UPMurphi: A Tool for Universal Planning on PDDL+ Problems

Giuseppe Della Penna and **Daniele Magazzeni** and **Fabio Mercorio**
Department of Computer Science, University of L'Aquila, Italy
{Giuseppe.DellaPenna,Daniele.Magazzeni,Fabio.Mercorio}@univaq.it

Benedetto Intrigila
Department of Mathematics, University of Roma "Tor Vergata", Italy
intrigil@mat.uniroma2.it

Abstract

Systems subject to (continuous) physical effects and controlled by (discrete) digital equipments, are today very common. Thus, many realistic domains where planning is required are represented by *hybrid systems*, i.e., systems containing both discrete and continuous values, with possibly a nonlinear continuous dynamics. The PDDL+ language allows one to model these domains, however the current tools can generally handle only *planning* problems on (possibly hybrid) systems with *linear* dynamics. Therefore, *universal planning* applied to hybrid systems and, in general, to nonlinear systems is completely out of scope for such tools. In this paper, we propose the use of explicit model checking-based techniques to solve universal planning problems on such hardly-approachable domains.

1. Introduction

An interesting feature of many realistic domains where planning and universal planning are required is the presence of both discrete and continuous values, which are also commonly regulated by complex (e.g., differential) equations. Such systems are usually called *hybrid systems*. Hybrid systems are very common in contexts where (continuous) physical rules are managed by (discrete) digital equipments: for example, the product processing in a plant (Aylett et al. 1998) or the activity management of an autonomous vehicle (Léauté and Williams 2005) are both well-known problems in the planning field, where reasoning about continuous change is fundamental to the planning process (Fox and Long 2006).

Moreover, several real world (universal) planning problems present complex nonlinear behaviors which are difficult to handle by any analytical method (Sontag 1995). Nonlinearity can arise from the intrinsic dynamics of the system (e.g., the regulation of a steering antenna, which leads to an inverted pendulum problem), or the saturation of actuators (e.g., valves that cannot open more than a certain limit, control surfaces in an aircraft that cannot be deflected more than a certain angle, etc.). Indeed, the behavior of nonlinear systems can be so complex to be completely unpredictable after a small interval of time (see, e.g., (Schuster 1988)).

1.1 Motivation

In the past years, a relevant effort has been made to extend PDDL (The AIPS-98 Planning Competition Committee 1998), the standard modelling language for planning problems, to PDDL+ (Fox and Long 2006; 2001), which allows one to model domains presenting both discrete and continuous variables and continuous resource consumption (Edelkamp 2003). Also the planning algorithms have been improved to cope with this extended setting (Edelkamp 2002), and the current tools have been proved very useful to perform planning w.r.t. such problems. However, universal planning on PDDL+ problems is still an open issue.

1.2 Contribution

In this paper, we propose the use of explicit model checking-based techniques to solve universal planning problems on hardly-approachable domains like hybrid systems and nonlinear systems.

In particular, we have developed the *UPMurphi* universal planner, which is able to build the transition graph for a large class of systems (including systems whose dynamics is hard to be inverted), by exploiting the capabilities of explicit model checking algorithms to deal with huge state spaces and complex dynamics. UPMurphi is thus able, given a goal to achieve, to generate a universal plan.

As an added value to our tool, and to easily integrate this new technology in the planning community, we designed and implemented a compilation procedure which allows the user to directly feed UPMurphi with PDDL+ domain and problem specifications, without the need to translate them in different formalisms, as usually required by other (especially model checking based) approaches. Finally, UPMurphi automatically generates PDDL+ plans and validates them using the well-known PDDL+ validator VAL (Howey, Long, and Fox 2004).

Therefore, the presented tool aims to offer a fully PDDL+ compliant universal planner that is able to cope with problems that are very hard to handle by the current state-of-the-art tools.

The UPMurphi implementation described in this paper is not final, but the presented case study shows how the tool can be used to solve a well-known and complex planning problem in a relatively easy way.

The paper is organized as follows. In Sec. 2we give an overview of the related work. Then, in Sec. 3we define the universal planning problem and describe the corresponding model checking based algorithm. The UPMurphi tool is presented in Sec. 4.where we also describe the PDDL+ compilation process. Sec. 5shows the experimental results related to the *planetary lander* case study. Finally, Sec. 6contains some concluding remarks.

2. Related Work

The PDDL+ language (Fox and Long 2006; 2001) is an extension of the PDDL language (The AIPS-98 Planning Competition Committee 1998) that allows one to model continuous change through the use of autonomous processes and exogenous events.

A great effort has been made to develop algorithms and tools that are able to solve PDDL+ planning problems (Edelkamp 2001; Shin and Davis 2005) and, more generally, planning problems with time and resources constraints (Smith and Weld 1999; Fox and Long 2002; Penberthy and Weld 1994). Unfortunately, none of the above approaches is capable to manage the dynamics of hybrid and nonlinear systems, which can be modelled through PDDL+.

The LPSAT planner (Wolfman and Weld 1999) is able to solve resource planning problems with real values, but it cannot handle PDDL+ features such as, e.g., processes or events, whereas TM-LPSAT (Shin and Davis 2005) can solve PDDL+ problems only on linear domains. This is also the case of the UPPAAL/TIGA tool (Behrmann et al. 2007; UPPAAL-TIGA web page 2009) that, being built on top of UPPAAL, allows one to use real variables only as clocks (i.e., real variables can be modelled only if their first derivative is 1), thus excluding systems with nonlinear dynamics. Furthermore, the Kongming planner (Li and Williams 2008), thanks to the concept of Flow Tubes, is able to compactly represent hybrid plans and encode hybrid flow graphs as a mixed logic linear/nonlinear program, solvable using an off-the-shelf solver. However, it can only address planning problems with constant action duration and does not support the use of PDDL+.

Moreover, the tools above are only planners. Universal planning is well known as a harder problem, and it cannot be efficiently solved by applying the same algorithms and tools of planning. Indeed, in this field, the research is still trying to provide a reliable and at least semi-automatic support to simple universal planning problems.

In particular, (Cimatti, Roveri, and Traverso 1998a; 1998b; Jensen and Veloso 2000) use a symbolic (OBBD-based) model checking approach to obtain optimal universal plans for non-deterministic plants. On the other hand, the DPlan (Schmid 2003) and FPlan (Schmid and Wysotzki 2000) tools use an explicit state-based representation instead of OBDDs. However, they both require the explicit definition of an inverse function for each operator used in the domain, and thus their application is hard when dealing with systems whose dynamics is difficult to invert (the typical situation for hybrid and/or nonlinear systems).

3. Planning through Explicit Model Checking

Model checking algorithms are typically divided in two categories: *symbolic* algorithms (e.g., (Burch et al. 1992)) and *explicit* algorithms (e.g., (Edelkamp, Lafuente, and Leue 2001)). Symbolic algorithms, however, do not work well on hybrid systems with nonlinear dynamics, due to the complexity of the state transition function (Della Penna et al. 2003). Therefore, explicit model checking performs better with the kind of systems we intend to approach. Also these algorithms are subject to the *state explosion problem*: however, the ability to build the system transition graph *on demand* and generate only the reachable states of the system (through the *reachability analysis*), together with many space saving techniques (see, e.g., (Della Penna et al. 2004)), help to mitigate it.

Generally speaking, given a set E of *error states*, a model checker looks, via an exhaustive search, for an error sequence leading to an error state $e \in E$. If we look at error states as *goal states*, and collect *all* the error sequences instead of the first one, we can use a model checker as a universal planner. This very simple fact allows one to use the model checking technology to automatically synthesize universal plans for complex systems.

3.1 Universal Planning on Finite State Systems

A hybrid system is a system whose state description involves continuous as well as discrete variables. In order to apply model checking algorithms and exploit the reachability analysis, the system should have a finite number of states. To this aim, we approximate the system by discretizing the continuous components of the state (which we assume to be bounded) and their dynamic behavior. For lack of space, we cannot describe here the approximation process, however the reader can see how our approach works by looking at the case study in Sec. 5.a discussion about the mapping from hybrid automata to LTSs can be found, e.g., in (Fox and Long 2006; T. Henzinger 1996). In the following we first give a formal definition of the approximated model, the *finite state system*, and then describe how the universal planning problem can be solved for such system.

Definition 1 (Finite State System) *A* Finite State System *(*FSS*) $\mathcal{S}$ is a 4-tuple (S,I,A,F), where: S is a finite set of* states*, $I \subseteq S$ is a finite set of* initial states*, A is a finite set of* actions *and $F : S \times A \times S \to \{0,1\}$ is the* transition function*, i.e. $F(s,a,s') = 1$ iff the system from state s can reach state s' via action a.*

By abuse of language, we denote with $F(s,a)$ the set of successor states of s through action a, i.e. $F(s,a) = \{s' | F(s,a,s') = 1\}$.

In order to define the universal planning problem for such system, we assume that a set of *goal states* $G \subseteq S$ has been specified. Moreover, to have a finite state system, we fix a *finite temporal horizon* T and we require each plan to reach the goal in at most T actions. Note that, in most practical applications, we always have a maximum time allowed to complete the execution of a plan, thus this restriction, although theoretically quite relevant, has a limited practical impact.

Now we are in position to state the universal planning problem for finite state systems.

Definition 2 (Universal Planning Problem on FSS) *Let $\mathcal{S} = (S, I, A, F)$ be an FSS. Then, an* universal planning problem *(*UPP *in the following) is a quadruple $\mathcal{P} = (\mathcal{S}, G, C, T)$ where $G \subseteq S$ is the set of the goal states, $C : S \times A \to \mathbb{R}^+$ is the cost function and T is the finite temporal horizon.*

Intuitively, a solution to an UPP is a set of minimal cost paths in the system transition graph, starting from any system state state and ending in a goal state. More formally, we have the following.

Definition 3 (Trajectory) *A* trajectory *in the FSS $\mathcal{S} = (S, I, A, F)$ is a sequence $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots$ where, $\forall i \geq 0$, $s_i \in S$ is a state, $a_i \in A$ is an action and $F(s_i, a_i, s_{i+1}) = 1$. If π is a trajectory, we write $\pi_s(k)$ (resp. $\pi_a(k)$) to denote the state s_k (resp. the action a_k). Finally, we denote with $|\pi|$ the length of π, given by the number of transitions.*

By abuse of language we denote with $C(\pi)$ the cost of a trajectory $\pi = s_0 a_0 s_1 a_1 \ldots s_k$, i.e., $C(\pi) = \sum_{i=0}^{k-1} C(s_i, a_i)$.

Definition 4 (Reachable States) *Let $s \in I$ be an initial state of the FSS $\mathcal{S} = (S, I, A, F)$. Then, we say that a state s' is* reachable *from s iff there exists a trajectory π in $\mathcal{S}$ such that $\pi_s(0) = s$ and $\pi_s(k) = s'$ for some $k >= 0$. We denote with* Reach(s) *the set of states reachable from s.*

Definition 5 (Solution for UPPs) *Let $\mathcal{S} = (S, I, A, F)$ be an FSS and let $\mathcal{P} = (\mathcal{S}, G, C, T)$ be an UPP. Then a solution for $\mathcal{P}$ is a map $\mathcal{K}$ from S to A s.t. $\forall s \in$* Reach(s_I) *with $s_I \in I$ there exist $k \leq T$ and a trajectory π^* in $\mathcal{S}$ s.t.: $\pi^*_s(0) = s$, $\forall t < k : \pi^*_s(t+1) = F(\pi^*_s(t), \mathcal{K}(\pi^*_s(t)))$ and $\pi^*_s(k) \in G$. We denote with $\mathcal{K}_\pi(s)$ the trajectory π^* generated by $\mathcal{K}$ and s.t. $\pi^*_s(0) = s$.*

An optimal solution *is a solution $\mathcal{K}$ s.t. for all other solutions $\mathcal{K}'$ the following holds: for all $s \in S$ s.t. $\mathcal{K}_\pi(s)$ and $\mathcal{K}'_\pi(s)$ are defined, then $C(\mathcal{K}_\pi(s)) \leq C(\mathcal{K}'_\pi(s))$.*

In the next section, we present an algorithm which takes as input an UPP and outputs an optimal solution for it.

3.2 Universal Planning Algorithm

Since we are interested in hybrid systems, possibly having a nonlinear dynamics, and we know that symbolic approaches do not work well on such systems, we adopt an explicit algorithm, organized in two phases, to solve the UPP.

In the first phase, we exploit the *reachability analysis* performed by an explicit model checking algorithm in order to build a representation of the system dynamics that can be later easily analyzed during the universal plan generation. The corresponding BUILD_GRAPH procedure, whose pseudocode is given in Fig. 1, can be seen as an extension of the common breadth-first visit performed by many explicit model checking algorithms.

Note that, in the general theory of universal planning, the concept of start state is not present (Schoppers 1987). However, in the practice, the concept of reachable state implies such a start state. In other words, we need to start-up the universal planning with a set of start states, that we call a *start state cloud*. These states should be distributed in the system state space so that all the interesting states are reachable from at least one of them. This can be accomplished, with a reasonable approximation, through a random selection algorithm. However, a start state cloud can be also suitably prepared to concentrate the planning process on the most interesting state space regions, or to exclude hardly reachable states from the universal plan. Indeed, a complete universal plan could generally contain many rarely-used plans, whose computation requires however time and space. Therefore, an user-assisted formulation of the start state cloud is a key feature of our algorithm, helping to minimize the universal plan generation effort and maximize its usefulness.

```
1  BUILD_GRAPH(UPP $\mathcal{P} = (\mathcal{S}, G, C, T)$) {
2   let $\mathcal{S}$ = ($S$, $I$, $A$, $F$);
3   foreach $s \in I$ {
4    Enqueue(Q_S, $s$); Insert(HT, $s$);
5    if($s \in G$) {Enqueue(Q_G, $s$); HT[$s$].cost := 0;}
6   }
7   while((Q_S $\neq \emptyset$) $\wedge$ (current_BF_level $\leq$ T)) {
8    $s$ := Dequeue(Q_S);
9    foreach $s' \in \{F(s, a) \mid (a) \in A\}$ {
10    if($s' \notin$ HT) {
11     Insert(HT, $s'$);
12     if($s' \in G$) {Enqueue(Q_G, $s'$); HT[$s'$].cost := 0;}
13     else Enqueue(Q_S, $s'$);
14    }
15    PT[$s'$] := PT[$s'$] $\cup$ $\{s\}$;
16 } } }
```

Figure 1: The BUILD_GRAPH procedure

The procedure uses the hash table HT to store already visited states, while the queues Q_S and Q_G store the states to be expanded and the reached goal states (to be used in the next phase), respectively. This information is also used to detect and exploit trajectories intersections, so avoiding work duplication. Note that the computation of the successor states involves discretized values, i.e., continuous components of both s and s' in line 9 of Fig. 1 are rounded according to the chosen discretization. Finally, the predecessor table PT contains the immediate predecessors of each visited state. This structure is at the heart of the second phase of the algorithm, represented by the UPLAN_GENERATION procedure, whose pseudocode is given in Fig. 2.

The UPLAN_GENERATION procedure performs another breadth first visit, this time on the *inverted* transition graph, starting from the reached goal states. To this end, the procedure uses the information in Q_G, HT and PT prepared by BUILD_GRAPH. The output is the table UPLAN, containing (state,action) pairs that represent the map $\mathcal{K}$ as described in Definition 5.

In particular, the check on line 11 of Fig. 2, which updates the action associated to a state only if either no action has been defined yet or the current action leads to a better result, and the ordered insertion in the queue Q_S, guarantee

```
UPLAN_GENERATION() {
 UPLAN := $\emptyset$;
 Q_S := Q_G; //this erases the previous content of Q
 while(Q_S $\neq \emptyset$) {
  $s$ := Dequeue(Q_S);
  prev_cost := HT[$s$].cost; //0 if $s \in G$
  foreach $\vec{s}$ $\in$ PT[$s$] { //$\vec{s}$ is a predecessor of $s$
   local_cost := $\min_{(a)\in A \mid F(\vec{s},a)=(s)} C(\vec{s},a)$;
   U := $\{a \in A \mid F(\vec{s},a) = s \wedge C(\vec{s},a) =$local_cost$\}$;
   local_action := pick an action in U;
   if(UPLAN[$\vec{s}$]=$\emptyset \vee$HT[$\vec{s}$].cost>prev_cost+local_cost){
    UPLAN[$\vec{s}$] := local_action;
    HT[$\vec{s}$].cost := prev_cost + local_cost;
    Enqueue_in_Order(Q_S, $\vec{s}$);
 } } }
 return UPLAN;
}
```

Figure 2: The UPLAN_GENERATION procedure

that the algorithm returns an *optimal solution* according to Definition 5.

Note that, since our approach rebuilds the system transition graph by a forward analysis of its dynamics, the system fed to the planning algorithm can be of any complexity, and in particular its transition function can be also very difficult to invert.

4. The UPMurphi Universal Planner

The UPMurphi tool is built on top of the CMurphi (Cached Murphi web page 2006) model checker. In particular, UPMurphi consists of two main modules: the *PDDL+ to UPMurphi compiler*, which automatically translates PDDL+ domains and problems into models written in the CMurphi description language, and the *UPMurphi engine*, which is the core of the tool and implements the BUILD_GRAPH and UPLAN_GENERATION algorithms on the top of the CMurphi algorithms and data structures. Thanks to this, the universal planning algorithm presented in Sec. 3.2 can exploit all the CMurphi built-in state space optimization techniques (such as bit compression (Murphi web page 2004), hash compaction (Stern and Dill 1995), symmetry reduction (Ip and Dill 1993), secondary memory storage and state space caching (Della Penna et al. 2004)) to handle large systems with huge state spaces.

The CMurphi input consists of a description of the system to be verified, modelled as a finite state system, and a definition of the property to be checked, both described in the *CMurphi description language*, a high-level programming language for finite state asynchronous concurrent systems, which offers many features found in common high-level programming languages such as Pascal or C, like user-defined data types, functions and procedures.

Moreover, CMurphi provides two important features to ease the hybrid systems modelling activity: the type real(m,n) of real numbers (with m digits for the mantissa and n digits for the exponent), and the use of externally defined C/C++ functions in the modelling language. In this way, for example, one can use the C/C++ language constructs and library functions to model complex dynamics.

The behavioral part of a CMurphi model is a collection of *transition rules*. Each transition rule is a guarded command which consists of a condition (a boolean expression on global variables) and an action (a statement that can modify the variable values).

Finally, to support the universal planning problem specification, the CMurphi input language has been extended to include the goal construct, used to define the properties that a goal state must satisfy.

4.1 Model Generation from PDDL+

In order to automate the universal planning on PDDL+ problems, an important issue is to provide automatic or semi-automatic methods which allow one to use model checking based planners *directly* on planning problems defined in such language.

To this aim, we designed and implemented the *PDDL+ to UPMurphi compiler*. In the following, we describe how the most interesting PDDL+ constructs are translated in the CMurphi description language.

Predicates and Functions. Each PDDL+ predicate pred is mapped on a global boolean variable pred_value which can be modified through appropriate get/set functions.

PDDL+ functions are mapped to bounded numeric variables. Bounds are defined through constants, as shown in Fig. 3.

PDDL+	UPMurphi
(**define** (*domain* car)	**const**
(:**functions**	acc_LB: 0;
(acc) ...)	acc_UB: 100;
(**define** (*problem* car_p)	**var** acc : acc_LB..acc_UB;
(:*domain* car)	**externfun** update_acc(
(:**init** (acc 0)...))	int_type):int_type "lib.h";

Figure 3: Mapping of PDDL+ functions

Since model checking works on finite state systems, the translation process needs to *discretize* and *bound* all these variables, casting them to *finite-precision real numbers*. Finding a suitable discretization is an important issue, since it can affect the plan generation speed, the precision of the solution and, sometimes, its correctness.

It is possible to prove that, given an error threshold, there always exists a discretization that generates correct solutions with an approximation error below that threshold. However, this issue will not be further discussed here, for sake of brevity. By default, in the tool, continuous variables are rounded up to the second decimal, whereas the time is rounded to the first decimal: our experiments proved that these values are usually "safe", generating errors below 2% (see the case study for details on how this error is computed). Anyway, the domain expert can interactively modify these settings, even for single variables, during the PDDL+ translation process.

During the universal plan generation, the tool computes an error estimation and presents it to the user at specific intervals, so he/she can quickly know if the discretization parameters are wrong and possibly stop the computation to adjust them.

Note that external C++ libraries (e.g., `lib.h` in Fig. 3) can be used to easily describe complex update functions.

Actions. PDDL+ actions represent discrete components of the control, and are thus mapped on CMurphi rules, where the action precondition becomes the rule guard and the action effect is copied in the rule update code. If the action has no parameters the result is a simple rule as in the example of Fig. 4.

PDDL+	UPMurphi

(:**action** accelerate
 :**parameters**()
 :**precondition** (running)
 :**effect** (**and** (running)
  (*increase* a 1)))
	rule"accelerate" (running
	 ()) ==>
	begin
	 a := a + 1 ;
	 set_running(**true**);
	end;

Figure 4: Mapping of a PDDL+ action

On the other hand, parametric actions are translated into a family of rules, one for each possible parameter value, using the CMurphi `ruleset` construct.

Events and Processes. Events are used to model instantaneous changes, not necessarily as a direct consequence of the actions, and are particularly interesting when used to initiate processes, which in turn model continuous changes of the system.

PDDL+	UPMurphi

(:**action** a
 :**parameters**()
 :**precondition** (running)
 :**effect** (**and**
  (p1_go) (p2_go)
  (*increase* a 1)))
(:**process** p1
 :**parameters** ()
 :**precondition** (p1_go)
 :**effect** (*increase* a 1))
(:**process** p2
 :**parameters** ()
 :**precondition** (p2_go)
 :**effect** (*increase* a 2))
	procedure p1();
	begin
	 if (p1_go()) **then**
	  a_value := a_value + 1;
	endif; **end**;
	procedure p2();
	begin
	 if (p2_go()) **then**
	  a_value := a_value + 2;
	endif; **end**;
	procedure proc_exec();
	begin
	 p1(); p2();
	end;
	rule "modify_a" running()
	 ==>
	begin
	 proc_exec();
	  a_value := a_value + 1;
	end;

Figure 5: Mapping of a PDDL+ process

Processes are mapped on *procedures* that encode their behavior and are invoked *by each rule* of the model. Indeed processes, once started, are executed in parallel with other actions and events. An example of this mapping is shown in Fig. 5. Note that, for sake of clarity, we use a single procedure `proc_exec()` to map all active processes.

PDDL+	UPMurphi

(:**process** moving
 :**precondition** (running)
 :**effect** (**and**
(*increase* (d)(∗ #t (v)))
(*increase* (v) (∗ #t (a)))
 ))
	const
	 T: 0.1 -- *user defined*
	procedure moving();
	begin
	 if(running()) **then**
	  d := update_d(d,v);
	  v := update_v(v,a);
	endif; **end**;

Figure 6: Mapping of PDDL+ process with continuous update expressions

In PDDL+, continuous update expressions are restricted to occur only in process effects. Continuous effects are represented by update expressions that refer to the special variable $\#t$. In CMurphi this effect is achieved through a global constant `T` which defines the sampling time. `T` is used in the update functions, as shown in Fig. 6, where `T=0.1` seconds and `update_t`, `update_v` compute next values of `t` and `v`, respectively, after a time `T`.

Durative Actions. PDDL+ offers an alternative to the continuous durative action model of PDDL 2.1, adding a more flexible and robust model of time-dependent behavior (Fox and Long 2006). Basically, PDDL+ introduces a three-part structure for modelling periods of continuous change: the *start-process-stop* model. An action or event *starts* a period of continuous change on a numeric variable expressed by means of a *process*. An action or event finally *stops* the execution of that process and terminates its effect on the numeric variable.

Having previously described the mapping of actions, events and processes, the start-process-stop model can be easily implemented in CMurphi as shown in the example of Fig. 7. Note that we use a special variable `obs_clk` in order to count the number of sampling times passed from the beginning of the action.

Problem. In PDDL+ a problem defines an instance of the domain, represented by an initial condition and a goal definition. These two elements simply become the *startstate* and the *goal* in the CMurphi model, respectively. An example is shown in Fig. 8. Of course, in universal planning problems no initial condition is given, so the start states should be selected as described in Sec. 3.2Moreover, note that UPMurphi currently supports only the $minimize(total - time)$ metric, which is however very common in (universal) planning problems.

State Mapping. Finally, model checking requires an explicit representation of the state in terms of *state variables*. To this end, it simply suffices to consider the set of variables deriving from the translation of the predicates and functions defined in the PDDL+ domain, as shown in the correspond-

PDDL+

```
(:durative-action
observe
 :parameters ()
 :duration
  (= ?duration (
   durTime) )
 :condition (and
  (at start (ready))
  (over all (day)) )
 :effect (and
  (at start (not (
   ready)))
  (at end (ready))
  (at end (
   obs_complete)) ))
```

UPMurphi

```
const
 T: 0.1 -- user defined
procedure process_observe();
begin
 if (obs_clk_started) then
  obs_clk := obs_clk + T;
endif; end;
function event_obs_fail() :
 boolean;
var temp:boolean;
begin
 if ((obs_clk_started) &
 (obs_clk != durTime) &
 !(get_day())) then
  durTime := durTime + T;
  return true;
 endif;
 return false;
end;
rule"obs_start" (ready()) &
 (!obs_clk_started) ==>
begin
 set_ready(false);
 obs_clk := obs_clk + T ;
 obs_clk := 0 ;
 obs_clk_started := true ;
 proc_exec();
end;
rule"obs_end" (obs_clk=
 durTime
 & obs_clk_started) ==>
begin
 set_obs_complete(true);
 set_ready(true);
 obs_clk := obs_clk - 1 ;
 obs_clk_started := false ;
 proc_exec();
end;
```

Figure 7: Mapping of a PDDL+ durative-action

PDDL+

```
(define (problem
 car_problem)
(:domain car)
(:init(not(engineBlown))
 (running)
 (= d 0)
 (= a 0)
 (= v 0))
(:goal (>= d 20))
(:metric minimize
 (total-time)))
```

UPMurphi

```
startstate "startstate"
begin
 v := 0 ;
 a := 0 ;
 d := 0 ;
 set_engineblown(false);
 set_running(true);
end;
goal ((d >= 20 ));
```

Figure 8: Mapping of a PDDL+ problem

ing sections above.

5. Case Study: The Planetary Lander

In this section we present a motivating case study for our model checking based universal planning technology. The problem is inspired by the specifications of the "Beagle 2" Mars probe (Blake and others 2004), designed to operate on the Mars surface with tight resource constraints. In particular, we use the PDDL+ domain presented in (Fox and Long 2006), based on a simplified model of a solar-powered lander, the *Planetary Lander*.

Basically, the lander must perform two observation actions, called *Observe1* and *Observe2*. However, before making each observation, it must perform the corresponding preparation task, called *prepareObs1* and *prepareObs2*, respectively. Alternatively, the probe may choose to perform a cumulative preparation task for both observations by executing the single long action *fullPrepare*. The shorter actions have higher power requirements than the single preparation action.

The power needed to perform these operations comes from the probe solar panels. The energy generated by the panels (through the *generating* process) is influenced by the position of the sun, i.e., it is zero at night, rises until midday and then returns to zero at dusk. Power coming from the solar panels is also used to charge a battery (the *charging* process), which is then discharged to give power to the lander (the *discharging* process) when the panels do not produce enough energy (e.g., at night). Moreover, the probe must always ensure a minimum battery level to keep its instruments warm.

The state of charge of the battery is therefore an important variable to monitor. Unfortunately, it follows a complex curve, since the charge/discharge process is nonlinear, and has several discontinuities, caused by the initiation and termination of the actions. Indeed, Table 1 shows the set of ordinary differential equations that are used to recalculate the values of the state variables *soc* (state of charge) and *supply* (solar panel generation). The symbols used in the equations have the following meaning: $s = soc$, $h = supply$, $d = demand$, $r = charge\_rate$, $sc = solar\_const$ and $D = daytime$. The equations clearly show the nonlinear dynamics of the system.

Name	ODE
charging	$\frac{ds(t)}{dt} = [h(t) - d(t)] \cdot r \cdot (100 - s(t))$
discharging	$\frac{ds(t)}{dt} = -[d(t) - h(t)]$
generating	$\frac{dh(t)}{dt} = [sc \cdot D(t)] \cdot$ $\cdot [(D(t) \cdot ((4 \cdot D(t)) - 90))] + 450$

Table 1: PDDL+ events and processes with associated ordinary differential equations

Obviously, the problem here is to find the best correct sequence of actions to achieve the probe goal in the shortest time possible, starting from any reasonable initial configuration. For sake of brevity, here we do not show the PDDL+ problem domain, which can be read in (Fox and Long 2006).

5.1 Model Generation

The mapping from PDDL+ was performed using the default approximations described in Sec. 4.1.

The start state cloud (see Sec. 3) for the universal planning algorithm was selected by taking into account a set of reasonable configurations of the state variables soc and $daytime$. Note that it is realistic to consider *only* these parameters, since they define the environmental conditions to which the lander will be subject at the beginning of its mission. All the other domain parameters were fixed to the values inferred by looking at (Blake and others 2004).

In particular, we suppose that the rover landing hour may be between 0 and 8, that corresponds to the central daylight hours in Martian time (the rover is supposed to land in this range of hours, since they offer the best possible starting conditions). On the other hand, since the battery is not used before landing, and its self-discharge rate is minimal, we can safely suppose that the initial battery state of charge will be between 90% and 100% with steps of 1%. Therefore, the start state cloud will be defined as the set $\{(s,d)|s \in [90\%, 100\%] \wedge d \in [0,8]\}$.

The complete case study details, including the UPMurphi code generated from the PDDL+ source, can be found on the UPMurphi web site (UPMurphi web page 2009), together with more complete experiment results.

5.2 Universal Planning

Given the domain variables and their ranges, we can easily calculate the state space size of the system, which is about of 10^{24} states. Thanks to the reachability analysis, UPMurphi generated an optimal solution for the universal planning problem, starting from the given start state cloud, and visiting only 31 million reachable states, in less than 40 minutes on a 2.2GHz CPU with 2 GB of RAM.

State space size	10^{24}
Search depth limit	200 BFS levels
First goal reached after	174 BFS levels
Reachable states	$31,965,220$
Start states to goal	100%
States to goal (generated plans)	$5,309,514$
Forward analysis time	$1,969.3$ seconds
Plan generation time	296.51 seconds
Total synthesis time	$2,265.81$ seconds
Peak memory requirements	1800MB
Universal plan size	28,6MB

Table 2: Universal Plan generation statistics

The synthesis statistics are in Table 2. Note that the first goal was found after 174 steps, but the synthesis was performed up to the fixed horizon of 200 steps, which is a reasonable upper bound for the lander activity completion (it represents about two Martian days).

The generated solution, which requires less than 30 MB of memory to be stored and used, contains more than 5 million plans, and thus it is able to bring to the goal more than 16% of the reachable states. Due to the exhaustive search performed by the tool, we can safely assert that, in the remaining 84% of the states, the lander could not complete its tasks and should therefore quit its mission or delay its initiation.

	Min	Max	Avg
soc	0 %	0.625,392 %	0.179,329 %
supply	0 %	2.060,061 %	0.742,575 %

Table 3: Normalized root mean squared error for variables *soc* and *supply*

In order to ensure the correctness of the solution, UPMurphi was instructed to check the validity of the generated plans using the VAL plan validator (Howey, Long, and Fox 2004).

In addition, to estimate the precision of the plans (and thus the error introduced by the model discretization), the tool compared the variable values computed by VAL during the validation process with the corresponding values output by its plan synthesis procedure, by computing the normalized root mean square error (NRMSE), as shown in Table 3. The NRMSE is at most 2% in all the generated plans for the nonlinear variable *soc*, at most 0.6% for nonlinear variable *supply* and always zero for the linear variable *daytime* (not shown in the table). Nevertheless, the average NRMSE is small: 0.179% for *soc* and 0.742% for *supply*, respectively.

Therefore, the discretization process, that is part of the model checking based universal planning process, produces an acceptable approximation error in the plans generated by UPMurphi. In particular, the error is small enough to not influence the correct plan execution, as proved by the plan validation performed by VAL.

6. Conclusions

The UPMurphi tool presented in this paper uses a model checking based algorithm to perform universal planning on domains that are completely out of reach for all the current universal planning tools. These domains include many realistic systems, like hybrid or nonlinear systems. Indeed, the presented case study shows that UPMurphi can be an effective and valuable universal planner. Moreover, the tool is able to directly read specifications written using the full power of PDDL+, thus being very friendly to the planning community.

UPMurphi is being actually refined and will be shortly released in a beta version. Further development will be mainly addressed to increase the precision of the tool calculations, either through the automatic selection of a suitable discretization for each given domain, or by embedding in the tool an enhanced equation solver that would allow a very fine discretization to be used on all systems.

References

Aylett, R.; Soutter, J. K.; Petley, G. J.; and Chung, P. W. H. 1998. AI planning in a chemical plant domain. In *ECAI*, 622–626.

Behrmann, G.; Cougnard, A.; David, A.; Fleury, E.; Larsen, K. G.; and Lime, D. 2007. UPPAAL-TIGA: Time for playing games! In *CAV*, 121–125.

Blake, O., et al. 2004. *Beagle2 Mars: Mission Report.* Lander Operations Control Centre, National Space Centre, University of Leicester.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.* 98(2):142–170.

Cached Murphi web page. 2006. `http://www.dsi.uniroma1.it/~tronci/cached.murphi.html`.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998a. Strong planning in non-deterministic domains via model checking. In *AIPS*, 36–43.

Cimatti, R.; Roveri, M.; and Traverso, P. 1998b. Automatic OBDD-based generation of universal plans in non-deterministic domains. 875–881. AAAI Press.

Della Penna, G.; Intrigila, B.; Melatti, I.; Minichino, M.; Ciancamerla, E.; Parisse, A.; Tronci, E.; and Venturini Zilli, M. 2003. Automatic verification of a turbogas control system with the murφ verifier. In *Proceedings of HSCC 2003*, volume 2623 of *Lecture Notes in Computer Science*, 141–155. Springer.

Della Penna, G.; Intrigila, B.; Melatti, I.; Tronci, E.; and Venturini Zilli, M. 2004. Exploiting transition locality in automatic verification of finite state concurrent systems. *STTT* 6(4):320–341.

Edelkamp, S.; Lafuente, A. L.; and Leue, S. 2001. Directed explicit model checking with hsf-spin. In *Proceedings of SPIN '01*, 57–79. Springer-Verlag.

Edelkamp, S. 2001. First solutions to PDDL+ planning problems. In *PlanSIG Workshop*, 75–88.

Edelkamp, S. 2002. Taming numbers and durations in the model checking integrated planning system. *J. Artif. Intell. Res. (JAIR), special issue on the 3rd International Planning Competition*.

Edelkamp, S. 2003. PDDL2.2 planning in the model checking integrated environment. In *UK Planning and Scheduling Special Interest Group (PlanSig), Glasgow*.

Fox, M., and Long, D. 2001. PDDL+: An extension to PDDL 2.1 for modelling planning domains with continuous time-dependent effects. Technical report, Dept. of Computer Science, University of Durham.

Fox, M., and Long, D. 2002. Fast temporal planning in a graphplan framework. In *Proceedings of AIPS Workshop on Planning for Temporal Domains*, 9–17.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res. (JAIR)* 27:235–297.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. *ICTAI* 00:294–301.

Ip, C. N., and Dill, D. L. 1993. Better verification through symmetry. In *Proceedings of CHDL '93*, 97–111. North-Holland.

Jensen, R. M., and Veloso, M. M. 2000. OBDD-based universal planning for synchronized agents in non-deterministic domains. *J. Artif. Intell. Res. (JAIR)* 13:13–189.

Léauté, T., and Williams, B. C. 2005. Coordinating agile systems through the model-based execution of temporal plans. In *AAAI*, 114–120.

Li, H. X., and Williams, B. C. 2008. Generative planning for hybrid systems based on flow tubes. In *ICAPS*, 206–213.

Murphi web page. 2004. `http://sprout.stanford.edu/dill/murphi.html`.

Penberthy, J. S., and Weld, D. S. 1994. Temporal planning with continuous change. In *Proceedings of AAAI'94*, volume 2, 1010–1015. American Assoc. for Artif. Intell.

Schmid, U., and Wysotzki, F. 2000. Applying inductive program synthesis to macro learning. In *Proceedings of AIPS 2000*, 371–378.

Schmid, U. 2003. *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Springer-Verlag.

Schoppers, M. 1987. Universal plans of reactive robots in unpredictable environments. In *Proceedings of IJCAI, 1987*.

Schuster, H. G. 1988. *Deterministic Chaos: An Introduction*. Weinheim Physik.

Shin, J.-A., and Davis, E. 2005. Processes and continuous change in a SAT-based planner. *Artif. Intell.* 166(1-2):194–253.

Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI-99*, 326–337.

Sontag, E. D. 1995. Interconnected automata and linear systems: A theoretical framework in discrete-time. In *Hybrid Systems*, 436–448.

Stern, U., and Dill, D. L. 1995. Improved probabilistic verification by hash compaction. In *Proceedings of CHARME '95*, 206–224. Springer-Verlag.

T. Henzinger. 1996. The theory of hybrid automata. In *Proceedings of the 11th Symposium on Logic in Computer Science*, 278–292. IEEE Press.

The AIPS-98 Planning Competition Committee. 1998. PDDL: the planning domain definition language. Technical report. `www.cs.yale.edu/homes/dvm`.

UPMurphi web page. 2009. `http://www.di.univaq.it/magazzeni/upmurphi.php`.

UPPAAL-TIGA web page. 2009. `http://www.cs.aau.dk/~adavid/tiga/`.

Wolfman, S. A., and Weld, D. S. 1999. The LPSAT engine and its application to resource planning. In *Proc. IJCAI*, 310–316.