiments. These results provide further proof for our conclusions. Search-control heuristics help both YoYo and ND-SHOP2 as they both outperform MBP with the increasing number of the preys. However, with increasing grid sizes, ND-SHOP2 runs into memory problems as before due to its explicit representations of states and solutions of the problems. YoYo, on the other hand, was able to cope with very well both with increasing the grid sizes and the number of preys in these problems.

Discussion on the Results. Our exprimental results demonstrate the importance of using HTN-based search-control heuristics and BDD-based representations in a single forward-chaining framework. The search-control heuristics exploit the structure of the underlying planning problems, and therefore, they result in a more compact and structured BDD representations of the planning problems and domains. For example, in the hunter-prey domain, the strategy, which tells YoYo to focus on catching one prey while ignoring other preys, provides a combinatorial reduction in the representations of the solutions for the problems and the state-transition relation for the domain. BDDs provide even further compactness in those reduced representations. Note that the same strategy did not work for ND-SHOP2 very well in large problems due to explicit representations of the problems and the domain. Note also that BDD-based representations alone did not work very well for MBP in problems with increasing number of the preys, since those representations are not sufficient to abstract away from the irrelevant portions of the state space. YoYo, on the other hand, was able to cope very well with problems with both characteristics.

Related Work

Over the years, several planning techniques have been developed for planning in nondeterministic do-

Table 1: Average running times of MBP, ND-SHOP2, and YoYo on Hunter-Prey problems with increasing number of preys and increasing grid size.

Grid	2 preys		
Grid	MBP	ND-SHOP2	YoYo
3x3	0.343	0.78	0.142
4x4	0.388	3.847	0.278
5x5	1.387	18.682	0.441
6x6	3.172	76.306	0.551
	3 preys		
Grid	MBP	ND-SHOP2	YoYo
3x3	1.1	1.72	0.329
4x4	11.534	12.302	0.521
5x5	133.185	58.75	0.92
6x6	368.166	250.315	1.404
	4 preys		
Grid	MBP	ND-SHOP2	YoYo
3x3	29.554	3.256	0.448
4x4	492.334	31.591	0.759
5x5	>40 mins	176.49	1.818
6x6	>40 mins	547.911	3.295
	5 preys		
Grid	MBP	ND-SHOP2	YoYo
3x3	233.028	5.483	0.655
4x4	>40 mins	56.714	1.275
5x5	>40 mins	304.03	3.028
6x6	>40 mins	memory-overflow	7.059
	6 preys		
Grid	MBP	ND-SHOP2	YoYo
3x3	2158.339	8.346	0.781
4x4	>40 mins	73.435	1.786
5x5	>40 mins	486.112	5.221
6x6	>40 mins	memory-overflow	11.826

mains. Examples include satisfiability and planning-graph based techniques, symbolic model-checking approaches, and forward-chaining heuristic search.

The planning-graph based techniques can address conformant planning, where the planner has nondeterministic actions and no observability, and a limited form of partial-observability (Smith & Weld 1998; Weld, Anderson, & Smith 1998; Brafman & Hoffmann 2004). To the best of our knowledge, examples for the nondeterministic satisfiability-based planners include (Castellini, Giunchiglia, & Tacchella 2003; Ferraris & Giunchiglia 2000) on conformant planning, and (Rintanen 1999) on conditional planning.

The idea of using symbolic model-checking (SMC) to do planning in nondeterministic domains was first introduced in (Cimatti *et al.* 1997; Giunchiglia & Traverso 1999; Cimatti, Roveri, & Traverso 1998). (Cimatti *et al.* 2003) gives a full formal account and an extensive experimental evaluation of planning for these three kinds of solutions. Other approaches include (Jensen & Veloso 2000; Jensen, Veloso, & Bowling 2001; Rintanen 2002; Jensen, Veloso, & Bryant 2003). SMC-

planning has been extended to deal with partial observability (Bertoli *et al.* 2001b) and extended goals (Pistore & Traverso 2001; Dal Lago, Pistore, & Traverso 2002). The MBP planning system (Bertoli *et al.* 2001a) is capable of handling both.

Planning based on *Markov Decision Processes (MDPs)* (Boutilier, Dean, & Hanks 1999) also has actions with more than one possible outcome, but models the possible outcomes using probabilities and utility functions, and formulates the planning problem as an optimization problem. For problems that can be solved either by MDPs or by model-checking-based planners, the latter have been empirically shown to be more efficient (Bonet & Geffner 2001).

(Kuter & Nau 2004) presents a generalization technique to transport the efficiency improvements that has been achieved for forward-chaining planning in deterministic domains over to nondeterministic case. Under certain conditions, they showed that a "nondeterminized" algorithm's time complexity is polynomially bounded by the time complexity of the deterministic version. ND-SHOP2 is an HTN planner developed using this technique for SHOP2 (Nau *et al.* 2003). YoYo, our HTN planner we described in this work, is built on both the ND-SHOP2 and the MBP planning systems.

In YoYo, we only focused on HTN-based heuristics as in ND-SHOP2 and combining them with BDD-based representations as in MBP. However, it is also possible to develop variants of YoYo, designed to work with other search-control techniques developed for forward planning, such as temporal-logic based ones as in (Bacchus & Kabanza 2000; Kvarnström & Doherty 2001). In our future work, we intend to investigate such techniques in YoYo along with the HTN-based ones we developed in this work, and compare the advantages/disadvantages of using different search-control mechanisms.

Conclusions

This paper describes a new algorithm for planning in fully observable nondeterministic domains. This algorithm enables us to combine the search-control ability of HTN planning with the state-abstraction ability of BDD-based symbolic model-checking. Our experimental evaluation shows that the combination is a potent one: it has large advantages in speed, memory usage, and scalability.

In the future, we plan to extend the comparison to other domains, to further confirm our hypothesis on the benefits of the proposed approach. We plan also to devise algorithms that further integrate symbolic model checking and HTNs, by combining HTN-based forward search with MBP's backward-search algorithms that are based on symbolic model-checking.

Acknowledgments

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001a. MBP: a model based planner. In *Proceeding of IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information*, 93–97.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001b. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 473–478. Seattle, USA: Morgan Kaufmann.

Bonet, B., and Geffner, H. 2001. GPT: a tool for planning with uncertainty and partial information. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 82–87.

Boutilier, C.; Dean, T. L.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Brafman, R., and Hoffmann, J. 2004. Conformant Planning via Heuristic Forward Search: A New Approach. In *"Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)"*. Whistler, Canada: Morgan Kaufmann.

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2003. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence* 147(1–2):85–117.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for AR. In *Proceedings of the European Conference on Planning (ECP)*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNAI)*, 130–142. Toulouse, France: Springer-Verlag.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Strong planning in non-deterministic domains via model checking. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*, 36–43. AAAI Press.

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals.

In *AAAI/IAAI Proceedings*, 447–454. Edmonton, Canada: AAAI Press/The MIT Press.

Ferraris, P., and Giunchiglia, E. 2000. Planning as satisfiability in nondeterministic domains. In *AAAI/IAAI Proceedings*, 748–753. AAAI Press.

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In *Proceedings of the European Conference on Planning (ECP)*, 1–20.

Jensen, R., and Veloso, M. M. 2000. OBDD-based universal planning for synchronized agents in nondeterministic domains. *Journal of Artificial Intelligence Research* 13:189–226.

Jensen, R.; Veloso, M. M.; and Bowling, M. H. 2001. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the European Conference on Planning (ECP)*.

Jensen, R.; Veloso, M. M.; and Bryant, R. 2003. Guided symbolic universal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Trento: AAAI Press.

Koenig, S., and Simmons, R. G. 1995. Real-time search in non-deterministic domains. In *IJCAI*, 1660–1669.

Kuter, U., and Nau, D. 2004. Forward-chaining planning in nondeterministic domains. *Proceedings of the National Conference on Artificial Intelligence (AAAI)* 513–518.

Kvarnström, J., and Doherty, P. 2001. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Articial Intelligence* 30:119–169.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 479–484. Seattle, USA: Morgan Kaufmann.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research* 10:323–352.

Rintanen, J. 2002. Backward plan construction for planning as search in belief space. In *Proceedings of the International Conference on AI Planning Systems (AIPS)*.

Smith, D. E., and Weld, D. S. 1998. Conformant Graphplan. In *AAAI/IAAI Proceedings*, 889–896.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending Graphplan to handle uncertainty and sensing actions. In *AAAI/IAAI Proceedings*, 897–904. Menlo Park: AAAI Press.