International Journal on Artificial Intelligence Tools
Vol. 24, No. 6 (2015) 1550025 (45 pages)

DOI: 10.1142/S0218213015500256



# Synthesis of Cost-Optimal Strong Plans in Non-Deterministic Domains

Giuseppe Della Penna
*Department of Information Engineering, Computer Science and Mathematics*
*University of L'Aquila, Via Vetoio, 67100 Coppito, L'Aquila, Italy*
*giuseppe.dellapenna@univaq.it*

Bedenetto Intrigila
*Department of Enterprise Engineering, University of Rome Tor Vergata*
*Rome, Italy*
*intrigil@axp.mat.uniroma2.it*

Daniele Magazzeni
*Department of Informatics, King's College London*
*Strand, London WC2R 2LS, UK*
*daniele.magazzeni@kcl.ac.uk*

Fabio Mercorio
*Department of Statistics and Quantitative Methods — C.R.I.S.P. Research Centre*
*University of Milan-Bicocca, Milan, Italy*
*fabio.mercorio@unimib.it*

Received 25 September 2013
Accepted 7 April 2015
Published 21 December 2015



Automatic planning tools can be a valuable aid to build control systems for a wide range of everyday appliances. Many systems which interact with the real world present a non-deterministic behaviour, often made more complex by nonlinear continuous dynamics. In this context, strong planning, which requires finding a plan that is guaranteed to achieve the goal regardless of non-determinism, plays an important role. With the increasing need for optimising the use of resources, finding strong solutions while minimising a cost function appears a significant research challenge, which has not previously been addressed.

In this paper we provide a formal description of the cost-optimal strong planning problem and present an algorithm to solve it with good complexity bounds. The algorithm correctness and completeness are formally proved, and its implementation in the disk-based SUPMurphi tool is described. Finally, two meaningful case studies are presented to show the effectiveness of the proposed approach compared with a state-of-the-art strong planning tool.

*Keywords*: Cost-optimal strong planning; non-deterministic systems; explicit model checking.

## 1. Introduction

In the last two decades, the use of sophisticated control systems and intelligent planning algorithms is growing apace, having a great impact in many industrial fields such as robotics and manufacturing processes. Many common processes take place in an environment having variable and unpredictable influences on the system dynamics, which need to be taken into account to design a correct and efficient control system: dealing with such *non-deterministic domains* is a very significant concern in this field.

In particular, a non-deterministic domain models a particular form of uncertainty where the system actions may have different outcomes. For each action, the set of possible outcomes are known (for example, the range of possible disturbances for an actuator, or the noise affecting a sensor). However, at run-time, it is not possible to predict which one of the expected outcomes will be actually obtained. Such a situation requires an extended definition of plans, where the uncertainty arising from non-determinism is taken into account at different levels.

As introduced by Cimatti *et al.*,[21] three kinds of plans are of interest when dealing with non-deterministic domains:

- a *Weak Plan* is a plan that *may* achieve the goal, but no guarantees are given about its success. More precisely, at least one of the many non-deterministic plan executions reaches the goal;
- a *Strong Plan* is a plan that is guaranteed to achieve the goal regardless of non-determinism. In other words, any plan execution always leads to a goal;
- a *Strong Cyclic Plan* is a plan that achieves the goal under the *fairness* assumption. At execution time, the plan applies an action infinitely often until the "lucky" outcome happens (i.e., the action's outcome that leads the system towards the goal). If it does not happen then the execution is called *unfair*.

Intuitively, weak solutions represent *optimistic* plans, i.e., we hope that the plan execution will reach the goal. On the contrary, strong solutions represent a *warranty* about the success of the plan, in any execution. Finally, strong cyclic solutions, which are based on a trial-and-error strategy, represent an intermediate alternative between strong and weak ones. In this paper we will focus on the concept of strong plan. Indeed, as mentioned above, in many real cases we need to cope with disturbances or noise, which affect the result of the actions. In such cases, we need a *strong* plan, when we want to guarantee that the goal will be reached notwithstanding the presence of disturbances in the execution of actions and/or of noise in the detection of signals.

A milestone in this area is represented by the MBP system,[66] which allows one to find weak, strong and strong cyclic plans.

To the best of our knowledge, existing approaches (including MBP) for finding weak, strong, and strong-cyclic plans do not take into account the plan cost, i.e., there is no strategy which considers the plan optimisation w.r.t. a cost function. This is mainly due to complexity problems: indeed, finding plan in a non-deterministic domain is a very complex task by itself, thus optimising such plan makes the problem even harder: indeed, roughly, the planning problem (without non-determinism) can be considered analogous to the shortest

path problem, where it is known that the introduction of weights adds at least a $\log C$ factor to the complexity, where $C$ is the maximum arc cost.[1] However, this aspect is often useful in real-world applications, where reaching the goal is a requirement, but reaching it with less resources is an advantage.

Another important aspect about planning (and then also for planning in non-deterministic domains) is the observability of the system variables, which can be partial,[6,7,47,10,2] or full. In this paper we restrict to full observability, where the values of all variables are known. Thus, plans can be executed by reactive controllers that iteratively read such variables, determines the current state of the system, and then select and execute an appropriate action.[51]



### 1.1. *Contribution*

The contribution of this paper goes in three directions.

- The main contribution is an algorithm to solve the *cost-optimal* strong planning problem in non-deterministic domains. Our algorithm has the work of Cimatti *et al.*[22] as starting point, however it presents the unique feature of considering a *cost function* and *optimising* the strong plan w.r.t. such function, still preserving a good complexity bound. To the best of our knowledge, this is the first approach to cost-optimal strong planning and no better solutions for this problem have been devised so far, even in other computer science fields.
- Second, we present a prototypal implementation of our algorithm in the SUPMurphi tool (built on top of the UPMurphi system[28,33,a]), which uses a disk-based approach in order to mitigate the so called *state-explosion* problem which affects all the explicit approaches. In particular, the framework is based on an *explicit* model checking approach, rather than a symbolic one (as in the case of the MBP system), so extending the class of problems on which strong planning can be applied to hybrid and/or nonlinear continuous domains, which are actually very common in practice. Indeed, the SUPMurphi system inherits from UPMurphi the *discretise and validate* approach,[28] that allows dealing with continuous domains through discretisation.
- Third, we show how SUPMurphi can be used to solve the cost-optimal strong *universal* planning problem, and we present its application on two case studies. In this kind of problems, we are interested in finding a cost-optimal strong plan for *every state of the system which can be reached from the initial states and has a strong plan to the goal*. The cost-optimal strong universal planning is actually also interesting since it is very similar to the (optimal) control problem, where the controller should try to bring the system to the goal (setpoint) regardless of some external disturbances and from any state that the system can reach. Also in this case, although an effective strong universal planning algorithm has been already proposed,[23] it cannot deal with costs on actions.

The paper is structured as follows. Next section introduces the cost-optimal strong universal planning problem together with all the required background notions. Then,

[a]The UPMurphi tool can be downloaded at http://www.di.univaq.it/gdellape/content.php?page=upmurphi

Section 3 describes an algorithm to solve this problem and proves its correctness. Section 4 introduces the SUPMurphi tool and shows how the disk based approach has been implemented. Then, Section 5 presents two non-deterministic planning problems: the *omelette* domain and the *inverted pendulum on a cart* domain. We use the former to compare our performance with the well-known MBP planner, and the latter to show how the approach can be applied to a continuous system using the discretise and validate technique. Finally, Section 6 provides an overview of the related work and Section 7 outlines some concluding remarks.

## 2. Statement of the Problem

In this section we first introduce some background notions about non-deterministic systems and non-deterministic planning, then we define the concept of cost-optimal strong plan and the corresponding planning problem.

### 2.1. *Non-deterministic finite state systems*

We now introduce the formal definition of the non-deterministic systems we are interested in.

**Definition 2.1.** A *Non-Deterministic Finite State System* (*NDFSS*) $\mathcal{S}$ is a 4-tuple $(S, s_0, \mathcal{A}, F)$, where: $S$ is a finite set of *states*, $s_0 \in S$ is the *initial state*, $\mathcal{A}$ is a finite set of *actions* and $F : S \times \mathcal{A} \to 2^S$ is the *non-deterministic transition* function, that is $F(s,a)$ returns the set of states that can be reached from state $s$ via action $a$.

It is worth noting that we are restricting our attention to NDFSS having a single initial state $s_0$ only for the sake of simplicity. Indeed, if we are given a NDFSS $\mathcal{S}'$ with a set of initial states $I \subseteq S$, we may simply turn it into an equivalent NDFSS by adding a dummy initial state connected to each the state in $I$ by a deterministic transition with fixed cost (see the definitions below). The algorithm given in Section 3 could then be trivially adapted to return the set of cost-optimal strong plans for $I$.

The non-deterministic transition function implicitly defines a set of *non-deterministic transitions* between states which, in turn, give raise to a set of *trajectories*, as specified in the following definitions.

**Definition 2.2.** Let $\mathcal{S} = (S, s_0, \mathcal{A}, F)$ be an *NDFSS*. A *non-deterministic transition* $\tau$ is a triple of the form $(s, a, F(s,a))$ where $s \in S$ and $a \in \mathcal{A}$. A *deterministic transition* (or simply a *transition*) $\tau$ is a triple of the form $(s, a, s')$ where $s, s' \in S$, $a \in \mathcal{A}$ and $s' \in F(s,a)$. We say that $\tau = (s, a, s')$ *is in* $(s, a, F(s,a))$ if $s' \in F(s,a)$. We denote with $\mathcal{S}_\tau$ the set of all the transitions in $\mathcal{S}$.

**Definition 2.3.** A *trajectory* $\pi$ from a state $s$ to a state $s'$ is a sequence of transitions $\tau_0, \ldots, \tau_n$ such that $\tau_0$ has the form $(s, a, s_1)$, for some $s_1$ and some $a$, $\tau_n$ has the form $(s_n, a', s')$, for some $s_n$ and some $a'$ and for every $i = 0, \ldots, n-1$ if $\tau_i = (s_i, a_i, s_{i+1})$, for some $s_i, a_i, s_{i+1}$, then $\tau_{i+1} = (s_{i+1}, a_{i+1}, s_{i+2})$, for some $s_{i+2}, a_{i+1}$. We denote with $|\pi|$ the length of $\pi$, given by the number of transitions in the trajectory.

As usual, we consider an empty set of transitions as a trajectory from any state to itself.

We also need to introduce several notions related to non-deterministic transitions and trajectories.

**Definition 2.4.** Let $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ be an *NDFSS* and $\Pi$ be a set of non-deterministic transitions.

- We say that a transition $\tau = (s, a, s')$ is *extracted from* $\Pi$ if $(s, a, F(s,a)) \in \Pi$ and $(s, a, s')$ is in $(s, a, F(s,a))$.
- Similarly, we say that a trajectory $\pi = \tau_0, \ldots, \tau_n$ is *extracted from* $\Pi$ if $\forall i \in \{0 \ldots n\}, \tau_i = (s_i, a_i, s_i')$ is in $(s_i, a_i, F(s_i, a_i))$ and $(s_i, a_i, F(s_i, a_i)) \in \Pi$.
- Finally, we say that a state $s$ *is in* $\Pi$ if there exists a transition $\tau = (s, a, s')$ extracted from $\Pi$.

Finally, since we are interested in cost-optimal solutions, we need to formally define the concept of cost function.

**Definition 2.5.** Let $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ be an *NDFSS*. A *cost function* (also called *weight function*) for $\mathcal{S}$ is a function $\mathcal{W} : \mathcal{S}_\tau \to \mathbf{R}_+ \setminus \{0\}$ that assigns a cost to each transition in $\mathcal{S}$.

Using the cost function for (deterministic) transitions defined above, we define the cost of the non-deterministic transition $(s, a, F(s,a))$, denoted by $\mathcal{W}(s,a)$, as follows:

$$\mathcal{W}(s,a) = \max_{s' \in F(s,a)} \mathcal{W}((s,a,s'))$$

It is worth noting that, for the sake of generality, the definition of the cost function allows the transition cost to depend on both the corresponding action and the source state.

### 2.2. *Cost-optimal strong planning problem*

In order to define the cost-optimal strong planning problem, we assume that a non-empty set of *goal states* has been specified for the given system.

**Definition 2.6.** Let $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ be an *NDFSS*. Then a *Cost-Optimal Strong Planning Problem (COSPP)* is a triple $\mathcal{P} = (\mathcal{S}, \mathcal{W}, G)$ where $G$ is the set of the goal states and $\mathcal{W} : \mathcal{S}_\tau \to \mathbf{R}_+ \setminus \{0\}$ is the cost function associated to $\mathcal{S}$.

A solution to the problem above is a *cost-optimal strong plan* from the initial state $s_0$ to $G$, i.e., a sequence of actions that, starting from $s_0$, leads the system to the goal states, regardless of the non-deterministic outcome of each action, and such that all the other strong plans from $s_0$ to $G$ have greater or equal weight.

Before formally describing such a solution, we need the following auxiliary definition.

**Definition 2.7.** Let $\mathcal{P} = \{\{S, s_0, \mathcal{A}, F\}, \mathcal{W}, G\}$ be a *COSPP* and $s \in S$. A *deterministic plan* $p$ from $s$ to a goal $g \in G$ is a trajectory $\pi$ such that either

- $s \in G$ and $|\pi| = 0$;
- or $\pi = \tau_0, \tau_1, \ldots, \tau_n$, with $\tau_0 = (s, a, s_1)$ and $\pi' = \tau_1, \ldots, \tau_n$ is a *deterministic plan* from $s_1$ to $g$.

For sake of simplicity, let us first give the general definition of a strong plan, without any cost optimisation.

**Definition 2.8.** Let $\mathcal{P} = \{\{S, s_0, \mathcal{A}, F\}, \mathcal{W}, G\}$ be a *COSPP*. Let $s$ be a state in $S$. A *strong plan* from $s$ to $G$ is a set $P$ of non-deterministic transitions such that either $s \in G$ and $P = \emptyset$ or $s \notin G$ and $P$ satisfies the following conditions:

(1) there exists a natural number $n_0$ such that every trajectory $\pi$ that can be extracted from P has length $|\pi| \leq n_0$ (i.e., all the trajectories extracted from P are finite);
(2) every trajectory $\pi$ starting from $s$ which can be extracted from $P$, can be extended to a deterministic plan $\pi'$ from $s$ to a goal $s_g \in G$ such that $\pi'$ is extracted from $P$ (i.e., every trajectory in P ends in a goal state);
(3) for every state $s'$ such that $s' \notin G$ and $s'$ is in $P$ there exists a trajectory $\pi$, extracted from $P$, starting from $s$ and ending in $s'$ (i.e., every state in the strong plan P is reachable from the initial state $s$ following a trajectory which can be extracted from P);
(4) for every state $s'$ such that $s' \notin G$ and $s'$ is in $P$, there exists exactly one non-deterministic transition in $P$ of the form $(s', a, F(s', a))$, for some $a \in \mathcal{A}$. We denote with $P(s')$ such non-deterministic transition (i.e., each state in the strong plan has an unique associated action).

From the definition above, we can derive the following characterisation of a strong plan.

**Proposition 2.1.** *Let $\mathcal{P} = \{S, \mathcal{W}, G\}$ be a* COSPP. *Let $s$ be a state in $S$. Then, $P$ is a strong plan from $s$ to $G$ iff $P$ is a set of non-deterministic transitions such that either $s \in G$ and $P = \emptyset$ or $s \notin G$ and there exists a unique non-deterministic transition in $P$ of the form $\tau = (s, a, F(s, a))$, for some $a \in \mathcal{A}$, such that either*

- *$F(s, a) \subseteq G$;*
- *or $P \setminus \{(s, a, F(s, a))\}$ is the union of strong plans $P_i$ from every state $s_i$ in $F(s, a)$ to $G$.*

**Proof.** Assume first that $P$ is a strong plan from $s$ to $G$. If $P$ is not empty, then there exists a unique non-deterministic transition $\tau = (s, a, F(s, a)) \in P$, for some $a$ (using assumption 4 of Definition 2.8). Let $s_i$ be an element of $F(s, a)$. We define $P_i$ as the set of non-deterministic transitions in $P$ such that $P_i$ contains some transitions of a deterministic plan from $s_i$.

Now observe that any sequence, extracted from $P$, starting from $s_i$ can be completed in $P$ itself to a deterministic plan (using assumption 2 of Definition 2.8) without using the node $s$. Indeed, no deterministic plan extracted from $P$ can return to the node $s$, since otherwise there would be a cycle, contradicting the requirement that every deterministic sequence in $P$ is bounded (using assumption 1 of Definition 2.8). It follows that $P_i$ is a subset of $P \setminus \{\tau\}$ and is a strong plan from $s_i$.

Moreover, let $s'$ be any node in $P \setminus \{\tau\}$. Then there exists a trajectory $\pi$ from $s$ to $s'$ (using assumption 3 of Definition 2.8). By the uniqueness of $\tau$, the first transition of $\pi$ is

Table 1. Flight scheduling.

| From | To | Flight # | Depart | Arrive |
|---|---|---|---|---|
| Rome-FCO | Paris-CDG | A | 08.00 | 09.00 |
| Rome-FCO | Berlin-BER | E | 08.00 | 10.00 |
| Rome-CIA | Amsterdam-AMS | D | 05.00 | 08.00 |
| Paris-CDG | San Francisco-SFO | B | 10.00 | 12.00 (GMT-7) |
| Paris-CDG | San Francisco-SFO | C | 19.00 | 21.00 (GMT-7) |
| Berlin-BER | San Francisco-SFO | F | 11.00 | 14.00 (GMT-7) |
| Berlin-BER | Amsterdam-AMS | I | 12.00 | 13.00 |
| Berlin-BER | San Francisco-SFO | G | 12.00 | 15.00 (GMT-7) |
| Amsterdam-AMS | San Francisco-SFO | H | 15.00 | 20.00 (GMT-7) |



in $\tau$ and therefore has the form $(s,a,s_i)$ for some $s_i$, it follows that $s'$ is in $P_i$, and that $P \setminus \{\tau\} = \bigcup_{s_i \in F(s,a)} P_i$.

The other direction can be proved similarly. □

Let us now reintroduce the concept of cost in our solutions. By Proposition 2.1 we can define the *cost of a plan* as follows:

**Definition 2.9.** The cost of a strong plan $P$ from $s$ to $G$, denoted by $\mathcal{W}(P)$, is recursively defined as follows:

- if $P$ is empty then $\mathcal{W}(P) = 0$;
- if $P$ is composed only of the non-deterministic transition $(s,a,F(s,a))$, for some $a$, then $\mathcal{W}(P) = \mathcal{W}(s,a)$;
- if $P$ is composed of the non-deterministic transition $(s,a,F(s,a))$, for some $a$, and of plans $P_i$ from every node $s_i$ in $F(s,a)$ then $\mathcal{W}(P) = \max_{s_i \in F(s,a)} (\mathcal{W}((s,a,s_i)) + \mathcal{W}(P_i))$.

It is easy to see that the cost of a plan $P$ is the maximum cost of a deterministic plan extracted from $P$. Now we can give the following complete definition of a cost-optimal strong plan.

**Definition 2.10.** Let $\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ be a *COSPP*, with $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$. Then a *cost-optimal strong solution* of the *COSPP* $\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ is a strong plan $P$ from $s_0$ to $G$ such that the cost of $P$ is minimal among the strong plans from $s_0$ to $G$.

**Example 2.1.** *The Hurried Passenger Cost-Optimal Strong Planning Problem.*

As an example of COSPP, let us consider the *hurried passenger* problem. A passenger wants to arrive to San Francisco airport (SFO) departing from one of the Rome airports (CIA or FCO) and according to the flight scheduling shown in Table 1. Moreover, there is a bus on every hour that allows the passenger to go from home to one of the Rome airports above in one hour.

The goal is to arrive to San Francisco as soon as possible, and, however, no later than 21.00 local time. We require the passenger to arrive at the airport at least one hour before a flight departure. Moreover, we assume that each flight may arrive at destination later

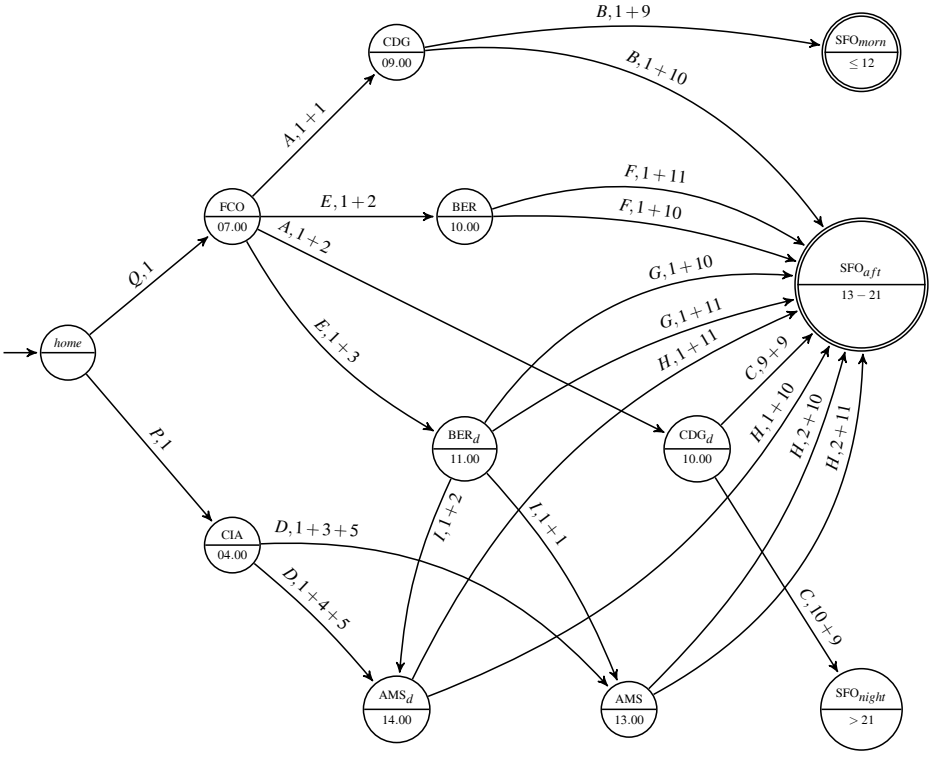


Fig. 1. Graphical description of the COSPP for the *hurried passenger* problem.

than the expected arrival time: this possible delay represents the non-determinism of the problem and, for the sake of simplicity, we assume it to be limited to one hour for each flight. Note that the delays are all equally possible, i.e., each flight has equal possibility to arrive on time or with a 1-hour delay. Therefore, our aim is to generate a strong plan (if any) that guarantees the passenger to reach the San Francisco airport before 21.00 local time regardless of possible flight delays.

A graphical description of the problem is given in Figure 1. Here, tagged nodes (states) represent the arrival time at the corresponding airport, indicated by its acronym (see Table 1). The subscript "d" is used to mark delayed arrivals, e.g., state BER means "arrived at Berlin without delay" whereas BER$_d$ means "arrived ad Berlin with a delay". It is worth noting that, for sake of simplicity, the final states in the diagram are actually macro-states, which include several arrival times: thus, the subscripts "morn", "aft" and "night" mark the three different final "SFO" states where the arrival is, respectively, in the morning ($\leq 12$), in the afternoon ($13-21$) or at night ($>21$). Finally the "home" state represents the initial state, i.e., where the passenger is still at home. Edges (actions) are labelled with the flight code and the cost of each transition, which is $t(d)+t(f)+t(a)$, where $t(d)$ is the time spent at airport $d$ waiting for the flight departure, $t(f)$ is the duration of the flight and $t(a)$

Table 2. COSPP for the *hurried passenger* problem.

| | |
|---|---|
| $S$ | home = $s_0$, AMS = $s_1$, AMS$_d$ = $s_2$, CDG = $s_3$, CDG$_d$ = $s_4$, CIA = $s_5$, FCO = $s_6$, BER = $s_7$, BER$_d$ = $s_8$. SFO$_{aft}$ = $s_9$, SFO$_{morn}$ = $s_{10}$, SFO$_{night}$ = $s_{11}$. |
| $\mathcal{A}$ | A, B, C, D, E, F, G, H, I, P, Q |
| $F$ | $F(s_0,Q)=\{s_6\}$, $F(s_0,P)=\{s_5\}$<br>$F(s_1,H)=\{s_9\}$, $F(s_2,H)=\{s_9\}$<br>$F(s_3,B)=\{s_9,s_{10}\}$ , $F(s_4,C)=\{s_9,s_{11}\}$<br>$F(s_5,D)=\{s_1,s_2\}$<br>$F(s_6,A)=\{s_3,s_4\}$, $F(s_6,E)=\{s_7,s_8\}$<br>$F(s_7,F)=\{s_9\}$, $F(s_8,G)=\{s_9\}$, $F(s_8,I)=\{s_1,s_2\}$ |
| $\mathcal{W}$ | $\mathcal{W}(s_0,Q,s_6)=1, \mathcal{W}(s_0,P,s_5)=1$;<br>$\mathcal{W}(s_1,H,s_9)=12, \mathcal{W}(s_1,H,s_9)=13$;<br>$\mathcal{W}(s_2,H,s_9)=11, \mathcal{W}(s_2,H,s_9)=12$;<br>$\mathcal{W}(s_3,B,s_{10})=10, \mathcal{W}(s_3,B,s_9)=11$;<br>$\mathcal{W}(s_4,C,s_9)=18, \mathcal{W}(s_4,C,s_{11})=19$;<br>$\mathcal{W}(s_5,D,s_1)=9, \mathcal{W}(s_5,D,s_2)=10$ ;<br>$\mathcal{W}(s_6,A,s_3)=2, \mathcal{W}(s_6,A,s_4)=3$,<br>$\mathcal{W}(s_6,E,s_7)=3, \mathcal{W}(s_6,E,s_8)=4$;<br>$\mathcal{W}(s_7,F,s_9)=11, \mathcal{W}(s_7,F,s_9)=12$;<br>$\mathcal{W}(s_8,G,s_9)=11, \mathcal{W}(s_8,G,s_9)=12$,<br>$\mathcal{W}(s_8,I,s_2)=3, \mathcal{W}(s_8,I,s_1)=2$; |
| $G$ | $s_9, s_{10}$ |
| $P$ | $P(s_0)=Q(17); P(s_5)=D(22); P(s_6)=E(16); P(s_7)=F(12)$;<br>$P(s_8)=G(12); P(s_2)=H(12); P(s_1)=H(13), P(s_3)=B(11)$ |

is the time spent at airport $a$ waiting for the next flight (which could be zero, and is omitted in this case). Note that the final macro-states can be reached by more than one edge with the same flight code but different cost: this is indeed correct since they represent more than one state. Finally, the special actions $P$ and $Q$ represent the bus journey from home to Rome-CIA and Rome-FCO, respectively: for the sake of simplicity, we do not consider delays on these actions, so the corresponding transitions are deterministic and have cost 1 (i.e., the bus journey takes an hour).

The corresponding COSPP (according to Definitions 2.1,2.5 and 2.6) is reported in Table 2. Note that, by abuse of notation, in the table we report different costs for some transitions (e.g., $\mathcal{W}(s_1,H,s_9)=12$ and $\mathcal{W}(s_1,H,s_9)=13$) to reflect the different edges leading to a macro-state in Figure 1.

The cost-optimal solution consists in flying from Rome-FCO to Berlin-BER and then to San Francisco-SFO. The total cost of the solution (considering all the possible delays) is 17. Note that another strong solution would be flying from Rome-CIA to Amsterdam-AMS and then to San Francisco-SFO, but its cost is 23. Finally, flying from Rome-FCO to Paris-CDG is not a strong solution since in case of delay of flight A it would be impossible to reach San Francisco on time.

## 3. The Cost-Optimal Strong Planning Algorithm

In this section we describe a procedure that looks for a solution to a given COSPP. The algorithm makes use of some auxiliary functions and data structures, described in the following.

- The *cost vector* (*Cost*) is used to maintain the set of states for which we have synthesised a strong plan, sorted by their cost.

  By abuse of notation, in the following we refer to $Cost(s)$ as the *cost function* which returns the minimum cost of a strong plan from $s$ to the goals calculated so far. The algorithm updates this function every time a better strong plan is found for $s$. Initially all the goal states have a cost equal to zero, while the cost of the other states is set to $\infty$.
- The set of *candidates* (*Cand*) contains the $(s,a)$ pairs corresponding to all the states $s$ which, at any step, are recognised to have a plan starting with action $a$, possibly of *non* minimum cost. By abuse of notation, in the following we also write $s \in Cand$ to denote a state which is in the set of candidates, i.e., $s \in Cand$ is a short-hand for $s|(\exists a|(s,a) \in Cand)$.

  The elements in the set *Cand* can be partially ordered with respect to the cost function *Cost*. Initially the set *Cand* is empty.
- The set of *extended goals* (*ExtGoals*) contains all the states $s$ for which, at any step, a plan $P$ of *minimum cost* has been computed.

  Initially the set *ExtGoals* contains all the goal states in $G$. The auxiliary set of *old extended goals* (*OldExtGoals*) contains, at any step, the extended goals collected up to the previous step: that is, the expression $ExtGoals \setminus OldExtGoals$ represents the states that have been just added to the extended goals.
- The set $\Delta(s,a)$ is initialised, for each state-action pair, with the states reachable from $s$ via action $a$, i.e., $F(s,a)$, which are consumed during the algorithm iterations.

The planning algorithm consists of three procedures described in the following. We assume that all procedures take as input the COSPP $\mathcal{P} = ((S, s_0, \mathcal{A}, F), \mathcal{W}, G)$ as well as all the data structures described above. Note that, in the algorithms, some arithmetic operations (i.e., min, max and sum) may involve infinity. In this case, we assume the usual semantics, e.g., $\max(x, \infty) = \infty$ or $x + \infty = \infty$.

### 3.1. *The* CANDIDATEEXTENSION *routine*

The CANDIDATEEXTENSION routine (Figure 2) extends the set *Cand* of candidates. The function $Pre(s)$ returns all the transitions leading to $s$ and is applied to the extended goals found in the previous iteration of the main algorithm. At any step, the set $\Delta(s,a)$ contains only the states reachable from $s$ via action $a$ which have not been moved to the extended goals yet. Thus, once $\Delta(s,a)$ is empty, $s$ is guaranteed to have a strong plan through action $a$, since all the transitions in $(s,a,F(s,a))$ lead to an extended goal. The pair $(s,a)$ is then added to the set of candidates if it improves the cost currently associated to $s$.

```
1:  for all s' ∈ (ExtGoals \ OldExtGoals) do
2:    Pre(s') ← {(s,a) ∈ S × A | s' ∈ F(s,a)};
3:    for all (s,a) ∈ Pre(s') do
4:      Δ(s,a) ← Δ(s,a) \ {s'};
5:      if Δ(s,a) = ∅ then
6:        c' ← max_{s̄∈F(s,a)} (W(s,a,s̄) + Cost(s̄));
7:        if c' < Cost(s) then
8:          Cand ← Cand ∪ (s,a);
9:          Cost(s) ← c';
10:       end if
11:     end if
12:   end for
13: end for
```

Fig. 2. Procedure CANDIDATEEXTENSION.

```
1: α ← min_{(s,a)∈Cand} Cost(s); // Uses the MINCOSTCAND routine
2: for all (s,a) ∈ Cand | Cost(s) = α do
3:   ExtGoals ← ExtGoals ∪ {s};
4:   Cand ← Cand \ {(s,a)};
5:   SP ← SP ∪ (s,a);
6: end for
```

Fig. 3. Procedure PLANEXTENSION.

### 3.2. *The* PLANEXTENSION *routine*

The effect of the PLANEXTENSION routine (Figure 3) is twofold. First, it selects the states of minimum cost in the candidates set and moves them to the set of extended goals. Indeed, the current solution for such states cannot be improved, since there are no actions which provide a strong solution with a lower cost (as discussed later in Proposition 3.2). Second, it inserts the new extended goals together with the associated action (i.e., the corresponding non-deterministic transition) in the strong plan *SP*.

Note that the extraction of the candidates with the lowest cost (first two lines of Figure 3) can be accomplished with a small complexity if we suppose to have a structure *costvector* where each element $costvector[c]$ holds a (unsorted) list of references to the states with cost $c$. Insertion in this structure requires only to push the new state reference on the top of the list, thus it can be achieved in constant time, whereas updates can be also accomplished in constant time by re-inserting the state with updated cost without removing the previous instance (i.e., creating a duplicate with different cost). Indeed, the states with minimum cost can be extracted from this structure as shown by the MINCOSTCAND routine (Figure 4).

The procedure takes as input the cost of the last states returned, and scans the *costvector* starting from the element corresponding to the next (higher) cost $c$ (for the sake of simplicity, in the pseudocode we suppose it to be $lastc+1$, but in general it depends on the approximation of the cost function). If $costvector[c]$ contains some states that are not yet

```
Input: lastc, the cost of the last states returned
1:  c ← lastc
2:  loop
3:    c ← c + 1
4:    AllCand_c ← costvector[c]
5:    if AllCand_c ≠ ∅ then
6:      Cand_c ← ∅
7:      for all s ∈ AllCand_c do
8:        if s ∉ ExtGoals then
9:          Cand_c ← Cand_c ∪ {s}
10:       end if
11:     end for
12:     if Cand_c ≠ ∅ then
13:       lastc ← c
14:       return Cand_c
15:     end if
16:   end if
17: end loop
```

Fig. 4. Procedure MINCOSTCAND.

```
Input: a COSPP P = ((S, s_0, A, F), W, G)
Output: a cost-optimal strong plan SP
1:  for all (s, a, s') ∈ S_τ do
2:    if s ∈ G then
3:      Cost(s) ← 0;
4:    else
5:      Cost(s) ← ∞;
6:      Δ(s, a) ← F(s, a);
7:    end if
8:  end for
9:  Cand ← ∅;
10: SP ← ∅;
11: OldExtGoals ← ∅;
12: ExtGoals ← G;
13: while (ExtGoals ≠ OldExtGoals) do
14:   if s_0 ∈ ExtGoals then
15:     return SP;
16:   end if
17:   CANDIDATEEXTENSION();
18:   OldExtGoals ← ExtGoals;
19:   PLANEXTENSION();
20: end while
21: return Fail;
```

Fig. 5. Procedure COSTOPTIMALSTRONGPLAN.

in the extended goals, the procedure returns them, otherwise it increases $c$ and loops. Thus, even if updates may create duplicates of the same state in different elements of *costvector*, since the algorithm always extracts *first* the minimum cost instance of a state, and inserts it in *ExtGoals*, all its further instances (with higher cost) in *costvector* will be simply ignored.

### 3.3. *The* COSTOPTIMALSTRONGPLAN *routine*

Finally, the COSTOPTIMALSTRONGPLAN routine (Figure 5) initialises the cost value of each state and the sets $\Delta$, *Cand*, *ExtGoals* and *OldExtGoals*, then iterates applying the two subroutines described above. In particular, the procedure loops until the initial state $s_0$ is added to the extended goals, i.e., a cost-optimal strong plan for $s_0$ has been calculated, and returns it. Otherwise, the procedure continues until a fixed point is reached, i.e., there are no new extended goals to be added, and then it returns *Fail* to indicate that $s_0$ has no strong plan.

It is worth noting that, while this version of the COSTOPTIMALSTRONGPLAN routine solves the COSPP (and can be trivially extended to handle more than one initial state), we may also modify it in order to solve the cost-optimal strong *universal* planning problem. To this aim, it would be enough to delete the termination check in the *while* loop and let the algorithm end when the fixed point is reached, returning the whole *SP* which, in this case, would contain the cost-optimal strong plan for any reachable system state that has it.

### 3.4. *Time complexity of the algorithm*

**Proposition 3.1.** *Let $\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ be a* COSPP*, with $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$. Let $n = |S|$ be the number of states, $r = |\mathcal{A}|$ be the number of actions and $m = |\mathcal{S}_\tau|$ be the number of transitions in the* NDFSS *$\mathcal{S}$. Moreover, let $w$ be the highest transition cost assigned by $\mathcal{W}$, i.e., $w = \max_{\tau \in \mathcal{S}_\tau} \mathcal{W}(\tau)$. Then the time complexity of the cost-optimal strong planning algorithm is $O((m + n \cdot r) \cdot (k + \log(m)))$, where $k = \log(w)$.*

**Proof.** We assume to implement the *costvector* as binary tree. In particular, we associate the binary digit 0 to the left child of each tree node and the digit 1 to the right one, thus a path from the root to a leaf can be encoded by a binary number $c$, which we shall call the *leaf cost*. Each tree leaf has an associated set of states with the corresponding cost, as devised by the algorithm. If the cost of a state is updated, the state is attached to the corresponding leaf but not removed from the previous leaf.

It it clear that the complexity of COSTOPTIMALSTRONGPLAN strongly depends on the complexities of the CANDIDATEEXTENSION and PLANEXTENSION routines.

In particular, when called, the CANDIDATEEXTENSION routine (Procedure 2) iterates over the *predecessors* of each new state $s'$ in *ExtGoals* (lines 1–3), i.e., the elements of $Pre(s')$. Since we assume to have in memory the expanded dynamics of the graph, it is not required to compute the *Pre* function of line 2, so this step is performed in constant time. Otherwise, we may apply an explicit state space exploration algorithm to build it in

$O(|\mathcal{S}_\tau|)$. The CANDIDATEEXTENSION routine removes one state from a set $\Delta(s,a)$ (where $(s,a) \in Pre(s')$) in each iteration of its inner loop (line 3). Since these elements represent transition targets, we can also say that it removes a transition from the system in each iteration, thus the loop of line 3 may be executed at most $m$ times among all the calls to CANDIDATEEXTENSION.

The core of the CANDIDATEEXTENSION loop is the cost update on line 9. Since the new cost $c'$ is computed on line 6 as the maximum cost of all the transitions from $s$ with the same action $a$, then $Cost(s)$ can be updated (on line 9) at most $r$ times. The time needed for a cost update, given the *costvector* tree structure described above, can be overapproximated by the maximum tree height, since we simply need to find (or insert) the leaf with the corresponding cost and add a state to its set. The number of nodes in the such tree derives from the maximum weight that a state could be assigned to, that is at most $w \cdot m$, therefore the height of the *costvector* tree would be $O(log(w \cdot m)) = O(k + log(m))$. Thus, the overall execution cost of the CANDIDATEEXTENSION procedure is $O(m \cdot (k + log(m)))$.

On the other hand, the PLANEXTENSION routine (Procedure 3) computes $\alpha$ and the corresponding set of minimal-weight candidate states. Again, in the binary structure above, to find the minimum cost we simply have to find the leftmost leaf, thus performing at most $O(k + log(m))$ steps as discussed above, and then scan the associated set of states until we find one that has not been already added to $ExtGoals$. Once a leaf has been consumed, we remove it from the tree. Since we have $n$ states and each can be assigned with $r$ different (decreasing) costs during the algorithm execution, then we will scan at most $n \cdot r$ states in this loop. Thus, the overall complexity of PLANEXTENSION is $O((n \cdot r) \cdot (k + log(m)))$

To sum up, the overall complexity of COSTOPTIMALSTRONGPLAN is therefore $O(m \cdot (k + log(m)) + (n \cdot r) \cdot (k + log(m))) = O((m + n \cdot r) \cdot (k + log(m)))$. □

### 3.5. *Correctness and completeness of the algorithm*

The algorithm given in Figure 5 essentially iterates the two procedures CANDIDATEEXTENSION and PLANEXTENSION until the desired state has a plan or the fixed point is reached.

Let us indicate with $ExtGoals_k$ and $Cand_k$ the contents of the $ExtGoals$ and $Cand$ sets, respectively, at the $k$-th step of the algorithm. Moreover, let us call $Goals_k$ the union $G \cup ExtGoals_k$. Then, we can prove the following important property of the COSTOPTIMALSTRONGPLAN algorithm.

**Proposition 3.2.** *Let $u_k$ be the maximum cost of a state in $Goals_k$, that is $u_k = \max_{s \in Goals_k} Cost(s)$. Then, in any step $k \geq 1$ of the* COSTOPTIMALSTRONGPLAN *algorithm, all the states with a plan of minimum cost no greater than $u_k$ are in $Goals_k$. That is $\forall s \in S$, $Cost(s) \leq u_k \Rightarrow s \in Goals_k$*

**Proof.** At the first iteration of the algorithm ($k = 1$), $Goals_k = ExtGoals_k = G$ contains, by definition, all the states with a plan having cost zero (i.e., the goals).

Now, let us assume by induction that the property holds at step $k$. We shall prove that it still holds at step $k+1$, i.e., the new elements inserted in $Goals_{k+1}$ do not falsify it.

To this aim, let $\alpha_{k+1}$ be the minimum cost of a candidate in $Cand_{k+1}$, that is $\alpha_{k+1} = \min_{s \in Cand_{k+1}} Cost(s)$. We can simply prove that $u_k < \alpha_{k+1}$. Indeed, assume that $u_k \geq \alpha_{k+1}$: then there exists a state $s \in Cand_{k+1}$ s.t. $Cost(s) \leq u_k$. However, a state in $Cand_{k+1}$ cannot be in $Goals_k$ (since the algorithm moves to the $ExtGoals$ only states that are already in $Cand$), and this contradicts the induction hypothesis.

Note that the fact above implies that, at the end of step $k+1$, i.e., after the execution of PLANEXTENSION, we have that $u_{k+1} = \alpha_{k+1}$, since the algorithm moves in $Goals_{k+1}$ all the candidates with cost $\alpha_{k+1}$, which is greater than the previous maximum cost $u_k$.

Now assume that the property to be proved is falsified at step $k+1$. This implies that there exist one or more states $s$ s.t. $Cost(s) \leq u_{k+1}$ but $s \notin Goals_{k+1}$. Let us choose among these states the one with minimum cost. Since we know that $u_{k+1} = \alpha_{k+1}$, we can also write that $Cost(s) \leq \alpha_{k+1}$.

By induction hypothesis, since a state which is not in $Goals_{k+1}$ could not also be in $Goals_k$, we have that $u_k < Cost(s)$. Let us consider a cost-optimal strong plan for $s$. Such plan must contain at least one state $s' \notin Goals_k$. Indeed, if all the states of such plan were in $Goals_k$, then $s$ should be in $Goals_{k+1}$. Let us choose among these states the one with minimum cost.

We have two cases:

- if $s' = s$, then we have that, for some suitable action $a$, $F(s,a) \subseteq Goals_k$. This would imply that $s \in Cand_{k+1}$ and, since $Cost(s) \leq \alpha_{k+1}$, we would have that $Cost(s) = \alpha_{k+1}$ (recall that $\alpha_{k+1}$ is the minimum cost of a candidate in $Cand_{k+1}$). But in this case the algorithm would move $s$ in $Goals_{k+1}$, contradicting the hypothesis;
- if $s \neq s'$, then $Cost(s') < Cost(s)$ (by definition of cost of a plan). Again, since $Cost(s) \leq \alpha_{k+1}$, we have that $Cost(s') < \alpha_{k+1}$, so $s' \notin Cand_{k+1}$. Thus we also have that $s' \notin Goals_{k+1}$, and this contradicts the hypothesis since $s$ would not be the state with minimum cost s.t. $Cost(s) \leq \alpha_{k+1}$ and $s \notin Goals_{k+1}$. □

The property of Proposition 3.2 implies that, if a state enters in the extended goals (and is therefore included in the strong optimal plan), then its cost, i.e., the cost of the corresponding strong plan, cannot be improved. This proves the algorithm correctness.

The algorithm completeness can be also trivially derived from Proposition 3.2. Indeed we have that, in each step $k \geq 1$ of the algorithm, the minimum cost of a candidate $\alpha_{k'}$ is strictly increasing (otherwise, $u_{k'} < \alpha_{k'+1}$ would not hold) and all the states with a plan of minimum cost no greater than $u_k$ are in $Goals_k$. Thus, if $s_0$ has a strong plan, it will eventually enter $Goals_k$ and the process will stop.

More in general, if we remove the termination check in the *while* loop of the COSTOPTIMALSTRONGPLAN routine (Figure 5), thus enabling the algorithm to build a (cost-optimal) *universal* strong plan, its completeness still holds, as proved by the following proposition.

**Proposition 3.3.** *Let $s \in S$. If $s$ has a cost-optimal strong plan $P$, whose cost is not greater than $Cost(s_0)$, then there exists $k > 0$ s.t. $s \in ExtGoals_k$ and $(s, a, F(s,a))$ is added to SP.*

**Proof.** The proof follows from Proposition 3.2. As already described, the minimum cost of a candidate $\alpha_{k'}$ is strictly increasing in each step of the algorithm. Thus, the process will eventually end with one of the following conditions:

- the initial state $s_0$ is in $ExtGoals_k$ (if a strong plan exists for such state): in this case, at step $k$ all the states whose cost is not greater than $Cost(s_0)$, including $s$, are guaranteed to be in $ExtGoals_k$, too.
- there are no more candidate states that can be reached from the (extended) goals: in this case, since by hypothesis $s$ has a strong plan, thus it can reach the goal, it would be included in the last set $ExtGoals_k$. □

Thus, the cost-optimal *universal* strong plan that can be output by the COSTOPTIMALSTRONGPLAN algorithm includes all the states which have a cost-optimal strong plan.

Finally, the algorithm termination is guaranteed by the arguments used in Proposition 3.3. Indeed, since the minimum cost of a candidate is strictly increasing, the algorithm will eventually build the cost-optimal strong plans for the states with highest cost: at this point, no new candidates will be available, and the process will terminate.

## 4. SUPMurphi: A Strong Planning Algorithm Implementation Using the Disk-Based Approach

In this Section we present SUPMurphi,[33,28] a tool based on the UPMurphi planner which contains a prototypal implementation of the cost-optimal strong planning algorithm presented in Section 3 exploiting disk storage to extend the applicability of such an algorithm to very complex systems.[67]

### 4.1. *Explicit model checking techniques*

The UPMurphi engine, which is the core of SUPMurphi, contains algorithms directly derived from the explicit model checker CMurphi.[19]

Generally speaking, a model checker takes as input a *system model M*, which describes the structure of the system *state* (defined by a set of *state variables*), the set of its *initial states*, and the rules driving its behaviour, i.e., describing how the system state can evolve during the time. This is often given by means of guarded *transition rules* that update the system state when some particular conditions are true in the current state (i.e., if the transition is *enabled* on a state, it can be *fired* to generate the corresponding next state).

In *explicit* model checking, the transition rules enabled on the current state are fired to produce the set of next states. The algorithm then filters out from such set all the states that have been previously generated by other transitions, to prevent loops. Finally, each next state becomes, in turn, the current one and the process is iterated until no more next states are available.

In this way, the model checker performs an exhaustive search and progressively collects the complete set of the possible system states (also called the system *state space*). As a consequence, in general, such tools can work on *finite state systems* only. Thus, systems

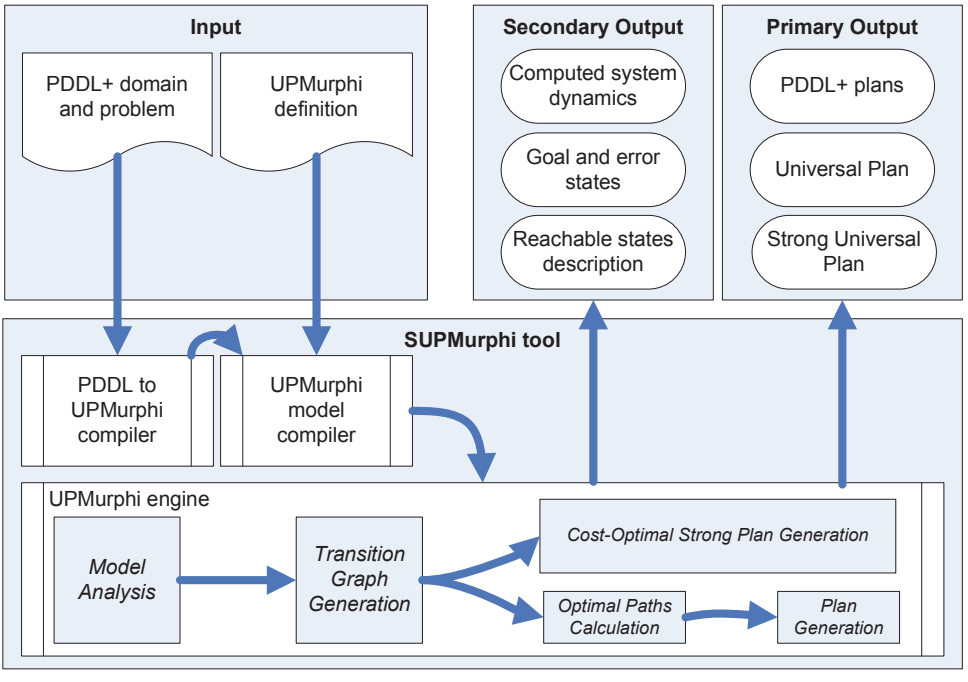


Fig. 6. Overall structure of the SUPMurphi tool.

with unbounded or continuous dynamics must be suitably limited and/or *discretised*, if possible, in order to be available for model checking techniques (for a discussion about how this discretisation can be correctly achieved and on its impact on the model reliability, see, e.g., Refs. 28, 33 and 37).

However, it is worth noting that, if a particular system state cannot be reached (e.g., a variable can never be set to a specific value, even if it is in its domain), it will be never generated or analysed. This kind of state space exploration is also called *reachability analysis*, and is the key of the effectiveness of explicit model checking with respect to symbolic ones on some classes of domains.

On the other hand, explicit model checking algorithms are subject to the *state explosion problem*, since collecting a large state space may require a very big storage space (usually RAM): however, the ability to generate only the reachable states of the system, together with many space saving techniques,[32] help to mitigate it.

Finally, if we feed the model checker with a set $E$ of *error states*, it will stop whenever one of these states is encountered. The sequence of transitions going from the initial state to the error state is called the *error trace*. However, if we look at error states as *goal states* and associate transitions with *actions*, we can use a model checker as a planner. Indeed, in this way the error trace would become a sequence of actions (if any) that reaches the goal starting from the system initial state, i.e, a plan (this is called planning-as-model-checking paradigm).

### 4.2. *Tool architecture*

Figure 6 shows the overall structure of SUPMurphi. Given the model of a system to analyse (written in the proprietary SUPMurphi description language or in the standard PDDL+ language), the SUPMurphi engine applies to it an explicit algorithm, organised as follows.

First, we exploit *reachability analysis* in order to build a representation of the system dynamics that can be later easily analysed and exploited during the other planning phases. This phase can be seen as an extension of the common breadth-first visit performed by classical explicit model checking algorithms. In particular, the tool starts the dynamics graph exploration from a set of user-defined *start states* and stops the expansion of a path when either a *goal state* or an *error state* is encountered. Here, we refer to goals as the states satisfying the planning problem goal, whereas we call errors the states satisfying some optional error conditions that can be included by the user within the domain description. These conditions are particularly useful to prune the dynamics graph, and thus speed-up the exploration, by explicitly excluding paths leading to an invalid or not interesting system state.

In the next phase, the tool rebuilds the complete system transition graph. Finally, if the system under analysis has a deterministic behaviour, then the tool calculates the optimal control paths (from the given initial states) and outputs them as a set of optimal plans. Otherwise, if the system is non-deterministic, it applies the algorithm given in Section 3 and returns the corresponding cost-optimal strong plan. The final plans can be exported to disk in various formats (binary, PDDL, CSV, etc.).

All the phases above make use of secondary memory (disk) to allow the manipulation of huge systems without incurring in out of memory errors, as discussed in Section 5. In particular, the disk feature of SUPMurphi:

(1) allows the use of the disk during the exploration of the system dynamics.
(2) implements an adaptation of the *hash compaction* technique,[84] which is compatible with the disk-based state space exploration above.
(3) allows one to use the disk to store the system graph(s), and query them directly without having to load data into memory.
(4) allows one to *pause* the synthesis process and resume the analysis later, also on another machine. It is even possible to restart the process from a previous phase to try different user settings.

However, to avoid an excessive time overhead, the disk structures used by SUPMurphi have been designed and implemented by taking into account their usage patterns, i.e., how (and how frequently) each structure is accessed during each phase of the planning process. This led to the definition of algorithms and data structures that minimise the number of disk seek-and-read operations, which are the bottleneck of any disk algorithm, since seeks suffer from a latency time that is much higher than the actual read/write time. For instance, SUPMurphi privileges sequential read/writes, at the cost of duplicating some information and/or requiring more disk space, which is not a problem since large disks are nowadays

very common. Moreover, SUPMurphi is able to adapt its algorithm to increase or decrease the disk usage with respect to the user specified options and the size of the system under analysis. Finally, when memory storage is absolutely required, the data stored in RAM is compressed. For example, states stored in the memory hash table are written as 40-bit signatures.

In the following we briefly describe the tasks accomplished by SUPMurphi in each phase, and the data structures used to support them.

### 4.3. *Disk data structures*

The SUPMurphi algorithm uses a set of memory and disk data structures. In particular, the only structure that is always stored in memory is the hash table $H$, used to remember visited states by storing their 40-bit signature and the associated index. The disk structures are described in the following.

- *Disk Queues.* Two FIFO queues $Q$ and $Q'$, used in several parts of the algorithm, are stored to disk. The head and tail segments of each queue are cached in memory to minimise disk accesses.
- *Reachables file RF*. It stores the complete definition of each reachable system state, and it is indexed by the hash table $H$. This file is created only if the user requires a complete symbolic dump of the state in the planner output.
- *Transitions file TF*. It compactly stores all the transitions encountered during the model analysis. Each entry in this file has the form $(s,d,(r_i,w_i,s'_i)_{i=1\ldots d})$ where $s$ is the index (as stored in $H$) of a state, $d$ is its out degree, $r_i$ is the index of the model rule which determines the $i$-th outgoing transition from $s$, $w_i$ its weight and $s'_i$ the corresponding target state. Note that states are stored as integer indexes, which are usually much smaller than the state description.
- *Startstates, Goals and Errors files.* The *startstates file SF*, *goals file GF* and *errors file EF* contain, respectively, the indexes of the start states and of the reached goal states, and the transitions (described as in the transitions file) which lead to an error state.
- *Actions file AF*. The *actions file AF* stores $(s,r)$ pairs where $s$ is a state index and $r$ is the action chosen for that state by the plan generation algorithm. Note that this is an internal encoding of a controller table.
- *Plans file PF*. The *plans file PF* contains strings of the form $s_0 r_0 s_1 r_1 \ldots s_n$, which represent a computed plan from the state (with index) $s_0$ to state $s_n$ through actions (rules) $r_0 \ldots r_{n-1}$.
- *Graph file TGF*. The *graph file TGF* is used to store the transition graph of the system in the form of adjacency lists. In practice, this file contains an ordered and further compacted representation of the data in the transitions file, to allow a faster navigation of the system dynamics. In particular, for each state (in index order), the file contains an adjacency list composed of $(r,w,s')$ triples, where $r$ is a rule index, $w$ its weight and $s'$ the reached state. The graph file is indexed by an in-memory structure that contains the disk position of the beginning of each list, to allow direct jumps to the adjacency list of a given state.

```
Input: S, the set of Startstates
 1: Q ← ∅;
 2: H ← ∅;
 3: for all s ∈ S do
 4:   i ←generate_index(s);
 5:   store(H,signature(s));
 6:   enqueue(Q,s);
 7:   write_state_index(SF,i);
 8: end for
 9: while Q ≠ ∅ do
10:   s ← dequeue(Q);
11:   outgoing ← ∅;
12:   for all (r,w,s') ∈ next(s) do
13:     if not contains(H,signature(s')) then
14:       i ←generate_index(s');
15:       store(H,signature(s'));
16:       outgoing ←outgoing∪(r,w,s');
17:       if is_goal(s') then
18:         write_state_index(GF,i);
19:       else if is_error(s') then
20:         write_transition(EF,s,r,w,s');
21:       else
22:         enqueue(Q,s');
23:       end if
24:     end if
25:   end for
26:   write_state_index(TF,s);
27:   write_number(TF,|outgoing|);
28:   for all (r,w,s') ∈transitions do
29:     write_transition(TF,r,w,s');
30:   end for
31: end while
```

Fig. 7. Procedure model analysis.

The transition graph storage *dynamically adapts* to the system size. Indeed, if the number of reachable system states is small enough to allow building the transition graph directly in memory, the graph file is not created and a corresponding memory structure is built instead, to allow faster graph navigation.

### 4.4. *Model analysis*

The model analysis algorithm (Figure 7), exploits a standard BFS search to explore the reachable system states, starting from the given start states. Clearly, when the search reaches an error state or a goal state, it does not explore further on that direction.

The first column of Figure 9 shows the memory and disk data structures used by this phase. The disk queue $Q$ is initially loaded with all the start states, whose assigned indexes are also written to the start states file $SF$. Then the algorithm dequeues a state from $Q$ and

```
1: while (TF is not completely read) do
2:   s ←read_state_index(TF);
3:   n ←read_number(TF);
4:   for i = 1 to n do
5:     (r, w, s') ← read_transition(TF);
6:     add_to_adjacency_list(TGF, s', r, w, s);
7:   end for
8: end while
```

Fig. 8. Procedure transition graph generation.

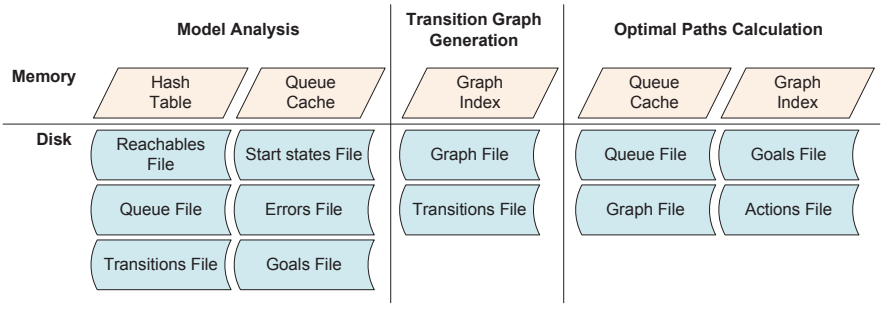


Fig. 9. Data structures used in the SUPMurphi planning phases.

visits all its successors, calculated by the next function. The exploration ends when there are no further states to expand in the exploration queue.

The memory hash table $H$ is used to avoid revisiting states. It is worth noting that, in this phase, *hash compaction*[84] is used to reduce the size of each entry of the hash table by substituting the states with their signatures in the in-memory hash table $H$, and storing the complete state representation to disk. These two state representation are kept synchronised by associating to each state signature a unique disk index which allows transparent access to the expanded state, if required.

Indeed, each time a state is reached by the search procedure, its signature is looked up in $H$ and, if it is not found (i.e., the state is *fresh*), then the state is given an index and written in the reachables file (if required) $RF$, whereas the corresponding transition is stored in the transitions file $TF$. If the state is a goal, its index is also written in the goals file $GF$, whereas if it is an error state the complete transition is written to the errors file $EF$. Finally, the state is enqueued in the disk queue $Q$ to be expanded later by the algorithm.

### 4.5. *Transition graph generation*

In this phase, the algorithm collects all the transition information generated by the model analysis and builds the inverted transition graph for the system.

Since the planning process requires to navigate the graph from the goals to the start states, the procedure in Figure 8 reads the transitions file $TF$ and writes each inverted

transition to a set of adjacency lists, which are stored in the disk graph file $TGF$ or in memory, if enough RAM is available. To this regard, the amount of RAM required to store the graph in memory is automatically evaluated by taking into account (1) the state space information collected in the previous phase, and (2) the size of the data structures needed to hold the memory graph. The second column of Figure 9 summarises the memory and disk data structures used by this phase.

Moreover, since the strong planning algorithm requires that the $Pre(s)$ function is given, the algorithm also stores the direct graph dynamics for non-deterministic systems, creating two *Transition Graph* files, which can be independently placed on disk or in memory, as explained above.



### 4.6. *Optimal paths calculation*

At this point, if the system under analysis is deterministic, the algorithm has all the information needed to decide the action to take in each system state, e.g., calculate the (optimal) control paths for each state that can reach a goal. The algorithm can be configured (via the user-specified options) to choose any feasible action, or the action with minimum/maximum weight.

The process is implemented as shown in Figure 10 and uses the memory and disk data structures shown in the last column of Figure 9.

The disk queue $Q'$ is initialised with the goal states found in the goals file $GF$, then the algorithm traverses the transition graph $TGF$ generated by the previous phase using a suitably modified version of the Dijkstra algorithm. The chosen edges and the corresponding weights are stored in memory and, when the process is complete, the whole structure is written to the actions file $AF$.

### 4.7. *Plan generation*

Finally, this phase accesses the startstates $SF$ and actions $AF$ files and writes in the plan file $PF$ the paths starting from the user-specified start states (or from all the states, if a universal plan is required) as illustrated in Figure 11.

### 4.8. *Cost-optimal strong plan generation*

If the system under analysis is non-deterministic, this phase applies to it the algorithm described in Section 3, which writes in the plan file $PF$ the cost-optimal strong plan, if any.

In particular, we implemented an enhanced version of the algorithm which, as discussed in Section 3.3, is also able to generate cost-optimal strong *universal* plans. To this aim, SUPMurphi can be instructed to stop when one of the following conditions is met:

- when the initial state (or any of the initial states given to the planner) is inserted in the plan;
- when the algorithm reaches the fixed point (i.e., no other state can be added to the plan);

The first termination condition, with a single initial state, returns a usual cost-optimal strong plan, whereas the second one returns a cost-optimal strong universal plan (if any).

```
1:  Q' ← ∅;
2:  for (i = 0 to number_of_states) do
3:    chosen_edge[s_i]← null;
4:    distance[s_i] ← ∞;
5:  end for
6:  while (GF is not completely read) do
7:    s ← read_state_index(GF);
8:    distance[s] ← 0;
9:    enqueue(Q', s);
10: end while
11: while (Q' is not empty) do
12:   s ←dequeue(Q');
13:   for (r, w, s') ∈adjacency_list(TGF, s) do
14:     if (chosen_edge[s'] = null ∨
        distance[s'] > distance[s] + w) then
15:       if (chosen_edge(s') = null) then
16:         enqueue(Q', s');
17:       end if
18:       chosen_edge[s'] ← (r, w, s);
19:       distance[s'] ← distance[s] + w;
20:     end if
21:   end for
22: end while
23: for all (s | chosen_edge[s] ≠ null) do
24:   write_action(AF, s,chosen_edge[s]);
25: end for
```

Fig. 10. Procedure compute optimal paths.

```
1: for (s ∈ SF) do
2:   plan ← ∅;
3:   while (has_action(AF, s)) do
4:     (r, w, s') ←read_action(AF, s);
5:     append_to_plan(plan, s, r);
6:     s ← s';
7:   end while
8:   write_plan(PF, plan);
9: end for
```

Fig. 11. Procedure plan generation.

**Example 4.1.** The SUPMurphi model given in Figure 12 shows how the *Hurried Passenger Problem* given in Example 2.1 can be modelled in the SUPMurphi description language.

In particular, the `ruleset` construct is used to model non-determinism of an action whilst the keyword `weight` gives the cost of the transitions, which can be a parametric function.

```
const
 HOME: 0; FCO : 1; CIA : 2; CDG : 3;
 BER : 4; AMS : 5; SFO : 6; GMT7: 7;
type
 day_type : 0..24;
 delay_t : 0..1;
 location_type: 0..6;
 start_type: 6..6;
var
 time[pddlname: time;]: day_type;
 location[pddlname: location;]:
  location_type;
-- the weight is given by time to wait
 in airport + flight time + delay
function w(delay: delay_t ; departure:
 day_type; length: day_type) : day_type;
begin
 return ((departure-time)+(length+delay)
  );
end;

ruleset t: start_type do
 startstate "At Home"
 location := HOME;
 end;
end;

rule "Q" (location=HOME)==>
weight: 1;
 begin
  location:=FCO;
  time:= 7;
 end;

rule "P" (location=HOME )==>
weight: 1;
 begin
  location:=CIA;
  time:= 4;
 end;

ruleset delay : delay_t do
rule "FlightA" (location=FCO & time<8)
 ==>
weight:  w(delay,8,1);
 begin
  location:=CDG;
  time:=time+w(delay,8,1);
 end; end;

ruleset delay : delay_t do
rule "FlightB" (location=CDG & time<10)
 ==>

weight: w(delay,10,9);
 begin
  location:=SFO;
  time:=time+w(delay,10,9)-GMT7;
 end; end;
```

```
ruleset delay : delay_t do
rule "FlightC" (location=CDG & time<19)
 ==>
weight: w(delay,19,9);
 begin
  location:=SFO;
  time:=time+w(delay,19,9)-GMT7;
 end; end;
ruleset delay : delay_t do
rule "FlightD" (location=CIA & time<5)
 ==>
weight: w(delay,5,3)+5;
 begin
  location:=AMS;
  time:=time+w(delay,5,3)+5;
 end; end;

ruleset delay : delay_t do
rule "FlighE" (location=FCO & time<8)==>
weight: w(delay,8,2);
 begin
  location:=BER;
  time:=time+w(delay,8,2);
 end; end;

ruleset delay : delay_t do
rule "FlightF" (location=BER & time<11)
 ==>
weight: w(delay,11,10);
 begin
  location:=SFO;
  time:=time+w(delay,11,10)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightG" (location=BER & time<12)
 ==>
weight: w(delay,12,10);
 begin
  location:=SFO;
  time:=time+w(delay,12,10)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightH" (location=AMS & time<15)
 ==>
weight: w(delay,15,10);
 begin
  location:=SFO;
  time:=time+w(delay,15,10)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightI" (location=BER & time<12)
 ==>
weight: w(delay,12,1);
 begin
  location:=AMS;
  time:=time+w(delay,12,1);
 end; end;

goal "On Time"
   (location=SFO & time <= 21);
metric: minimize;
```

Fig. 12. The SUPMurphi model for the *Hurried Passenger Problem* example.

```
 -- Strong Plan Filename: hurried.strong(text mode)
 -- (source[type],action name,Max Cost to goal) --> (reached states list)
 -- [I] = startstate [G] = goalstate [IG] = both start and goal state
(0[I],Q,17)-->(2)
(1,FlightD,22)-->(4, 3)
(2,FlighE,16)-->(6, 5)
(3,FlightH,13)-->(10[G], 9[G])
(4,FlightH,12)-->(10[G], 9[G])
(5,FlightF,12)-->(11[G], 13[G])
(6,FlightG,12)-->(12[G], 11[G])
(7,FlightB,11)-->(17[G], 16[G])
```

Fig. 13. SUPMurphi strong plan for The *Hurried Passenger Problem* model of Figure 12.

```
-- Source: Action(cost)->Target
State 0: Q(1)->2 P(1)->1
State 1: FlightD(10)->4 FlightD(9)->3
State 2: FlightA(3)->8 FlightA(2)->7 FlightE(4)->6 FlightE(4)->5
State 3: FlightH(13)->10 FlightH(12)->9
State 4: FlightH(12)->10 FlightH(11)->9
State 5: FlightF(12)->11 FlightF(11)->13 FlightG(13)->12 FlightG(12)->11 FlightI
 (4)->4 FlightI(3)->3
State 6: FlightG(12)->6 FlightG(11)->11 FlightG(4)->3 FlightG(3)->2
State 7: FlightB(11)->17 FlightB(10)->16 FlightC(20)->15 FlightC(19)->14
State 8: FlightC(19)->15 FlightC(18)->14
```

Fig. 14. The *Hurried Passenger Problem* graph of model in Figure 12.



In our example, the state is composed of the location of the passenger and the local clock time. The `startstate` construct models the initial passenger's time (i.e., 6*am* at *home*). Then, each flight of Table 1 is modelled by an action that requires to stay in the airport at least one hour before the departure time. The cost of a flight is given by function `w()` which sums the time spent in the airport waiting for the flight, the duration of the flight and the delay of the flight (which could be zero).

Figure 13 shows the solution stored in the plan file, which contains a cost-optimal strong plan for the initial state as well as for every state of the system which can be reached from the initial state (i.e., a cost-optimal strong *universal* plan). For the sake of completeness, the graph of reachable states generated by UPMurphi is reported in Figure 14.

## 5. Experimentation

In this section we present two case studies for the cost-optimal strong planning algorithm. First, we show a classical benchmark example, namely the Omelette domain,[59] which has been also used to test the MBP planner.[21] This allows us to better compare our approach and its scalability with this well known (strong) planning tool. Then, we introduce the *inverted pendulum on a cart*, a more realistic and complex domain where we show how the approach can be applied to continuous systems using the discretise and validate technique and exploiting the disk feature of SUPMurphi. All the experiments were performed using a Linux machine equipped with an Intel x86 CPU at 2.66 Ghz, with 4 Gb of RAM.

### 5.1. *Omelette*

The classical Omelette domain, described by Levesque,[59] is structured as follows: we have a supply of eggs, some of which may be nondeterministically bad or good. We have a bowl and a saucer, which can be emptied at any time. It is possible to grab an egg and to break it into the saucer, if it is empty. In such a case, if the egg is good, the content of the saucer will be poured to the bowl, otherwise it will be discarded. It is also possible to break an egg directly into the bowl avoiding the transfer action. However, if the broken egg is bad then the whole content of the bowl will be discarded. The goal is to get $g$ good eggs and no bad ones into the bowl.

For a strong plan to be available, we need to limit the number of bad eggs in our supply, say up to $n_b$ eggs, and we also fix the total number of available eggs (good or bad), to $n_e$. Otherwise, the problem would admit only a strong cyclic solution.[21]

#### 5.1.1. *System modelling*

The basic Omelette domain can be easily defined as a propositional PDDL domain and requires no cost optimisation. However, since our strong planning algorithm was designed to handle complex, continuous domains with costs, we want to increase progressively the number of eggs ($n_e$). In this setting, modelling the domain with only propositions would be equally possible, although the model would be unnatural (just consider that we may need a proposition for each possible combination of used eggs and found rotten eggs). Thus, we need to use counters, i.e., *fluents* in the PDDL definition.

Basically, the planning domain and problem are defined as follows. The *grab* action is executed if there is at least one egg left in the supply. Then, *break_egg* or *break_egg_into_bowl* can be executed, to break the egg in the saucer or in the bowl, respectively. In both cases, the action effect nondeterministically decides if the egg is good or not by considering if the maximum number of bad eggs $n_b$ has been reached or not. Then, the "bad eggs counter" is increased accordingly.

Once an egg is in the saucer, the planner can discard it (action *discard_egg*) if it is rotten. Otherwise, the planner can transfer the egg to the bowl (action *add_egg*). Differently, if the new egg has been broken directly in the bowl, and it is rotten, the planner can only discard all the contents of the bowl (action *empty_bowl*). The process then restarts. Every time an egg is grabbed, the supply count is decreased, whereas whenever a good egg is added to the bowl, the "omelette counter" is increased. The planning problem requires, therefore, to set the "omelette counter" to $g$, starting with a supply of $n_e$ eggs.

#### 5.1.2. *Strong universal planning*

In the first experimental evaluation, we want to compare the SUPMurphi performances on such a domain with the ones of MBP, which still represents a milestone in this field. We expect MBP to be far superior to SUPMurphi in the simple cases, thanks to the great optimisation given by the symbolic representation of the domain. Thus, we performed several experiments with increasing number of eggs, i.e., $n_e \in [5, 10, 50, 100, 500]$. The goal $g$ was set

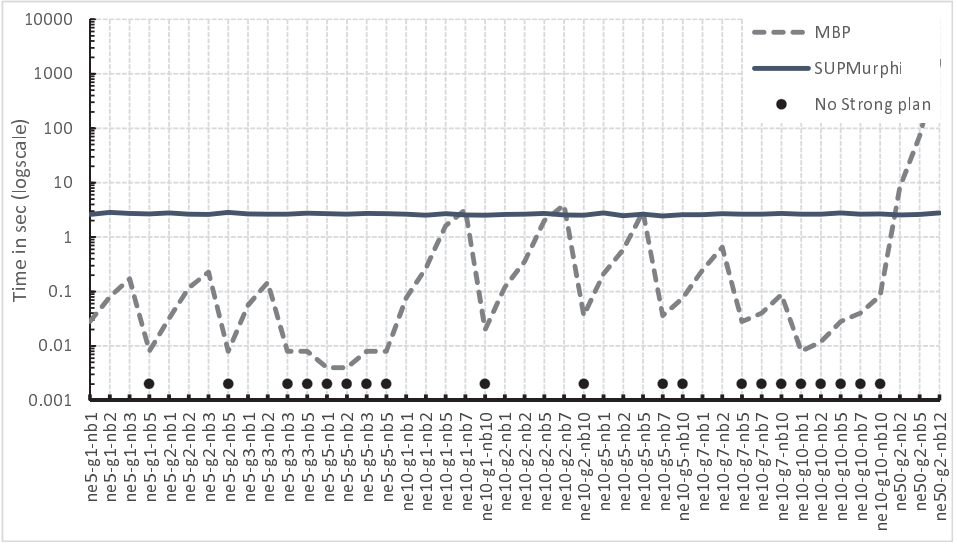


Fig. 15. Omelette problem instances with no costs. $n_e$ total eggs, at most $n_b$ bad eggs and $g$ good eggs as goal.

to $n_e \cdot p$ and the number of bad eggs to $n_e \cdot q$, with $p, q \in [5\%, 10\%, 25\%, 50\%, 75\%, 100\%]$. Clearly, problem instances with $g > (n_e - n_b)$ cannot admit strong solutions (i.e., there are not enough good eggs to reach a goal in the worst case). Nevertheless, we considered these instances to evaluate the strength of the tools also in problems with no solutions.

**Omelette with No Costs.** The results in Figure 15 provide two different kinds of feedbacks: from an algorithmic perspective, they give an empirical validation about the SUPMurphi's strong algorithm, since SUPMurphi and MBP agree about the existence (or not) of the strong solution on the same problem instances.

From a computational point of view, we see that, as we expected, MBP is far superior to SUPMurphi on the smaller instances. However, MBP does not scale well when the instance size grows (in particular, it seems that the increasing number of bad eggs upper bound $n_b$ is the main issue affecting the scalability). On the contrary, SUPMurphi, in this experimental setting, presents a constant execution time mainly thanks to the numerical representation of the PDDL fluents.

**Omelette with Costs.** Now we want to push further the comparison, so we put costs on our actions and try to cost-optimise the strong plan. Modelling these costs in SUPMurphi is straightforward, whereas to put them in the MBP model we have to *unwind* the costs, by substituting each action having cost $k$ with $k$ consecutive actions. To this aim, we used the PDDL `when` construct which allows us to mitigate the model growth due to the number of unwind actions. Each action $a$ has been supplied with a counter $c_a \in [0, k]$. Then, each action has now two (mutually exclusive) conditional effects: the former increases $c_a$ by 1 when the $c_a < k$, the latter applies the original action and resets the counter $c_a$.

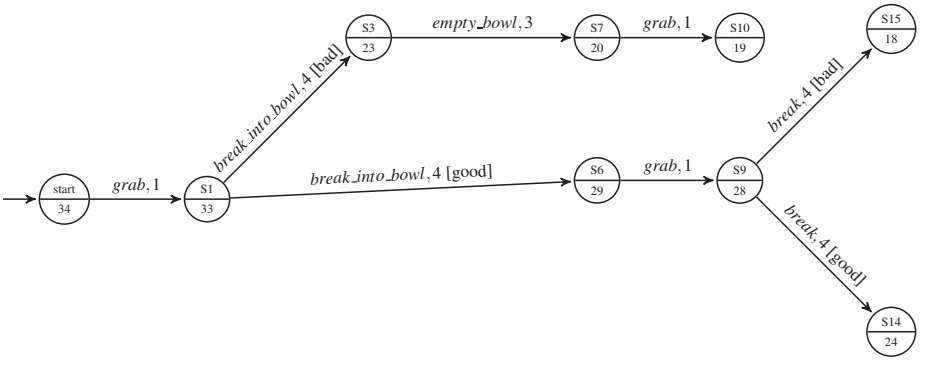


Fig. 16. Detail of the graphical representation of a cost-optimal strong plan for the Omelette instance $n_e 5 - g4 - n_b 1 - w1$.

The cost distribution is as follows. Note that, to stress the impact of the costs on the tools performances, we executed several experiments with an increasing cost scale factor $w \in \{1,2,3\}$. The *grab* action has cost $w \cdot 1$, *break_egg* and *break_egg_into_bowl* have both cost $w \cdot 4$. Transferring the contents of the saucer to the bowl (*add_egg*) and discarding the saucer content (*discard_egg*) cost $3 \cdot w$. Differently, once a bad egg has been broken into the bowl, the cost to empty the bowl (*empty_bowl*) is $3 \cdot k \cdot w$ where $k$ is the number of eggs in the bowl. Thus, the cost of emptying the saucer is constant, whereas emptying the bowl has a cost which changes according to the number of eggs into the bowl.

We run the same set of experiments as in the non-weighted case, by limiting each execution up to 90 minutes. This time the solutions are less trivial, so it is worth looking at one of them: Figure 16 shows a detail of the cost-optimal strong plan generated by SUPMurphi for a problem instance with $n_e = 5$ total eggs, requiring $g = 4$ good eggs with at most $n_e = 1$ bad egg. In the graph, the lower part of each state contains the cost of its strong plan, whereas on the edges we put the corresponding action, its cost and the nondeterministic outcome, if any.

Initially the only allowed action is to grab an egg. Then, the planner suggests to break the egg directly into the bowl, leading the system either to a "unlucky" state $s3$ in the case of a rotten egg, or in a "lucky" state $s6$ if the egg is good. Continuing from $s6$ on the lucky trajectory, the planner proposes to grab a new egg and then break it into the saucer (state $s9$). From there, two feasible states are available, again a lucky one ($s14$) and unlucky one ($s15$).

We can observe now that reaching the goal from the unlucky states ($s3$ and $s15$) has a cost *lower* than the cost from the lucky ones ($s6$ and $s14$). Indeed, the longest lucky plan, which implies breaking always an egg into the saucer and the transfer it to the bowl, is only a feasible plan, but not optimal since, for a generic instance, a feasible plan as "always break an egg into the saucer" will have in the worst case a total cost of $(g + n_b) \cdot (grab_{cost} + break_{cost}) + g \cdot add_{cost} + n_b \cdot discard_{cost}$, namely the sum of the costs to grab/break/add the required eggs plus the cost to grab/break/discard the rotten eggs. For example, such a feasible plan would cost 40 in the instance of Figure 16.

On the other hand, the presence of the nondeterminism along the plan makes disadvantageous to break an egg into the bowl, especially when it is already filled with several good eggs, due to the high cost required to empty it in case of rotten egg. However, when the bad eggs upper bound is reached, the remaining part of the plan will not be affected by the nondeterminism. Therefore, from the state $s7$ onwards the cost to reach a goal is given by the cost to grab $g$ eggs plus the cost to break them directly into the bowl, which becomes *always* the best choice.

The experiment results are summarised in the Figure 17. Note that the graph omits instances with no strong solutions as well as instances with running time over the imposed timeout.

Looking at MBP's execution time, we observe that the increasing $n_b$ is still a drawback for MBP, as in the previous experimental setting. Moreover, the results show that the unwinding operation strongly affects the MBP's performances, limiting the number of instances solved with respect to the ones of Figure 15. Indeed, MBP is not able to find a solution (within the maximum time allowed) for instances bigger than $n_e = 5$, $g = 3$, $n_b 1$ with cost factor $w = 1$. This behaviour is due to the action-cost unwinding operation, indeed the higher the scale factor $w$, the larger the resulting state space.

SUPMurphi is not directly influenced by the raising of the cost factor $w$, since it has been explicitly developed to deal with real numbers. Nevertheless, the growing of the number of total eggs $n_e$ (e.g., from 50 to 100 and from 100 to 500) as well as the maximum number of bad eggs $n_b$, although to a lesser extent, forces SUPMurphi to explore a wider state space, generating a larger system graph.

It is worth noting that the comparison has been aimed to show the peculiarities of both MBP and SUPMurphi, providing an attempt to build a strong planning problem with costs upon a problem with no costs. The results confirm that MBP is well suited to perform strong planning in propositional domains when no action costs are given, whilst SUPMurphi is useful in case of strong planning problems with fluents and action costs.

Finally, for the sake of completeness, the entire set of PDDL domain and problem instances, together with the cost-optimal strong plans found by SUPMurphi, are available on the authors website.[b]



### 5.2. *The inverted pendulum on a cart*

In this section we show how SUPMurphi can be applied to a continuous domain, namely the *inverted pendulum on a cart*, exploiting the discretise and validate approach,[28] which we briefly recall in the following.

#### 5.2.1. *The discretise and validate approach*

Instead of reasoning about the continuous dynamics of a system directly, the discretise and validate approach allows one to choose a discretisation and handle discretised versions of

[b]Repository of the Omelette Case Study Reports: Problems and SUPMurphi plans. http://www.di.univaq.it/gdellape/download.php?fn=supm_omelette

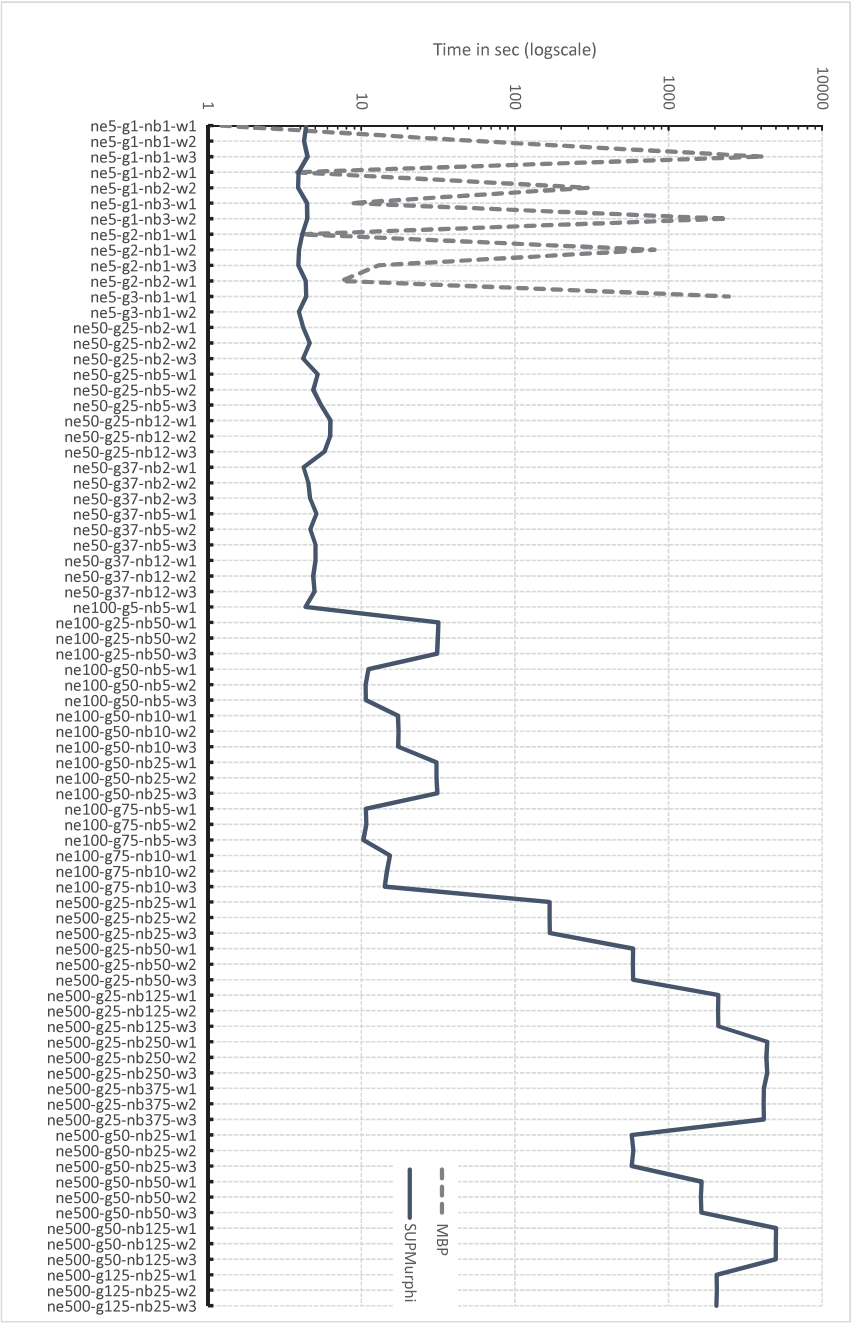


Fig. 17. Omelette problem instances with costs (unwinding). $n_e$ total eggs, at most $n_b$ bad eggs, $g$ good eggs as goal and $w$ cost multiplier scale factor.

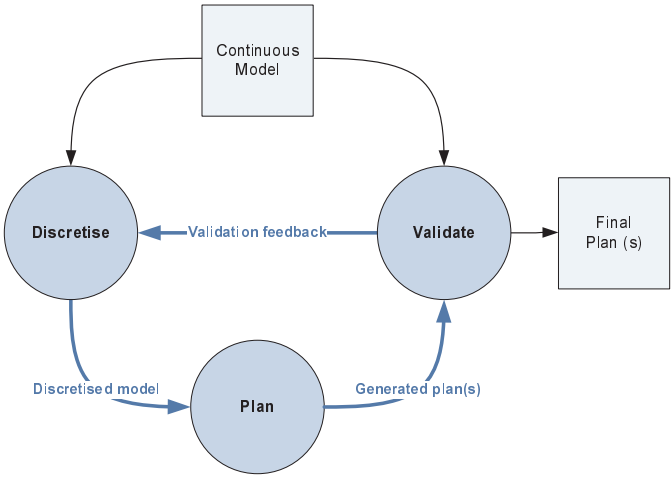


Fig. 18. The discretise and validate approach.

the problem, refining the discretisation until a valid solution for the original continuous problem is found.

This approach has been first implemented in the UPMurphi tool,[28] and works as sketched in Figure 18.

UPMurphi takes as input the continuous version of the problem and starts by discretising it. Then the planning-as-model-checking paradigm is used to explore the discretised state space and find a plan. Once a plan has been found it is then validated against the original continuous problem. The validation phase may state that either the discretised solution is valid or that a refinement of the discretisation is required. In the latter case, the current discretisation is refined and the process restarts, looping until the validation returns a satisfying result.

The discretisation involves both the continuous domain variables and the continuous-time dynamics. In particular, the dynamics discretisation is achieved by quantising the timeline in uniform discrete time steps and executing transitions and state updates at the beginning of each of these *clock ticks*.

In general, finding a suitable discretisation for a continuous system is a nontrivial task. Indeed, the finer the discretisation, the bigger the resulting state space, and a discretisation may lead to a prohibitive state space. On the other hand, a coarse discretisation may "hide" important aspects of the domain dynamics, thus making the discretised model (and the plan found for it) incorrect. This is why the key of the approach is the combination of the discretisation refinement and the validation.

The validation can be done either using the plan validator VAL,[44] or, if the system dynamics cannot be handled by VAL, using a monte-carlo technique, as discussed in section 5.2.5.

In the following, we describe how we performed the different phases of the approach to find a valid strong universal plan for the inverted pendulum.

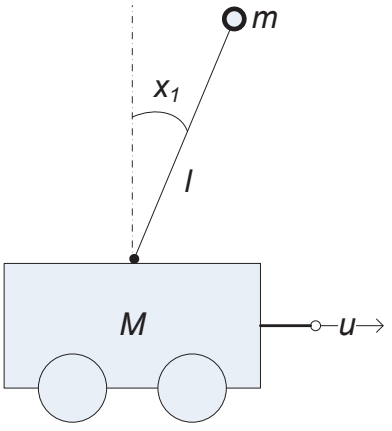


Fig. 19. Inverted pendulum on a cart.



### 5.2.2. *The continuous model*

The inverted pendulum on a cart (depicted in Figure 19) is a non-linear continuous system where the objective is to move the pole in the vertical position and then maintain it in this position, by applying an appropriate horizontal force to the cart.

Note that this apparently simple case study presents instead an important issue common to many real-world systems. Indeed, many control problems, e.g., engineering (i.e., the regulation of a steering antenna[48]) or robotics,[86] can be reduced to an inverted pendulum problem. Indeed, previous works dealing with this system are based on neural networks,[3] as well as cell mapping[75,83] and aim to minimise the *time* spent to reach the equilibrium.[83]

The pendulum state is described by two real variables:

- $x_1$ is the pendulum angle (w.r.t. the vertical axis) with $x_1 \in [-1.5, 1.5]$ rad;
- $x_2$ is the angular velocity with $x_2 \in [-8, 8]$ rad/sec.

The continuous dynamics is described by the following system of differential equations:

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x_2} = \dfrac{g_e \sin(x_1) - \left[\dfrac{\cos(x_1)}{m_p + m_c}\right] \left[m_p l x_2^2 \sin(x_1) + u\right]}{4l/3 - [m_p/(m_p + m_c)] l \cos^2(x_1)} \end{cases}$$

where $g_e$ is the gravitational constant, $m_p = 0.1$ Kg is the mass of the pole, $m_c = 0.9$ Kg is the mass of the cart, $l = 0.5$ m is the half-length of the pole and $u \in [-50, 50]$N is the force applied to the cart.

#### 5.2.3. *Model discretisation*

Discretising the inverted pendulum model requires choosing a discretisation for the real variables $x_1$ and $x_2$ and using the discrete-time function of the dynamics. Furthermore, the actions need to be discretised as well, by considering a set of specific values from their admissible range. In this case, actions model the amount of force $u$ applied to the cart, and the discretisation will result in several versions of the action `apply_force`.

Clearly, the key issue in this process is the discretisation of the real variables, as their approximation can have a significant impact on the correctness of the computation, given the high non-linearity of the transition function. On the other hand, the discretisation is also the main cause of the state explosion which would prevent from finding a plan. Following the iterative process discussed above, we started with an initial discretisation, rounding $x_1$ and $x_2$ to 0.001 rad and 0.01 rad/sec, respectively. This discretisation results in a state space of 4 800 000 states. After generating the strong plan, we found out that it was not valid, therefore we refined the discretisation rounding $x_1$ and $x_2$ to 0.0001 rad and 0.0001 rad/sec, respectively, ending with a state space of 4 800 000 000 states.[c]

The transition function is discretised using a time discretisation of 0.01 seconds (according to the setting presented by Papa *et al.*[75]). The goal of having the pole in the vertical position is defined with a small tolerance of $\pm 0.025$ rad for $x_1$ and $\pm 0.05$ rad/sec for $x_2$.

Finally, the set of possible values for the force $u$ applied to the cart is discretised in 10 discrete actions, namely $\mathcal{A} = \{\pm 50, \pm 25, \pm 10, \pm 5, \pm 2\}$.

In this setting, the *cost of a transition* is given by the absolute value of the applied force, i.e., for any $s$, $s'$, $\mathcal{W}(s,a,s') = |a|$. Therefore, a plan of minimum cost minimises the *worst-case* energy consumption.

The *non-determinism* of the system is given by possible disturbances on the actuator that may result in a small variation of the force actually applied. Hence, due these disturbances, $x_2$ can non-deterministically assume, with uniform probability distribution, a value that differs from the expected one by a small $\lambda \in [-\Lambda, \Lambda]$ with steps of 0.01 rad/sec.

#### 5.2.4. *Strong universal plan*

We generated a set of initial states defined as the set $\{(x_1,x_2)|x_1 \in [-1.5, -1.49, \ldots, -0.04, -0.03] \wedge x_2 = 0\}$.

Then, we considered different instances of the problem, taking into account disturbances of increasing size, with $\Lambda \in \{0.01, 0.02, 0.03, 0.04\}$ rad/sec. For each instance, we used SUPMurphi to generate a cost-optimal strong universal plan (if any), and the corresponding results are summarised in Table 3.

Here, for each problem instance, we report some statistics about the corresponding-graph $G$, i.e., the state space size ($|S|$), the number of reachable states and edges (*Reach* and $Reach_\tau$, respectively), the number of actions ($|\mathcal{A}|$) and the average and maximum

[c]For sake of clarity, in the following we will present only the results obtained using the discretisation with which a valid strong plan was generated.

Table 3. Experimental results for the inverted pendulum on a cart problem.

| Instance | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $\Lambda$ | | 0.01 | 0.02 | 0.03 | 0.04 |
| *Graph* | $\|S\|$ | 4,800,000,000 | | | |
| | *Reach* | 2,607,874 | 8,692,125 | 17,433,451 | 29,738,571 |
| | $Reach_\tau$ | 5,579,592 | 29,965,390 | 84,126,372 | 192,120,479 |
| | $\|\mathcal{A}\|$ | 10 | | | |
| | $\max(\delta(s))$ | 30 | 50 | 70 | 90 |
| | $\text{avg}(\delta(s))$ | 2.14 | 3.45 | 4.83 | 6.46 |
| *Disk* | .transitions | 60 Mb | 315 Mb | 873 Mb | 2 Gb |
| | .graph (directed) | memory mode | memory mode | memory mode | 2.4 Gb (*disk* mode) |
| | .graph (inverse) | memory mode | memory mode | memory mode | memory mode |
| *SP* | size | 17,413 | 59,187 | 113,605 | 193,124 |
| | $\max(C(s_0))$ N | 302 | 305 | 305 | 305 |
| | $\min(C(s_0))$N | 100 | 100 | 100 | 100 |
| | iterations | 49 | 67 | 75 | 82 |
| | time (min) | 1.2 | 3.5 | 8 | 36 |

out degree ($avg(\delta(s))$ and $\max(\delta(s))$, respectively) of the states. Note that the disk-based algorithm has been stressed during the plan synthesis, as in the case of the last instance.

Finally, in the cost-optimal strong plan *SP* section, we report the size of the *SP* as the number of plans that can be extracted from *SP*, the maximum and minimum cost ($\max(C(s_0)$ and $\min(C(s_0))$ of a strong plan starting from an initial state (i.e., the minimum and maximum amount of energy required to reach a goal from a startstate in the worst-case), the number of iterations performed before reaching the fix point, and the total time required to generate the *SP*.[d] Note that the strong plan is intended to be generated off-line, and then embedded into the inverted pendulum system. Therefore, there are no strict temporal constraints in generating the strong plan, while an important requirement is to have a fast response from it when used online. To this aim, we experimented that querying the OBDD representing the *SP* takes negligible time (following the approach first presented by Della Penna *et al.*[27]). The synthesis time (which includes all the phases) never required more than one hour.

Note that, for the sake of completeness, all the strong plans of Table 3 have been made publicly available on the authors website.[e] Moreover, in order to provide a 2D graphical

[d]Here total time refers to the time needed to complete all the phases reported in Figure 6.
[e]Repository of the Inverted Pendulum Case Study Reports: problems and SUPMurphi plans.
http://www.di.univaq.it/gdellape/download.php?fn=supm_inverted

**Input:** Strong plan $SP$,plan time discretisation $\gamma_T$, validation time discretisation $\delta_T$, number of iterations $N$

1: valid_plans $\leftarrow 0$
2: **for all** $i \in [1,N]$ **do**
3: $s_0 \leftarrow$ sample_state_from($SP$);
4: $\pi = \tau_0, \tau_1, \ldots, \tau_{n-1} \leftarrow$ extract_trajectory($SP(s_0)$);
5: **for all** $j \in [0, n-1]$ **do**
6: $\Omega \leftarrow \lfloor \frac{\gamma_T}{\delta_T} \rfloor$;
7: **for all** $k \in [1, \Omega - 1]$ **do**
8: $s_{j_{k+1}} \leftarrow F(s_{j_k}, a) | (s_j, a, F(s_j, a)) \in SP, a \in \tau_j$; *//using* $\delta_T$ *and variable discretisation of* $10^{-7}$
9: **end for**
10: $s_{j+1} \leftarrow s_{j_\Omega}$;
11: **end for**
12: **if** is_goal($s_n$) **then**
13: valid_plans++;
14: **end if**
15: **return** valid_plans / N;
16: **end for**

Fig. 20. Monte-Carlo validation.

representation of a generated strong plan, we created an interactive demo which is available on the web,[f] exploiting a well-suited multidimensional visualisation technique, namely the *parallel-coordinates*.

### 5.2.5. *Plan validation*

The inverted pendulum dynamics cannot be handled by the current version of the validator VAL, as it is not able to deal with trigonometric functions.

Nevertheless, it is possible to use a Monte-Carlo based approach[68] to verify that the discretised solution (i.e., a plan for the discretised model with a given initial state) is valid also when tested against the continuous model. Indeed, the continuous dynamics of the inverted pendulum can be well-approximated using a *validating discretisation* of $10^{-7}$ for representing the real variables (therefore considering an approximation to $\mu$rads).[g]

Namely, the Monte-Carlo algorithm we used for validation is reported in Figure 20.

At each iteration, a state is randomly chosen, and the corresponding trajectory (as defined in Definition 2.3) is extracted from the strong plan. The plan is then executed using the validating discretisation ($10^{-7}$) for the real variables. In doing this, in order to make the validation even more reliable, a finer time discretisation is used ($\delta_T = 0.001$) than the one

[f]Web Demo of the Strong Plan generated for the Inverted Pendulum Case Study http://www.di.univaq.it/fabio.mercorio/parcord.html

[g]Clearly, such a discretisation could not be used during the generation phase, due to the resulting enormous state explosion. On the other hand, validating a plan only requires the computation of a new state for each action of the plan, and no search is involved, and this makes possible the use of such a fine discretisation.

Table 4. Monte-Carlo validation results.

| Instance | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| valid $\pi$ % | 100% | 100% | 100% | 100% |
| $\bar{D_{x_1}}$ | $2.82 \cdot 10^{-5}$ | $2.81 \cdot 10^{-5}$ | $2.8 \cdot 10^{-5}$ | $2.8 \cdot 10^{-5}$ |
| $\bar{D_{x_2}}$ | $2.78 \cdot 10^{-5}$ | $2.75 \cdot 10^{-5}$ | $2.1 \cdot 10^{-4}$ | $2.2 \cdot 10^{-4}$ |
| $\sigma^2_{x_1}$ | $3.5 \cdot 10^{-10}$ | $3.24 \cdot 10^{-10}$ | $3.45 \cdot 10^{-10}$ | $3.52 \cdot 10^{-10}$ |
| $\sigma^2_{x_2}$ | $3.24 \cdot 10^{-10}$ | $3.5 \cdot 10^{-10}$ | $3.41 \cdot 10^{-10}$ | $3.56 \cdot 10^{-10}$ |
| $\max(D_{x_1})$ | $1.8 \cdot 10^{-4}$ | $1.6 \cdot 10^{-4}$ | $1.9 \cdot 10^{-4}$ | $2 \cdot 10^{-4}$ |
| $\min(D_{x_1})$ | 0 | 0 | 0 | 0 |
| $\max(D_{x_2})$ | $1.6 \cdot 10^{-4}$ | $1.8 \cdot 10^{-4}$ | $2.1 \cdot 10^{-4}$ | $2.2 \cdot 10^{-4}$ |
| $\min(D_{x_2})$ | 0 | 0 | 0 | 0 |

used at planning time ($\gamma_T = 0.01$), and this increases the number of state updates in each iteration (lines 6–7). The check in line 11 is used to verify that the plan brings the system to the goal even using the validating variable and time discretisations. Here, in order to handle border states, we consider the goal region $[-0.0251, 0.0251] \times [-0.06, 0.06]$.

We applied the Monte-Carlo validation to the four strong plans reported in Table 3. Clearly, the effectiveness and the reliability of the Monte-Carlo based validation depend on the execution of a very high number of iterations until a desired indicator converges on an expected value, as described by Rubinstein.[80] To this aim, `is_goal(`$s_n$`)` is used also to compute the 1-norm distances between the final state reached using the strong plan discretisation ($x1^p_{s_n}$, $x2^p_{s_n}$) and the corresponding states reached using the validating discretisation ($x1^m_{s_n}$, $x2^m_{s_n}$), therefore for each validation $\pi_i$ we have $D_{x_1}(\pi_i) = |x1^p_{s_n} - x1^m_{s_n}|$, $D_{x_2}(\pi_i) = |x2^p_{s_n} - x2^m_{s_n}|$. Finally, the average distances $\bar{D}_{x_1}$, $\bar{D}_{x_2}$ and their variance $\sigma^2_{x_1}$, $\sigma^2_{x_2}$ are computed.

Table 4 shows the results of the Monte-Carlo with 1 million iterations. All the plans are valid, and the average error due to the discretisation is in the order of $10^{-5}$ radians. Furthermore, the variance values for both $x_1$ and $x_2$ tend to zero, thus making our validation results quite effective and reliable.

## 6. Related Work

The application of planning techniques ranges from robotics to manufacturing processes,[76,29,85] and embedded systems.[20,38] More recently the planning via model-checking paradigm has been applied to knowledge discovery tasks, such as data consistency

analysis and cleaning,[70,15,89,90] data integration[14] as well as the sensitivity analysis over aggregated indicators.[71] Nowadays they make an important role in real world applications in the sea,[26,39] on earth,[81,53] as well as in space.[79,30] Therefore, a growing number of automatic planning tools have been developed, implementing advanced planning strategies and heuristics (see, e.g., Mali *et al.*,[62] Hsu *et al.*[45]).

With the increasing need for modelling real-world scenarios, in the last years, the (strong) planning in non-deterministic domains has had a growing interest in the planning community and then many techniques[57,55,54,78] and heuristics[40,63] have been proposed.

**Strong Planning with Classical Planners.** In the work of Kuter *et al.*[57] the NDP algorithm exploits the use of some classical planners (e.g., FF by Hoffmann *et al.*[43] and SGPlan by Hsu *et al.*[46] are used in instance) to deal with non-deterministic domains, looking for strong cyclic solutions in fully observable domains. Given a planning problem $P$ and a classical planner $R$, the algorithm generates a *deterministic* subproblem of $P$ (e.g., if an action $a$ has two distinct outcomes then it replaces action $a$ with two new actions $a_1$, $a_2$ with a single deterministic outcome). Then, it calls $R$ on each sequence of the subproblem. Finally, it collects and combines the results for each sequence providing the final solution, if any. It is worth noting that NDP does not use BDDs to represent the system dynamics, but the *conjunctive abstraction*, which gives less compression but can be used by any classical planner which deal with STRIPS domains, avoiding any modification to the planner. Other approaches to deal with uncertainty and nondeterminism using classical planners have been developed, e.g., by Sonto *et al.*[74] and Bonet *et al.*[13]

Several heuristics have been exploited in combination with BDDs in order to improve the performance of non-deterministic planners in real-world domains. To give an example, UMOP[50] uses both BDDs and heuristics to *guide* the search for strong and strong cyclic solutions. An improvement of UMOP has been presented recently in the work of Jensen *et al.*,[49] where the authors introduce NADL (a new description language for non-deterministic domains). Moreover, the new UMOP planner provides an algorithm (based on the previous one) which looks for *optimistic* plans by relaxing domain optimality constraints when no strong nor strong cyclic solutions exist. Mattmuller *et al.*[63] use a LAO* search (based on AND/OR graph) with heuristics based on pattern databases[36] to synthesise strong and strong cyclic solutions.

**Strong Planning Tools and Algorithms.** Despite a lot of approaches dealing with non-determinism have been, a key contribution in this field still comes from Cimatti *et al.*,[21] where the authors present an algorithm to find strong plans implemented in MBP, a planner based on symbolic model checking. MBP produces a universal plan,[82] which provides optimal solutions with respect to the plan length (i.e., the worst execution among the possible non-deterministic plan executions is of minimum length). Moreover, the use of Ordered Binary Decision Diagrams (OBDDs) together with symbolic model checking techniques allows a compact encoding of the state space and very efficient set theoretical operations.

More recently MBP was improved by *Gamer*,[55,54] a BDD based planner which is able to synthesise strong and strong cyclic solutions for non-deterministic domains. In

particular, Gamer transforms the non-deterministic planning problem into a two-players turn-taking game using a non-deterministic version of the PDDL language (i.e., NPDDL[5]). To give the main idea, the first player chooses an action (i.e., it sets the value of a variable) and the second player (i.e., the behaviour) chooses one of the possible outcomes for the action. Clearly, the translation process guarantees that the second player will choose all the possible non-deterministic behaviours for the selected action. Then, a minimised state encoding for the domain is computed[35] providing a very compact BDD representation. Finally, Gamer applies a modified version of the algorithm by Cimatti *et al.*[21] which is able to deal with the two players behaviour. Despite a little overhead needed to enlarge the state definition in order to support of players' turn, the main benefit given by the translation process is to lead to a small BDD, and then to a better performance, with respect to MBP. This approach, which is very promising, needs to be further tested on non-deterministic domains.

It is worth noting that both MBP and Gamer are backward-search planners based on a symbolic approach and so they require to compute the inverse function of the system dynamics, which may be often difficult to invert, especially for nonlinear domains. On the other hand, explicit algorithms, such as those used in the present paper, can easily handle such systems, since they perform a forward exploration of the system dynamics graph. Finally, three planners are worth of mentioning as very effective solutions related to our approach, namely MyND,[63] PRP[73] and POND.[17] Specifically, MyND and PRP aim at synthesising strong *cyclic* solutions in case of fully observability, whilst POND looks for strong solutions in partially observable domains.

**Cost-Optimal Strong Planning.** In general, the cost-optimal strong planning problem could be cast as a strategy synthesis problem for a multistage game with two players moving simultaneously or alternating moves.[41] Note that, in the latter case, the formulation would be very similar to the mentioned approach of Edelkamp *et al.*,[35] with the difference that we are not restricted to unitary costs. Thus, an universal plan could be also devised by applying some game theory algorithms, such as a *minimax* strategy for the first player where, in each game state, the player selects the action that minimises the maximum cost (to reach the goal) that the disturbance (i.e., the other player), with its choice, may inflict to it. Such a game theoretic casting, however, would be of very little help from a computational point of view, since in our setting the (matrix-like) *normal form* of the game would be intractable even for small systems. Indeed, the choices of the players depend on the possible configurations of the system, where each configuration is determined by a valid assignment of the domain variables and by the actions available for both players. Moreover, the pay-off to be inserted in the matrix are available only as "local costs" of the actions, whereas the global (final) cost of an action is available only when the problem is solved. The same can be said with respect to the so-called (tree-like) *extended form* of the game, which is usually the input for the minimax solvers.

It is worth noting that, with respect to the cost-optimality, both MBP and Gamer are able to synthesise optimal strong solutions only with respect to the plan length (i.e., each action has a unitary cost). It could be adapted to support costs and devise cost-optimal

solutions only by "unary encoding" the weights, that is, by replacing a transition of weight $k$ with $k$ contiguous deterministic transitions. In this case, however, its complexity, in the worst case, would be exponentially higher than the one of the algorithm presented in this paper.

The situation is exactly analogous to model checking based analysis of Markov chains.[58,31] Of course, in principle, stationary distributions for Markov chains can be computed using classical numerical techniques[4] for Markov chain analysis. However, for dynamic systems, our setting here, the number of states (easily beyond $10^{10}$) of the Markov chains to be analysed rules out matrix based methods. The same would happen if our problem is cast as a Mixed Integer Linear Programming (MILP) problem: it would be possible but, again, it would generate a MILP of size exponential in the input.

In the literature, planning based on Markov Decision Precesses (MDPs) has been proved very effective,[16,12,87] and, more recently, a variety of techniques have been proposed to solve continuous MDPs.[69,65,64] However, MDP-based approaches deal with probabilistic distributions taking into account the stochastic outcomes of actions. Therefore, whether a solution provided by MDP planning algorithms can be defined *strong* depends on probability and cost distribution.

**Strong Planning and Model Checking.** There is also a link between strong planning and CTL model checking. Indeed, it has been observed[50,54] that the problems of strong and strong cyclic planning can be also addressed using CTL model checking. More precisely, applying the common idea of planning via model checking,[42] the search of a strong cyclic goal φ can be casted as the problem of satisfying the CTL formula $AGEF\neg\varphi$. Similarly, the synthesis of a strong solution can be solved by satisfying the formula $AF\neg\varphi$.[25,77]

**Strong Planning and Dynamic Programming.** The cost-optimal strong planning problem, due to its recursive definition (see Section 2), could be also approached through dynamic programming. Indeed, the solution presented in this paper has the general setting of a dynamic programming algorithm. However, although for strong planning problems *without costs* a dynamic programming-based solution is quite straightforward, as in the work of Cimatti *et al.*,[21] in our setting the selection of the (cost-optimal and strong) action to perform in each state may require many update cycles and scans of the system state space, due to the cost-optimality property. Therefore, without specifically-designed data structures and support algorithms, the time and space complexity of a general dynamic programming solution for this problem would quickly become too high. These optimised structures and algorithms are indeed at the heart of the approach presented in this paper, which has good complexity bounds thanks to its particular implementation (see Section 3.4).

**Strong Planning and Control Theory.** Many of the cited strong planning techniques are also applied in the control field, which has some similarities with universal planning.[11] For an overview of other control approaches that are applied to multistage control in uncertain environments the reader can refer, e.g., to the book of Zdzislaw Bubnicki,[18] in particular

for what concerns the relational modeling of uncertainty. Moreover, among the other well-known approaches to such kind of control problems, we may cite the fuzzy dynamic programming by Kacprzyk *et al.*,[52] or the robust control.[56,34] All these techniques, however, cannot be easily adapted to solve cost-optimal strong planning problems, in particular for what concerns the strongness property.

**Planning in Continuous Domains.** Finally, it is worth noting that this paper also deals with planning in continuous domains, a research area that is receiving a growing attention in the planning community.[24,72,60,88,61,8,9]



## 7. Conclusions and Future Work

Automatic planning and control techniques are nowadays deeply studied, and their applied results are embedded in a lot of widespread products. In real world scenarios, systems very often have to act in environments which might influence the actuators and sensor readings, thus making the system's behaviour nondeterministic, particularly due to possible different outcomes of planned actions.

In this paper we presented an algorithm that solves the strong planning problem in nondeterministic domains. The main novelty of this algorithm is that it is also able to optimise the resulting plan with respect to a generic cost function, i.e., it performs cost-optimal strong planning, without any remarkable complexity growth. In addition, as opposite to most strong planning approaches currently available, the presented algorithm is based on explicit model checking techniques, which allows one to extend the class of treatable problems to a wide range of hybrid or nonlinear domains, which are very common when dealing with nondeterminism.

The devised algorithm has been formally proved as correct and complete, and its complexity, if the number of transitions in the system or the range of possible transition costs are reasonable (to say, a billion transitions or different costs), is dominated by the number of (visited) transitions in the system graph, which is a good bound for such kind of problems.

We also presented a prototypal implementation of the algorithm in a (strong) planning tool, namely SUPMurphi, which exploits an intelligent disk usage pattern to further extend the size and complexity of the systems it can analyse.

Finally, the proposed methodology has been validated through two case studies: the benchmark omelette problem, where we compared our approach with the state-of-the-art strong planning tool MBP, and the inverted pendulum on a cart problem, where we showed how our framework can be used to deal with continuous and nonlinear systems. The results of this first experimentation show that the presented algorithm is correct and can be a valuable aid to cost-optimal strong planning problems. The next step will be to refine and optimise the current prototypal implementation, which already shows a good performance, in order to effectively experiment our methodology on more challenging models, investigate the algorithm scalability in real-world planning problems, and systematically compare our approach against competitor tools and algorithms present in the literature.

## References

1. Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin and Robert E. Tarjan, Faster algorithms for the shortest path problem, *J. ACM* **37**(2) (April 1990) 213–223.
2. Alexandre Albore, Héctor Palacios and Hector Geffner, Compiling uncertainty away in non-deterministic conformant planning, in *ECAI* (2010), pp. 465–470.
3. Charles W. Anderson, Learning to control an inverted pendulum using neural networks, *IEEE Control System Magazine* **9**(3) (1989) 31–37.
4. Ehrhard Behrends, *Introduction to Markov Chains* (Vieweg Teubner, 2000).
5. Piergiorgio Bertoli, Alessandro Cimatti, Ugo Dal Lago and Marco Pistore, Extending PDDL to nondeterminism, limited sensing and iterative conditional, in *ICAPS 03, Workshop on PDDL* (AAAI Press, 2003), pp. 15–24.
6. Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri and Paolo Traverso, Planning in non-deterministic domains under partial observability via symbolic model checking, in *17th IJCAI* (Morgan Kaufmann, 2001), pp. 473–478.
7. Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri and Paolo Traverso, Strong planning under partial observability, *Artificial Intelligence* **170** (April 2006) 337–384.
8. Sergiy Bogomolov, Daniele Magazzeni, Stefano Minopoli and Martin Wehrle, PDDL+ planning with hybrid automata: Foundations of translating must behavior, in *Proc. of the 25th Int. Conf. on Automated Planning and Scheduling* (*ICAPS-15*) (2015).
9. Sergiy Bogomolov, Daniele Magazzeni, Andreas Podelski and Martin Wehrle, Planning as model checking in hybrid domains, in *Proc. of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (Québec City, Québec, Canada, July 27–31, 2014), pp. 2228–2234.
10. Blai Bonet and Hector Geffner, Planning with incomplete information as heuristic search in belief space (2000).
11. Blai Bonet and Hector Geffner, Planning and control in artificial intelligence: A unifying perspective, *Applied Intelligence* **14** (2001) 2001.
12. Blai Bonet and Hector Geffner, mGPT: A probabilistic planner based on heuristic search, *Journal of Artificial Intelligence Research* **24** (2005) 933–944.
13. Blai Bonet, Héctor Palacios and Héctor Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners, in *the Nineteenth Int. Conf. on Automated Planning and Scheduling* (*ICAPS*) (2009).
14. Roberto Boselli, Mirko Cesarini, Fabio Mercorio and Mario Mezzanzanica, A policy-based cleansing and integration framework for labour and healthcare data, in *Knowledge Discovery and Data Mining*, *LNCS 8401* (Springer, 2014), pp. 141–168.
15. Roberto Boselli, Mario Mezzanzanica, Mirko Cesarini and Fabio Mercorio, Planning meets data cleansing, in *The 24th Int. Conf. on Automated Planning and Scheduling* (*ICAPS 2014*) (AAAI, 2014), pp. 439–443.
16. Craig Boutilier, Thomas Dean and Steve Hanks, Decision-theoretic planning: Structural assumptions and computational leverage, *Journal of Artificial Intelligence Research* **11** (1999) 1–94.
17. Daniel Bryce, POND: The partially-observable and non-deterministic planner, *ICAPS 2006* (2006), p. 58.
18. Zdzislaw Bubnicki, *Modern Control Theory* (Springer, 2005).
19. Cached Murphi Web Page, http://www.dsi.uniroma1.it/~tronci/cached.murphi.html (2006).
20. Graziano Chesi and Yeung Sam Hung, Global path-planning for constrained and optimal visual servoing, *IEEE Transactions on Robotics* **23**(5) (2007) 1050–1060.
21. Alessandro Cimatti, Marco Pistore, Marco Roveri and Paolo Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, *Artificial Intelligence* **147**(1) (2003) 35–84.
22. Alessandro Cimatti, Marco Roveri and Paolo Traverso, Strong planning in non-deterministic domains via model checking, in *AIPS '98* (1998), pp. 36–43.

23. Alssandro Cimatti, Marco Roveri and Paolo Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in *IAAI 98* (AAAI Press, 1998), pp. 875–881.
24. Amanda Jane Coles, Andrew Coles, Maria Fox and Derek Long, COLIN: Planning with continuous linear numeric change, *J. Artif. Intell. Res.* (*JAIR*) **44** (2012) 1–96.
25. Marco Daniele, Paolo Traverso and Moshe Y. Vardi, Strong cyclic planning revisited, in *the 5th European Conference on Planning: Recent Advances in AI Planning* (*ECP '99*) (Springer-Verlag, London, UK, 2000), pp. 35–48.
26. Jnaneshwar Das, Frederic Py, Thom Maughan, Tom O'Reilly, Monique Messie, John P. Ryan, Gaurav S. Sukhatme and Kanna Rajan, Coordinated sampling of dynamic oceanographic features with underwater vehicles and drifters, *I. J. Robotic Res.* **31**(5) (2012) 626–646.
27. Giuseppe Della Penna, Benedetto Intrigila, Nadia Lauri and Daniele Magazzeni, Fast and compact encoding of numerical controllers using OBDDs, in *Informatics in Control, Automation and Robotics* (Springer, 2009), pp. 75–87.
28. Giuseppe Della Penna, Benedetto Intrigila, Daniele Magazzeni and Fabio Mercorio, UPMurphi: A tool for universal planning on PDDL+ problems, in *ICAPS 09* (AAAI Press, 2009), pp. 106–113.
29. Giuseppe Della Penna, Benedetto Intrigila, Daniele Magazzeni and Fabio Mercorio, A PDDL+ benchmark problem: The batch chemical plant, in *ICAPS 10* (AAAI Press, Toronto, Canada, 2010), pp. 222–225.
30. Giuseppe Della Penna, Benedetto Intrigila, Daniele Magazzeni and Fabio Mercorio, Planning for autonomous planetary vehicles, in *The Sixth Int. Conf. on Autonomic and Autonomous Systems* (IEEE, Cancun, Mexico, 2010), pp. 131–136.
31. Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci and Marisa Venturini Zilli, Finite horizon analysis of Markov chains with the murphi verifier, *STTT* **8**(4–5) (2006) 397–409.
32. Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci and Marisa Venturini Zilli, Exploiting transition locality in automatic verification of finite-state concurrent systems, *International Journal on Software Tools for Technology Transfer* **6**(4) (2004) 320–341.
33. Giuseppe Della Penna, Daniele Magazzeni and Fabio Mercorio, A universal planning system for hybrid domains, *Applied Intelligence* **36**(4) (2011) 932–959.
34. Giuseppe Della Penna, Daniele Magazzeni, Alberto Tofani, Benedetto Intrigila, Igor Melatti and Enrico Tronci, Automatic synthesis of robust numerical controllers, in *ICAS 07* (IEEE Computer Society, 2007), p. 4.
35. Stefan Edelkamp and Malte Helmert, Exhibiting knowledge in planning problems to minimize state encoding length, in *ECP 99* (1999), pp. 135–147.
36. Stefan Edelkamp and Peter Kissmann, Partial symbolic pattern databases for optimal sequential planning, in *KI* (2008), pp. 193–200.
37. Maria Fox, Derek Long and Daniele Magazzeni, Automatic construction of efficient multiple battery usage policies, in *the 21st Int. Conf. on Automated Planning and Scheduling* (*ICAPS 2011*) (Freiburg, Germany, June 11–16, 2011).
38. Maria Fox, Derek Long and Daniele Magazzeni, Plan-based policies for efficient multiple battery load management, *J. Artif. Intell. Res.* (*JAIR*) **44** (2012) 335–382.
39. Maria Fox, Derek Long and Daniele Magazzeni, Plan-based policy-learning for autonomous feature tracking, in *The Twenty-Second Int. Conf. on Automated Planning and Scheduling* (*ICAPS*) (2012).
40. Jicheng Fu, Vincent Ng, Farokh B. Bastani and I-Ling Yen, Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems, in *IJCAI* (2011), pp. 1949–1954.
41. Drew Fudenberg and Jean Tirole, *Game Theory* (MIT Press, August 1991).
42. Fausto Giunchiglia and Paolo Traverso, Planning as model checking, in *5th European Conference on Planning: Recent Advances in AI Planning* (Springer-Verlag, London, UK, 2000), pp. 1–20.

43. Jörg Hoffmann and Bernhard Nebel, The FF planning system: Fast plan generation through heuristic search, *Journal of Artificial Intelligence Research* **14** (2001) 253–302.
44. Richard Howey, Derek Long and Maria Fox, VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL, in *16th IEEE Int. Conf. on Tools with Artificial Intelligence* (*ICTAI 2004*) (IEEE, 2004), pp. 294–301.
45. Chih-Wei Hsu, Yixin Chen and Benjamin W. Wah, Subgoal ordering and granularity control for incremental planning, *International Journal on Artificial Intelligence Tools* **16**(04) (2007) 707–723.
46. Chih-Wei Hsu, Benjamin W. Wah, Ruoyun Huang and Yixin Chen, New features in SGPlan for handling preferences and constraints in PDDL3.0, in *Proc. of the Fifth International Planning Competition* (Citeseer, 2006), pp. 39–42.
47. Wei Huang, Zhonghua Wen, Yunfei Jiang and Lihua Wu, Observation reduction for strong plans, in *20th IJCAI* (Morgan Kaufmann, 2007), pp. 1930–1935.
48. Stojce Dimov Ilcev, Antenna systems for mobile satellite applications, in *CriMiCo 2009* (2009), pp. 393 –398.
49. Rune M. Jensen and Manuela M. Veloso, OBDD-based universal planning for synchronized agents in non-deterministic domains, *JAIR* **13** (2000) 189–226.
50. Rune M. Jensen, Manuela M. Veloso and Randal E. Bryant, Guided symbolic universal planning, in *ICAPS* (2003), pp. 123–132.
51. F. Kabanza, M. Barbeau and R. St-Denis, Planning control rules for reactive agents, *Artificial Intelligence* **95** (1997) 67–113.
52. Janusz Kacprzyk and L. Sugianto, Multistage fuzzy control involving objective and subjective aspects, in *Proc. 2nd Int. Conf. on Knowledge-Based Intelligent Electronic Systems* (*KES 1998*), eds. Lakhmi C. Jain and R. K. Jain, Vol. 3 (IEEE, 1998, Adelaide, South Australia, April 21–23, 1998), pp. 564–573.
53. Scott Kiesel, Ethan Burns, Christopher Makoto Wilt and Wheeler Ruml, Integrating vehicle routing and motion planning, in *ICAPS* (2012).
54. Peter Kissmann and Stefan Edelkamp, Solving fully-observable non-deterministic planning problems via translation into a general game, in *KI 2009: Advances in Artificial Intelligence*, eds. Bärbel Mertsching, Marcus Hund and Zaheer Aziz, Vol. 5803 of *Lecture Notes in Computer Science* (Springer Berlin/Heidelberg, 2009), pp. 1–8.
55. Peter Kissmann and Stefan Edelkamp, Gamer, a general game playing agent, *KI* **25**(1) (2011) 49–52.
56. Gerhard Kreisselmeier and Thomas Birkholzer, Numerical nonlinear regulator design, *IEEE Transactions on Automatic Control* **39**(1) (1994) 33–46.
57. Ugur Kuter, Dana S. Nau, Elnatan Reisner and Robert P. Goldman, Using classical planners to solve nondeterministic planning problems, in *ICAPS 08* (AAAI Press, 2008), pp. 190–197.
58. Marta Z. Kwiatkowska, Gethin Norman and David Parker, Probabilistic symbolic model checking with PRISM: A hybrid approach, *STTT* **6**(2) (2004) 128–142.
59. Hector J. Levesque, What is planning in the presence of sensing? in *The Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference* (*AAAI 96, IAAI 96*), eds. William J. Clancey and Daniel S. Weld, Vol. 2 (AAAI Press/The MIT Press, 1996, Portland, Oregon, August 4–8, 1996), pp. 1139–1146.
60. Hui X. Li and Brian C. Williams, Generative planning for hybrid systems based on flow tubes, in *ICAPS* (2008), pp. 206–213.
61. Johannes Löhr, Patrick Eyerich, Thomas Keller and Bernhard Nebel, A planning based framework for controlling hybrid systems, in *ICAPS* (2012).
62. Amol Dattatraya Mali and Minh Tang, State-space planning with variants of A*, *International Journal on Artificial Intelligence Tools* **15**(03) (2006) 433–464.
63. Robert Mattmüller, Manuela Ortlieb, Malte Helmert and Pascal Bercher, Pattern database heuristics for fully observable nondeterministic planning, in *ICAPS* (2010), pp. 105–112.

64. Mausam Mausam, Piergiorgio Bertoli and Daniel S. Weld, A hybridized planner for stochastic domains, in *20th IJCAI* (Morgan Kaufmann, 2007), pp. 1972–1978.
65. Mausam Mausam and Daniel S. Weld, Planning with durative actions in stochastic domains, *Journal of Artificial Intelligence Research* **31** (January 2008) 33–82.
66. MBP: A Model Based Planner, http://mbp.fbk.eu/ (2013).
67. Fabio Mercorio, Model checking for universal planning in deterministic and non-deterministic domains, *AI Communications* **26**(2) (2013) 257–259.
68. N. Metropolis and S. Ulam, The Monte Carlo method, *J. Amer. Stat. Assoc.* **173**(5–6) (1949) 335–341.
69. Nicolas Meuleau, Emmanuel Benazera, Ronen I. Brafman, Eric A. Hansen and Mausam Mausam, A heuristic search approach to planning with continuous resources in stochastic domains, *JAIR* **34** (January 2009) 27–59.
70. Mario Mezzanzanica, Roberto Boselli, Mirko Cesarini and Fabio Mercorio, Improving data cleansing accuracy: A model-based approach, in *DATA 2014 — The International Conference on Data Technologies and Applications* (SciTePress, 2014).
71. Mario Mezzanzanica, Roberto Boselli, Mirko Cesarini and Fabio Mercorio, A model-based approach for developing data cleansing solutions, *The ACM Journal of Data and Information Quality* **5**(4) (March 2015) 1–28.
72. Matthew Molineaux, Matthew Klenk and David W. Aha, Planning in dynamic environments: Extending HTNs with nonlinear continuous effects, in *AAAI* (2010).
73. Christian J. Muise, Sheila A. McIlraith and J. Christopher Beck, Improved non-deterministic planning by exploiting state relevance, in *ICAPS* (2012).
74. Hoang-Khoi Nguyen, Dang-Vien Tran, Tran Cao Son and Enrico Pontelli, On computing conformant plans using classical planners: A generate-and-complete approach, in *The Twenty Second Int. Conf. on Automated Planning and Scheduling* (*ICAPS*) (2012).
75. Mauricio Papa, Jason Wood and Sujeet Shenoi, Evaluating controller robustness using cell mapping, *Fuzzy Sets and Systems* **121**(1) (2001) 3–12.
76. Simon Parkinson, Andrew Longstaff, Andrew Crampton and Peter Gregory, The application of automated planning to machine tool calibration, in *The Twenty-Second Int. Conf. on Automated Planning and Scheduling* (*ICAPS*) (2012).
77. Marco Pistore and Moshe Y. Vardi, The planning spectrum — One, two, three, infinity, in *The 18th Annual IEEE Symp. on Logic in Computer Science* (*LICS '03*) (IEEE Computer Society, Washington, DC, USA, 2003), p. 234.
78. Cédric Pralet, Gérard Verfaillie, Michel Lemaître and Guillaume Infantes, Constraint-based controller synthesis in non-deterministic and partially observable domains, in *The 19th European Conference on Artificial Intelligence* (*ECAI*) (2010), pp. 681–686.
79. Sudhakar Y. Reddy, Jeremy Frank, Michael Iatauro, Matthew E. Boyce, Elif Kürklü, Mitchell Ai-Chang and Ari K. Jónsson, Planning solar array operations on the international space station, *ACM TIST* **2**(4) (2011) 41.
80. Reuven Y. Rubinstein, *Simulation and the Monte Carlo Method*, Vol. 190 (Wiley-interscience, 2009).
81. Wheeler Ruml, Minh Binh Do, Rong Zhou and Markus P. J. Fromherz, On-line planning and scheduling: An application to controlling modular printers, *J. Artif. Intell. Res.* (*JAIR*) **40** (2011) 415–468.
82. Marcel Schoppers, Universal plans for reactive robots in unpredictable environments, in *IJCAI 87* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987), pp. 1039–1046.
83. S. M. Smith and D. J. Comer, An algorithm for automated fuzzy logic controller tuning, in *IEEE Int. Conf. on Fuzzy Systems* (1992), pp. 615–622.
84. Ulrich Stern and David L. Dill, Improved probabilistic verification by hash compaction, in *CHARME '95: The IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (Springer-Verlag, London, UK, 1995), pp. 206–224.

85. Kai M. Wurm, Christian Dornhege, Bernhard Nebel, Wolfram Burgard and Cyrill Stachniss, Coordinating heterogeneous teams of robots using temporal symbolic planning, *Auton. Robots* **34**(4) (2013) 277–294.
86. Kazuhito Yokoi, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Shuji Kajita and Hirohisa Hirukawa, Experimental study of biped locomotion of humanoid robot HRP-1S, in *Experimental Robotics VIII*, Vol. 5 of *Springer Tracts in Advanced Robotics* (Springer, 2003), pp. 75–84.
87. Sung Wook Yoon, Alan Fern and Robert Givan, Inductive policy selection for first-order MDPs, in *UAI 02* (Morgan Kaufmann, San Francisco, CA, 2002), pp. 568–576.
88. Zahra Zamani, Scott Sanner and Cheng Fang, Symbolic dynamic programming for continuous state and action MDPs, in *AAAI* (2012).
89. Roberto Boselli, Mirko Cesarini, Fabio Mercorio and Mario Mezzanzanica, Accurate data cleansing through model checking and machine learning techniques, in *Data Management Technologies and Applications*, eds. Markus Helfert, Andreas Holzinger, Orlando Belo and Chiara Francalanci, Vol. 178 of *Communications in Computer and Information Science* (Springer International Publishing, 2015), pp. 62–80.
90. Mario Mezzanzanica, Roberto Boselli, Mirko Cesarini and Fabio Mercorio, A model-based evaluation of data quality activities in KDD, *Information Processing & Management* **51**(2) (2015) 144–166.