Interleaving Execution and Planning via Symbolic Model Checking

Piergiorgio Bertoli, Alessandro Cimatti, Paolo Traverso
[bertoli,cimatti,traverso]@irst.itc.it

Abstract

Interleaving planning and execution is the practical alternative to the problem of planning off-line with large state spaces. While planning via symbolic model checking has been extensively studied for off-line planning, no framework for interleaving it with execution has been ever devised. In this paper, we extend planning via symbolic model checking with the ability of interleaving planning and execution in the case of nondeterministic domains and partial observability, one of the most challenging and complex planning problems. We build a planning algorithm such that the interleaving of planning and execution is guaranteed to terminate, either because the goal is achieved, or since there is no longer chance to find a plan leading to the goal. We experiment with the planner and show that it can solve planning problems that cannot be tackled by the off-line symbolic model checking techniques.

Introduction

Recent research in planning addresses the problem of dealing with nondeterministic domains and partial observability, see, e.g., (Kabanza *et al.* 1997; Weld *et al.* 1998; Bertoli *et al.* 2001b; Rintanen 1999; Bonet and Geffner 2000; Jensen *et al.* 2001). Planning via symbolic model checking has been recently shown to be a promising approach that can tackle some problems that have never been solved before (Bertoli *et al.* 2001b; 2001a; Rintanen 2002). However, building plans purely off-line still remains unfeasible in most realistic applications, because of the complexity due to nondeterminism and partial observability. Methods that interleave planning and execution, see, e.g., (Koenig and Simmons 1998; Koenig 2001) are the practical alternative to the problem of planning off-line with large state spaces.

In this paper, we extend the framework of "planning via symbolic model checking" with the ability of interleaving planning and execution in the case of nondeterministic domains and partial observability. We define an architecture for interleaving plan generation and plan execution, where a planner generates conditional plans that branch over observations, and a controller executes actions in the plan and monitors observations to decide which branch has to be executed.

We propose a top-level algorithm for interleaving planning and execution that exploits the same data structures used in off-line planning via symbolic model checking (Bertoli *et al.* 2001b; 2001a) to generate plans, to execute them and to monitor their execution. Within this framework, we propose a novel "embedded" algorithm for plan generation that is based on the state-of-the-art off-line planning algorithm presented in (Bertoli *et al.* 2001a). At each plan generation step, the embedded algorithm generates plans that make a progress toward the goal without necessarily reaching it. This has the advantage that the algorithm does not necessarily need to consider all possible contingencies and can thus scale up to large state spaces.

An important desired property for the original algorithm is that it generates strong solutions, i.e., solutions that are guaranteed to achieve the goal in spite of nondeterminism and partial observability. In an interleaving framework, it is in general impossible to guarantee that a goal state will be reached, since execution can get trapped in a dead-end situation, from which there is no plan exists (anymore) that is guaranteed to reach the goal. However, the embedded algorithm guarantees that the planning/execution loop terminates either by reaching the goal, or by reaching a state where there is no chance to find a strong plan, ensured to reach the goal[1]. Moreover, thanks to this property, the embedded algorithm is guaranteed to reach the goal in safely explorable domains, i.e., domains where execution cannot get trapped in situations where no strong plan to the goal exists anymore.

We implement the algorithms for interleaving planning and execution, and experiment in the robot maze domain (Koenig 2001) augmented with uncertainty in actuators. We compare the new planner with a state-of-the-art off-line planner based on symbolic model checking, MBP (Bertoli *et al.* 2001b; 2001a). The experimental results show that the new planner can scale up to much harder problems than the off-line technique.

The paper is organized as follows. We provide first some intuitions on the problem of planning with nondeterminism and partial observability through an example (Section). We then formalize the algorithm for interleaving planning and execution and discuss the embedded planning algorithm (Section). We finally describe the experimental results (Section) and discuss some related work.

Planning with Nondeterminism and Partial Observability

As a reference example, consider the simple robot navigation domain in Figure 1, in the upper left corner, consisting of a 2x2 room with an extra wall. The robot can move in the four directions, provided that there is not a wall in the direction of motion. The action is not applicable, otherwise. At each time tick, the information of walls proximity in each direction is available to the robot. When planning under partial observability, we have to deal with conditions of uncertainty, i.e. belief states. A *belief state* is a collection of all the states of the world that are compatible with the

[1]Notice that considering weak plans, that may not reach the goal, would remove the guarantee of termination. Nondeterministic behaviors of the domain might in fact cause endless planning/execution loops where weak plans always exist, and everytime fail to reach the goal.

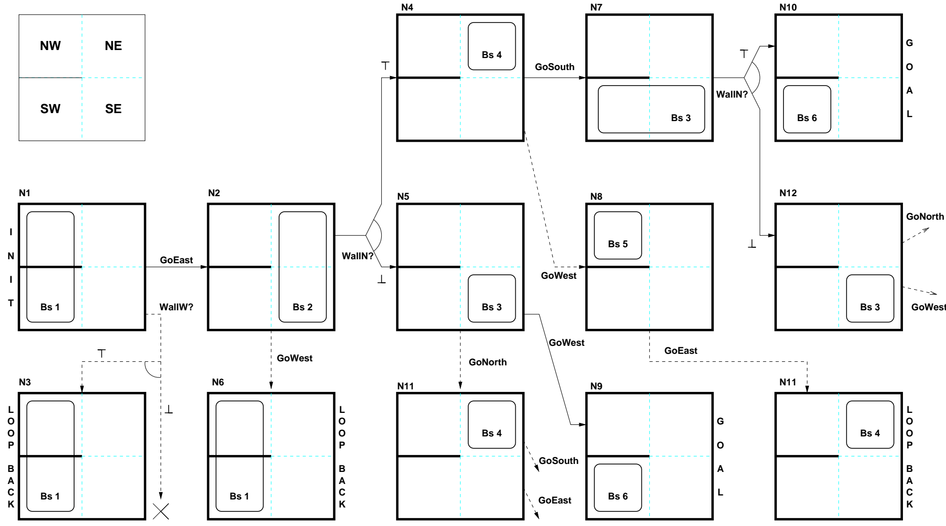


Figure 1: A simple robot navigation domain

initial knowledge and the information acquired through observation. The initial belief state in Figure 1 is $\{NW, SW\}$; our goal is to reach the condition $\{SW\}$. Actions transform belief states into new belief states, and observations identify subsets of the current belief state. In the example, the actions are `GoNorth`, `GoSouth`, `GoWest` and `GoEast`, and are deterministic, with the exception of moving `GoSouth` from $\{NE\}$, which may cause the robot to slip in one of two states. Observation variables are `WallN`, `WallS`, `WallW` and `WallE`.

The search space can be seen as an and-or graph, recursively constructed from the initial belief state, expanding each encountered belief state by every possible combination of applicable actions and observations. The graph is possibly cyclic; in order to rule out cyclic behaviors, however, its exploration – at planning time – can be limited to the acyclic prefix of the graph. Figure 1 depicts a portion of the finite prefix of the search space for the described problem. The prefix is constructed by expanding each node in all possible ways, each represented by an outgoing arc. Single-outcome arcs correspond to simple actions (action execution is deterministic in belief space). For instance, N4 expands into N7 by the action `GoSouth`. Multiple outcome arcs correspond to observations. For instance, node N2 results in nodes N4 and N5, corresponding to the observation of `WallN`.

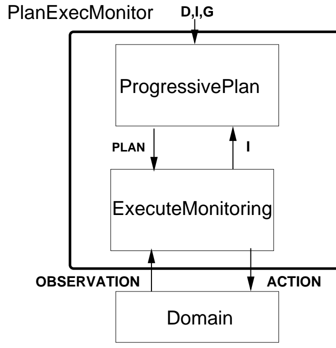
More formally, a partially observable nondeterministic planning domain is defined in terms of its *states*, of the *actions* it accepts, and of the possible *observations* that the domain can exhibit. Some of the states are marked as *initial*. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. Finally, an *observation function* defines what observations are associated to each state.

Definition 1 *A* nondeterministic planning domain with partial observability *is a tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$, *where:*
• $\mathcal{S}$ *is the set of* states.
• $\mathcal{A}$ *is the set of* actions.
• $\mathcal{O}$ *is the set of* observations.
• $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*; we require* $\mathcal{I} \neq \emptyset$.
• $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$ *is the* transition function*; it associates to each current state* s *and to each action* a *the set* $\mathcal{T}(s, a) \subseteq \mathcal{S}$ *of next states.*
• $\mathcal{X} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{O})$ *is the* observation function*; it associates to each state* s *the set of possible observations* $\mathcal{X}(s) \subseteq \mathcal{O}$.
Action a *is* executable *in state* s *if* $\mathcal{T}(s, a) \neq \emptyset$*; it is executable in a set of states* b *iff it is executable in every state* $s \in b$*. We require that in each state* $s \in \mathcal{S}$ *there is some executable action. We also require that some observation is associated to each state* $s \in \mathcal{S}$*, that is,* $\mathcal{X}(s) \neq \emptyset$.

We can have uncertainty in the initial states and in the outcome of action execution. Also, since the observation associated to a given state is not unique, it is possible to model noisy sensing and lack of information.

In the domain of figure 1, the actions are `GoNorth`, `GoSouth`, `GoWest` and `GoEast`. We have four states, corresponding to the four positions of the robot in the room. It is in general possible to present the state space by means of state variables. Each (atomic) state is associated with a truth assignment to the state variables. In the example, the state variables might be `E` and `S`. In state `NW` they would be both associated with a false value ($\bot$), while in `SE` they would be associated with $\top$. We have 16 observations, each corresponding to one of the possible configurations of walls around the robot. Again, the space of observation can be conveniently presented by means of observations variables, (e.g. `WallN`, `WallS`, `WallW` and `WallE`). Each observation associates a truth value to each observation variable. In the following, we will assume a variable-based presentation for $\mathcal{S}$ and $\mathcal{O}$, writing o to represents an observation variable. We define $\mathcal{X}_o[\top]$ to denote the set of states that are compatible with a $\top$ assignment to the observation variable o; $\mathcal{X}_o[\bot]$ is interpreted similarly. In partially observable domains, we consider plans that branch on the value of observable variables.

Definition 2 ((Conditional) Plan) *A plan for a domain* $\mathcal{D}$



is either the empty plan ϵ, an action $a \in \mathcal{A}$, the concatenation $a; \pi$ of an action and a plan, or the conditional plan $\mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2$*, with $o \in \mathcal{O}$.*

For instance, **if** `WallN` **then** `GoSouth` **else** `GoWest` corresponds to the plan "if you see a wall north, then move south, otherwise move west".

A plan is executable starting from a set of states b if b is empty, or:

- $\pi = \epsilon$, or b is empty, or
- $\pi = a; \pi_1$, a is executable on b, and π_1 is executable on $exec(a,b) = \bigcup_{t \in b} \mathcal{T}(t, a)$
- $\pi = \mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2$, and the plans π_1 , π_2 are executable over $b \cap \mathcal{X}_o[\top]$ and $b \cap \mathcal{X}_o[\bot]$ respectively.

Intuitively, given a domain $\mathcal{D}$, a set of initial states I and a set of goal states G in $\mathcal{S}$, a plan π is a strong solution for the planning problem $\langle \mathcal{D}, I, G \rangle$ iff it is executable on I, and every execution on the states of I results in G; see e.g. (Bertoli *et al.* 2001b).

Interleaving Planning and Execution

Rather than searching the huge and-or graph of belief states off-line, we propose a framework where a *planner* searches the graph partially, and a *controller* executes the partial plan and monitors the current state of the *domain*, then the process is iterated until the goal is hopefully reached.

The Top Level

The top level algorithm for embedded planning PLANEXECMONITOR is the following:

```
PLANEXECMONITOR(bs, G)
1  if (bs ⊆ G)
2      return Success;
3  π := PROGRESSIVEPLAN(bs, G);
4  if (π = Failure)
5      return Failure;
6  else
7      newbs := EXECUTEMONITORING(bs,π);
8      PLANEXECMONITOR(newbs, G);
```

The top level is first invoked by passing to it the set I of possible initial states, and the set of goal states G. We assume the domain representation to be globally available to its subroutines. The PLANEXECMONITOR routine provides a simple recursive implementation of a reactive planning loop; it relies on alternatively invoking a planner PROGRESSIVEPLAN to build a plan, and a monitored executor EXECUTEMONITORING that both executes it over the domain, and reports the new belief resulting from execution. It is easy to show that the top level stops either when the planner returns failure, or when a belief b is reached such that the goal is known to be reached (that is , $b \subseteq G$). The properties of the top level depends on those of the planner and monitored executor, which we discuss in turn.

```
EXECUTEMONITORING(bs, π)
1  MARKNODEASEXECUTED(bs);
2  if (π = ϵ)
3      return bs;
4  if (π = α; π′)
5      ACTUATE(α);
6      newbs := exec (α, bs);
7      EXECUTEMONITORING(newbs, π′);
8  if (π = if o then π1 else π2)
9      if CURRENTVALUE(o)
10         returnEXECUTEMONITORING (bs ∩ X_o[⊤],π1);
11     else
12         returnEXECUTEMONITORING (bs ∩ X_o[⊥],π2);
```

Figure 2: The algorithm for Execution and Monitoring

Execution and Monitoring

Execution and monitoring can be described in terms of runs, i.e., in terms of sequences of belief states generated along the execution. Given an initial belief state b_0, a plan can generate a set of possible runs, i.e., sequences of belief states, due to nondeterminism and noisy sensing.

Definition 3 (Runs of a plan) *Let π be a plan for a domain $\mathcal{D}$. The set of runs of π from an initial belief state $b_0 \subseteq \mathcal{S}$ is inductively defined as follows.*
• If π is ϵ, then b_0 is a run of π from b_0.
• If π is $a; \pi_1$, then the sequence b_0, r is a run of π from b_0, where r is a run of π_1 from $\{s : s \in \mathcal{T}(s_0, a)$ and $s_0 \in b_0\}$.
• If π is $\mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2$*, then the sequences b_0, r_1 and b_0, r_2 are runs of π from b_0, where r_1 is a run of π_1 from $b_0 \cap \mathcal{X}_o[\top]$, and r_2 is a run of π_2 from $b_0 \cap \mathcal{X}_o[\bot]$.*

For the top level to be sound, the monitoring executor must guarantee that, for each run $r = b_0, \ldots, b_n$ of the plan, it returns a belief state $b \subseteq b_n$. An executor that guarantees this property is presented in Fig.2. The executor EXECUTEMONITORING recursively applies the plan actions to the domain, via ACTUATE. The plan execution is driven by the observations in the plan: it branches over the actual observation values, retrieved from the domain via CURRENTVALUE. Parallel to this, EXECUTEMONITORING exploits the domain model, namely $\mathcal{X}$ and $exec$ (where $exec(a,b) = \bigcup_{t \in b} \mathcal{T}(t, a)$), to have the initial belief progress consistently with the execution. Each belief state traversed during the monitored execution is marked as traversed via MARKNODEASEXECUTED.

Strong Progressive Planning

In order to guarantee the termination of the top level, the planner must be *progressive*, i.e. the plans it produces must

```
PROGRESSIVEPLAN(I, G)
1  graph := MKINITIALGRAPH(I, G);
2  while ( ¬ISSUCCESS(GETROOT(graph)) ∧
3          ¬ISEMPTYFRONTIER(graph) ∧
4          ¬(ISPROGRESS(GETROOT(graph)) ∧
5            TERMINATIONCRITERIA(graph)))
6      node := EXTRACTNODEFROMFRONTIER(graph);
7      if (SUCCESSPOOLYIELDSSUCCESS(node, graph))
8          MARKNODEASSUCCESS(node);
9          NODESETPLAN(node,RETRIEVEPLAN(node, graph));
10         PROPAGATESUCCESSONTREE(node,graph);
11         PROPAGATESUCCESSONEQCLASS(node,graph);
12     else
13         orexp := EXPANDNODEWITHACTIONS(node);
14         andexp := EXPANDNODEWITHOBSERVATIONS(node);
15         EXTENDGRAPHOR(orexp, node, graph);
16         EXTENDGRAPHAND(andexp, node, graph);
17         if (SONSYIELDSUCCESS(node))
18             MARKNODEASSUCCESS(node);
19             NODESETPLAN(node,BUILDPLAN(node));
20             PROPAGATESUCCESSONTREE(node, graph);
21             PROPAGATESUCCESSONEQCLASS(node, graph);
22     if (¬ISEXECUTED(node) ∨ ISSUCCESS(node)
23         MARKNODEASPROGRESS(node, graph);
24         PROPAGATEPROGRESSONTREE(node, graph);
25         PROPAGATEPROGRESSONEQCLASS(node, graph);
26 end while
27 if (ISSUCCESS(GETROOT(graph)))
28     return EXTRACTSUCCESSPLAN(graph);
29 if (ISPROGRESS(GETROOT(graph))
30     return EXTRACTPROGRESSINGPLAN(graph);
31 return Failure;
```

Figure 3: The planning algorithm

be guaranteed to traverse at least one belief state that has not been previously encountered during execution.

Definition 4 (Progressive Plan) *Let r be a run $b_1, \dots, b_n$. Let π be a plan for $\mathcal{D}$. The plan π is progressive for the run r iff, for any run r_π of π from b_n, there is at least one belief state in r_π that is not a belief state of r.*

Figure 3 presents a progressive planning algorithm. It takes as input the initial belief state and the goal belief state, and proceeds by incrementally constructing a finite acyclic prefix of the search space, implemented as a $graph$. In the graph, each node n is associated with a belief state $b(n)$; a directed connection $n_1 \rightarrow n_2$ between a node n_1 and a node n_2 results either from an action α such that $Exec(\alpha, b(n_1)) = b(n_2)$, or from an observation o such that $b(n_1) \cap \mathcal{X}_o[v] = b(n_2)$, with $v = \top$ or $v = \bot$. In that case, we call n_1 the father of n_2 and n_2 the son of n_1; we call "brothers" all the nodes that result from the same observation expansion of the same node. The graph is annotated with a frontier of the nodes that have not yet been expanded, and with a success pool, containing the nodes for which a strong plan has been found. In addition, the graph defines equivalence classes for the nodes that share the same belief state.

In order to guarantee that the produced plans are progressive, all the conjunctive branches in the search graph need to be taken into account. The graph is therefore extended in order to maintain up-to-date information on progress of nodes. The core of the algorithm consists of a search loop (lines 2-26), iteratively selecting and expanding a node in the graph, previously initialized by constructing its root, corresponding to I, and the success pool $\{G\}$ (line 1). The loop terminates either when (a) the root of the graph is signaled as a success node, (b) the graph frontier is empty, or (c) a progressing plan has been found and an arbitrary termination criterion holds true (lines 2-5). Case (a) takes place when a strong plan is found; case (b) corresponds to the failure of the planner, and case (c) realizes progressing replanning.

Inside the loop, first a $node$ is extracted from the frontier (line 6). The EXTRACTNODEFROMFRONTIER primitive embodies the selection criterion and is responsible, together with the termination criterion, for the style (and the effectiveness) of the search being carried out. Then, at line 7, we check, via the primitive SUCCESSPOOLYIELDSSUCCESS, whether there is a node n_{succ} in the success pool such that $b(node) \subseteq b(n_{succ})$. If so, the algorithm takes care of the newly solved node. In particular, $b(node)$ is marked as success, and inserted in the success pool (MARKNODEASSUCCESS, line 8). Then, at line 9, $node$ is associated with the plan that has been previously computed for n_{succ}. (As a consequence, the plan constructed is in fact a DAG rather than a tree.) At line 10, the PROPAGATESUCCESSONTREE primitive propagates success bottom-up as follows. If $node$ is the result of an action expansion, the father node n_f is marked as success, and the open descendants of n_f are removed from the frontier, since their expansion is no longer necessary. If $node$ is the result of an observation, the above process is carried out only if all the brothers of $node$ are already marked as success. In this case, n_f is associated with a conditional plan. If n_f is marked as success, bottom-up success propagation process is iterated on n_f. Bottom-up propagation may reach the root, in which case the problem is solved. At line 11, the primitive PROPAGATESUCCESSONEQCLASS triggers bottom-up success propagation for each node in the equivalence class of $node$.

If the success of $node$ is not entailed by the success pool, then the expansion of $node$ is attempted, computing the nodes resulting from possible actions (line 13) and observations (line 14). The result of EXPANDNODEWITHACTIONS is the list of belief state-action pairs $\langle exec(\alpha, b(node)) \,.\, \alpha \rangle$. Similarly, EXPANDNODEWITHOBSERVATIONS returns a list of observation results, each composed of an observation mask, describing what variables are being observed, and a list of pairs $\langle b_i \,.\, v_i \rangle$, each associating a belief state b_i to an observation results $v_i \in \{\top, \bot\}$. The graph extension steps, at lines 15-16, construct the nodes associated to the expansion, and add them to the graph, also doing the bookkeeping operations needed to update the frontier and the links between nodes. Loops in the search are avoided, by checking that an expanded node does not belong to the list of its ancestors.

Once expanding the graph is done, if it is possible to state the success of $node$ based on the status of the newly introduced sons (primitive SONSYIELDSUCCESS at line 17), then the same operations at line 8-11 for success propagation are executed. The plan for $node$, in this case, is built by BUILDPLAN, starting from the plans associated to the sons

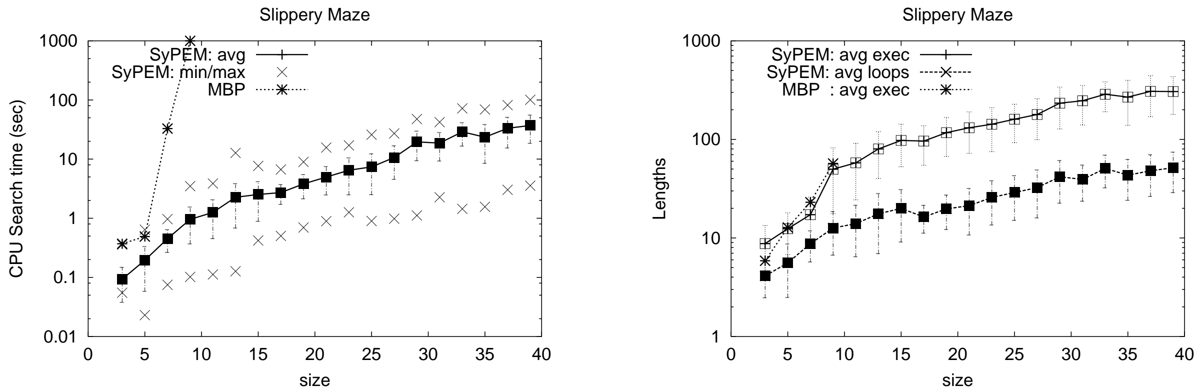


Figure 4: Search times and Plans for the Slippery Maze

of $node$. In order to guarantee progressiveness, at line 22, we check if $b(node)$ has already been visited at execution time. If not (or if $node$ is marked as successful), mark $node$ as a "progress" node (line 23). In that case, the progress information is recursively propagated bottom-up on the tree (PROPAGATEPROGRESSONTREE, line 24): if the node is the result of the application of an action, then its father is marked as progress. If the node is the result of an observation, in order to propagate its progress backward it is necessary to check that all of its brothers are also marked as progress nodes. Progress is also propagated on the equivalence classes, similar to what happens for the success (line 25).

Finally, when the loop is exited, either a strong plan has been found, and is returned by EXTRACTSUCCESSPLAN; or, a progressing plan exists in the graph, and is extracted by EXTRACTPROGRESSINGPLAN; or, failure is returned. While extracting the success plan is simple (it is associated with I by the bottom-up propagation), the progressing plan might not be unique: several such plans may exist. The selection operated by EXTRACTPROGRESSINGPLAN may affect the overall performance. Our implementation privileges, amongst progressing plans, the ones performing more observations.

Discussion

The algorithm presented above is guaranteed to terminate, based on the fact that the explored search space is finite. It either returns a strong solution plan, a progressing plan w.r.t. the current run for the top level, or failure. The planner is guaranteed to be progressive by the fact that the main loop cannot exit before a progressing plan is found, unless the whole reachable belief space has been searched, or success takes place. Notice that the termination criteria is inhibited if the progressiveness condition holds false. Also, the planner is guaranteed not to terminate before the reachable belief space is exhausted, thus it will not return failure for the planning problem corresponding to the current run, unless no chance of producing a strong plan for the goal exists. Clearly, this is not enough to guarantee completeness of the top level. This is due to the possibility of generating and executing a series of progressing plans that lead the domain in a situation where no strong plan to reach the goal exists anymore (even if it existed at the start). However, notice that for domains guaranteed to be "safely explorable" (see (Koenig and Simmons 1998)), this can never be the case.

Experiments

We implemented a planner called SyPEM (Symbolic Planner with Execution and Monitoring – the name is fictitious name for the sake of blind review) based on the algorithm shown in Section . SyPEM uses symbolic data structures based on Binary Decision Diagrams (BDDs), similarly to planners such as MBP (Bertoli *et al.* 2001b) and UMOP (Jensen *et al.* 2001). BDDs are extremely well suited to represent and handle large sets of states, and provide highly efficient primitives for the key low-level state-set operations in the algorithm. For our experiments, we considered a set of robot navigation problems in a maze, similar to our reference example. The robot may start at any position in the maze, and has to reach the top left corner. The robot may move in the four directions, and is equipped with (reliable) wall-presence sensors in the four directions. The main variation is that the robot may slip on the floor while trying to move, so that it stays in the same position. Slipping can occur nondeterministiccally, at most once every $N \geq 5$ moves. The problems always have a strong solution. However, finding a strong solutions offline is extremely complex, even for small mazes (see (Tovey and Koenig 2000)). An algorithm looking for an offline solution has to face the combined effect of the initial uncertainty, and of the increased uncertainty resulting from slipping, i.e. a very high branching rate in the search space. The belief states include not only the possible robot positions, but also the hypothesis on whether there is currently a chance of slipping on the floor.

We tested a set of domains and problems, comparing the interleaved approach of SyPEM with the state of the art (offline) planner MBP. We ran MBP with all the available selection functions; we report the results obtained with the one that provided the best results for the tackled problem. The tests were run on a Pentium III, 700 MHz with 6GB RAM running Linux. The memory limit was set to 512MB, and CPU timeout was set at 1000 sec. For both systems, we collected information on performance and quality of the associated executions. Figure 4 reports the results. On the left, we

have running times. For MBP we only report planning times. For SyPEM, running times include both planning and execution on a simulator. The information is statistical, since the performance of the architecture may depend on the actual behavior of the domain, i.e. on the actual initial state, and on the outcomes of nondeterministic actions. For each problem instance 100 runs were generated, with initial states and nondeterministic outcomes selected randomly. (Whenever possible, we avoid runs from the same initial state.) For each problem, we report the average runtimes, the standard deviation, and also the minimum and peak value. On the right in Figure 4, we report information concerning the quality of the plans (for MBP) and of the executions (for SyPEM). In particular, we report the average number of actions in the plans produced by MBP, and the average number of total actions executed by SyPEM. For SyPEM, we also report the average number of execution loops. Again, we report max/min/standard deviation for the samples.

MBP is unable to solve the problem instances above 9x9. Notice that, MBP is reported to scale up much better in the deterministic version of the maze problem (see (Bertoli *et al.* 2001a)), dealing with mazes of 51x51 in less than a minute. This confirms the intuition on the complexity of the problem. SyPEM, on the other hand, is able to deal with much larger mazes, showing average times that increase smoothly and are less than 40 seconds for mazes as large as 39x39.

The experimental evaluation is limited by the choice of the test cases, and should be seen more as a proof of concept than a strong point in support of a scalability argument. Still, we believe that the results are very promising: the interleaved approach by SyPEM scales up much better, and is capable of efficiently dealing with very complex problem instances, that cannot be dealt by the state-of-the art planner MBP. Intuitively, this is an obvious consequence of the interleaving architecture exploiting the runtime-acquired knowledge to restrict and drive the search.

Conclusions and Related work

The idea of interleaving planning and execution is certainly not new and has been around for a long time, see, e.g., (Genereseth and Nourbakhsh 1993). However, as far as we know, no previous attempt has been done to extend the planning via symbolic model checking framework to deal with the interleaving of planning and execution. Some other approaches address the problem of interleaving planning with nondeterministic domains and partial observability, among which, most notably the work by (Koenig and Simmons 1995; 1998; Koenig 2001), which proposes different techniques based on real-time heuristic search. The algorithms and heuristics prevent cycling, and guarantee to reach the goal in safely explorable domains. Algorithms presented in (Koenig 2001) have the nice property that they can amortize learning over several planning episodes. This approach can also be modified to address the problem of planning in stochastic domains with probability distributions on action outcomes, like in POMDP (see, e.g., (Bonet and Geffner 2000; Kaelbling *et al.* 1998; Cassandra *et al.* 1994; Dean *et al.* 1995)). We propose a very different technique, based on symbolic model checking rather than real time heuristic search on an explicit state representation. The domain for testing our approach is inspired by the work by Koenig, which has been tested extensively in the problem of robot navigation and localization. However, it should be noticed that the experimental domain of Section is much harder than the one used in (Koenig 2001), which assumes that there is no uncertainty in actuation and sensing. It would be interesting an in depth experimental comparison in different domains of the two different approaches.

Somehow related to our work, even if very different in scope and objective, are all the works that propose architectures for interleaving planning and execution, reactive planning and continuous planning, see, e.g., (Myers 1998). Among them, CIRCA (Goldman *et al.* 1997; 1999; 2000) is an architecture for real-time planning and execution where model checking with timed automata is used to verify that generated plans meet timing constraints.

References

P. Bertoli, A. Cimatti, and M. Roveri. Conditional planning under partial observability as heuristic-symbolic search in belief space. In *Proceedings of the Sixth European Conference on Planning (ECP'01)*, 2001.

P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 473–478. Morgan Kaufmann Publishers, August 2001.

B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 52–61. AAAI Press, April 2000.

A. Cassandra, L. Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1023–1028. AAAI Press, August 1994.

T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

M. Genereseth and I. Nourbakhsh. Time-saving tips for problem solving with incomplete information. In *Proceedings of the National Conference on Artificial Intelligence*, pages 724–730, 1993.

R. P. Goldman, D. J. Musliner, K. D. Krebsbach, and M. S. Boddy. Dynamic abstraction planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI 97), (IAAI 97)*, pages 680–686. AAAI Press, 1997.

R.P. Goldman, M. Pelican, and D.J. Musliner. Hard Real-time Mode Logic Synthesis for Hybrid Control: A CIRCA-based approach, mar 1999. Working notes of the 1999 AAAI Spring Symposium on Hybrid Control.

R. P. Goldman, D. J. Musliner, and M. J. Pelican. Using model checking to plan hard real-time controllers. In *Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, Breckeridge, Colorado, April 2000.

R. M. Jensen, M. M. Veloso, and M. H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the Sixth European Conference on Planning (ECP'01)*, 2001.

F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observale domains. *Artificial Intelligence*, 1-2(101):99–134, 1998.

S. Koenig and R. Simmons. Real-time search in non-deterministic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1660–1667. Morgan Kaufmann Publisher, August 1995.

S. Koenig and R. Simmons. Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In R. G. Simmons, M. Veloso, and S. Smith, editors, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 145–153. AAAI Press, 1998.

S. Koenig. Minimax real-time heuristic search. *Artificial Intelligence*, 129(1):165–197, 2001.

K. L. Myers. Towards a framework for continuous planning and execution. In *Proceedings of the AAAI Fall Symposium on Distributed Continual Planning*, 1998.

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

J. Rintanen. Backward plan construction for planning as search in belief space. In *n Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 2002.

C. Tovey and S. Koenig. Gridworlds as Testbeds for Planning with Incomplete Information. In Proceedings of the National Conference on Artificial Intelligence, pages 819–824, 2000.

D. S. Weld, C. R. Anderson, and D. E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 897–904, Menlo Park, July 26–30 1998. AAAI Press.