Appl Intell (2012) 36:932–959
DOI 10.1007/s10489-011-0306-z

A universal planning system for hybrid domains

**Giuseppe Della Penna · Daniele Magazzeni ·
Fabio Mercorio**

Published online: 29 June 2011

Abstract Many real world problems involve hybrid systems, subject to (continuous) physical effects and controlled by (discrete) digital equipments. Indeed, many efforts are being made to extend the current planning systems and modelling languages to support such kind of domains. However, hybrid systems often present also a nonlinear behaviour and planning with continuous nonlinear change that is still a challenging issue.

In this paper we present the UPMurphi tool, a universal planner based on the discretise and validate approach that is capable of reasoning with mixed discrete/continuous domains, fully respecting the semantics of PDDL+. Given an initial discretisation, the hybrid system is discretised and given as input to UPMurphi, which performs universal planning on such an approximated model and checks the correctness of the results. If the validation fails, the approach is repeated by appropriately refining the discretisation.

To show the effectiveness of our approach, the paper presents two real hybrid domains where universal planning has been successfully performed using the UPMurphi tool.

Keywords Universal planning · Hybrid systems · PDDL+

A preliminary version of this paper appears in G. Della Penna, B. Intrigila, D. Magazzeni, F. Mercorio, UPMurphi: a Tool for Universal Planning on PDDL+ Problems, in: Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009), pp. 106–113.

G. Della Penna (✉) · F. Mercorio
Department of Computer Science, University of L'Aquila, L'Aquila, Italy
e-mail: giuseppe.dellapenna@univaq.it

F. Mercorio
e-mail: fabio.mercorio@univaq.it

D. Magazzeni
Department of Sciences, University of Chieti-Pescara, Pescara, Italy
e-mail: magazzeni@sci.unich.it

1 Introduction

Many real-world problems involve complex hybrid systems, where the state is described by both discrete and continuous components and the time-dependent dynamics is governed by differential equations. Indeed, in the past years, a relevant effort has been made to extend PDDL [44, 46], the standard modelling language for planning problems, in order to model mixed discrete/continuous domains with continuous resource consumption. This effort has lead to the definition of PDDL+ [29] that makes it possible to capture continuous processes and events.

In particular, the main contribution made by PDDL+ is the ability to model continuous change through the initiation and termination of *processes* which continuously modify the value of the numeric components of the state. For example, if a pump is switched on, then a filling process starts that continuously increases the water level in a tank and terminates when the pump is switched off. Moreover, PDDL+ allows the modelling of predictable exogenous *events* that occur as a consequence of some process. For example, a tank will eventually overflow if the pump is left switched on. Processes and events allow one to model in a more realistic way the behaviour of continuous systems, and this represents a further step in applying planning technology to real world problems.

The planning systems have been also improved to deal with numeric and temporal constraints (see, e.g.

[17, 22, 31, 57] or [52]), and they have been proved very useful to perform planning w.r.t. such problems. However, developing planners able to deal with PDDL+ domains is still an open issue.

In this paper we present a new planning methodology implemented into the UPMurphi tool, a universal planner capable of reasoning with mixed discrete continuous domains, fully respecting the semantics of PDDL+.

1.1 Motivation

A growing number of motivating applications shows the importance of dealing with mixed discrete continuous domains. Some examples are: product processing in a plant [1], activity management of an autonomous vehicle [41], voltage regulation planning [3], solar array operations on the International Space Station [51], oil refinery operations planning [7], planning for an airport control system [33], or slag foaming control [60].

These problems are described by hybrid systems where continuous physical rules are managed by discrete digital equipments. In such a context, it is crucial to reason about continuous change during the planning process [29]. To this aim, when the problems have a linear dynamics, it is possible to use planners such as COLIN [17] or TM-LPSAT [57] that make use of a hybrid approach combining planning techniques and linear programming.

However, several real world planning problems present complex *nonlinear* behaviours which are difficult to handle by any analytical method (see, e.g., [6, 9, 58]) or hybrid reasoning approach. Nonlinearity can arise from the intrinsic dynamics of the system (e.g., the regulation of a steering antenna, which leads to an inverted pendulum problem), or the saturation of actuators (e.g., valves that cannot open more than a certain limit, control surfaces in an aircraft that cannot be deflected more than a certain angle, etc.). Indeed, the behaviour of nonlinear systems can be so complex to be completely unpredictable after a small interval of time (see, e.g., [56]).

Thus, planning with continuous nonlinear change is a challenging issue.

1.2 Contribution

In this paper, we propose the *discretise and validate* approach to deal with PDDL+ domains having complex nonlinear dynamics. In this approach, the continuous problem is relaxed into a discretised one using rounded values and uniform discrete time steps. A forward reachability analysis based on model checking algorithms is then used to solve the relaxed problem, and the solutions are validated against the continuous problem using the validator VAL [36].

This approach has been implemented in the *UPMurphi* tool, which performs universal planning [55] on PDDL+ problems, i.e., it generates a *set of policies* for all states reachable from the initial ones.

In particular, by exploiting the discretisation, UPMurphi is able to build and analyse the transition graph for a large class of systems, including systems whose dynamics is nonlinear and hard to be inverted. Obviously, in order to have a precise analysis of a hybrid domain, a suitable discretisation of the continuous behaviour is required. On the other hand, the finer is the discretisation used to round the continuous component of the state, the bigger is the resulting state space. To this aim, model checking algorithms offer powerful compression techniques and very effective compact encodings [11, 18], that make UPMurphi able to deal with huge state spaces.

Moreover, discretising the continuous behaviour of a system involves a discretisation of the timeline, where the time flow is modelled using uniform discrete time steps. This, in turn, requires to encode the PDDL+ description of the domain into such a discretised setting. To this aim, we designed and implemented a compilation process which takes in input a domain/problem pair written in PDDL+ and generates the corresponding discretised model to be used by UPMurphi. This completely eliminates the need of learning any planner-specific language and manually re-encoding domains with different formalisms.

Therefore, the presented tool aims to offer a fully PDDL+ compliant universal planner that is able to cope with problems that are very hard to handle by the current state-of-the-art tools. To support this claim, the paper presents a set of case studies where UPMurphi was successfully applied to perform universal planning: two benchmark planning problems, i.e., a nonlinear variant of *Generators* and *Cooling System*, and two real world applications, namely the *Planetary Lander*, and the *Batch Chemical Plant*.

The paper is organised as follows. In Sect. 2 we provide a brief survey of related work, focusing on planners for hybrid domains and planning systems based on model checking techniques.

In Sect. 3 we present the discretise and validate approach, providing a schematic view of the overall process and describing a compilation procedure which takes a PDDL+ domain/problem pair and generates a corresponding discretised model. We consider a detailed example of a domain, the Satellite domain, and show how each PDDL+ element is mapped into the new discretised setting.

In Sect. 4 we describe the UPMurphi universal planner. In particular, we provide a formal description of the universal planning task on finite state systems which represent the formal model underlying the UPMurphi engine, then describe the model checking based algorithm used by the tool and give some details about its implementation.

In Sect. 5 we consider four case studies, with increasing complexity, and show how UPMurphi was successfully applied to their universal planning.

Finally, some concluding remarks are outlined in Sect. 6.

2 Related work

A great effort has been made to develop algorithms and tools able to deal with hybrid planning domains. The early work was [50], where the authors presented the partial-order planner Zeno, which uses differential equations to describe continuous processes. However, Zeno cannot cope with concurrent processes and does not scale well to large problems. More recent works include the OPTOP planner [45] that deals with linear continuous domains where concurrent processes do not affect the same variable.

The TM-LPSAT system, developed by Shin and Davis [57], combines SAT and LP solvers. The former is used to deal with the discrete component of the domain while the latter is used to handle the continuous one. TM-LPSAT can deal with processes modelled in PDDL2.1, however, it is limited to small linear problems.

Kongming [41, 42], thanks to the concept of Flow Tubes, is able to compactly represent hybrid plans and encode hybrid flow graphs as a mixed logic linear/nonlinear program, solvable using an off-the-shelf solver. However, it can only address planning problems with constant action duration.

COLIN [17] is a powerful tool for planning in domains with linear continuous processes. It extends the forward chaining temporal planner CRIKEY3 [16], making it able to reason with actions with continuous linear effects. COLIN integrates a guided state space search with linear programming, and supports duration-dependent effects, durative actions with continuous change and concurrent continuous change.

However, none of the approaches above can handle PDDL+ domains, which represent the target of the approach proposed in the present paper.

Finally, [47] deals with nonlinear continuous effects written in PDDL+, using a state projection algorithm implemented into a Hierarchical Task Network planner. The approach is very interesting and effective, however, no information about the optimality of the synthesised solutions is given in the paper and, as the authors argue, the scalability of their approach has not yet been evaluated on more complex case studies.

Our methodology has been implemented in a model checking based tool. Planning-as-model-checking has a strong heritage (see, e.g., [32]), since proving states reachability can be viewed as finding plans. In particular, in [13] the authors use a symbolic approach based on OBDDs, which allow a very compact encoding of the state space. Indeed, symbolic algorithms have been successfully applied to classical planning. On the other hand, they do not work well on hybrid systems with nonlinear dynamics, due to the complexity of the state transition function.

The Model Checking Integrated System (MIPS) [21, 22] is a very powerful and complex framework that makes use of a combination of both explicit and symbolic model checking based heuristic search. The MIPS performed very well in different planning competitions, however it is restricted to PDDL2.1 while the extended version MIPS-XXL [25] deals with PDDL3.

The UPPAAL/TIGA tool [2] is built on top of UPPAAL which allows the use of real variables only as clocks, thus excluding systems with nonlinear dynamics.

Other examples of model checking based deterministic planners include, among others, ProPlan [26] and BDDPlan [34], while the Model Based Planner (MBP) [4] uses symbolic model checking for non-deterministic domains.

Finally, in this paper we deal with universal planning, which has been introduced by [55] (see also, e.g., [8, 43]). Cimatti et al. [15] and Jensen and Veloso [38] use a symbolic (OBDD-based) model checking approach to synthesise optimal universal plans for non-deterministic plants. On the other hand, the DPlan [53] and FPlan [54] tools use an explicit state-based representation instead of OBDDs. However, they both require the explicit definition of an inverse function for each operator used in the domain, and thus their application is hard when dealing with systems whose dynamics is difficult to invert (the typical situation for hybrid and/or nonlinear systems).

3 The discretise and validate approach

In order to handle continuous domains, possibly having nonlinear dynamics, in this paper we propose the *discretise and validate* approach. The basic idea is to make a complex continuous problem much simpler by exploiting the discretisation, then perform planning on this simplified problem, and finally validate the plans with respect to the continuous model.

However, in a general setting, finding a suitable discretisation for a continuous system is a hard and challenging task, since it can affect the plan generation speed, the precision of the solution and, sometimes, its correctness. To this aim, the discretise and validate approach works using iterative refinement of an initially coarse discretisation of the continuous values. Indeed, the validation phase may state that either the discretised solutions are valid for the original continuous problem, or that a refinement of the discretisation is required. In the latter case, the current discretisation is refined and the process restarts.

A schematic view of the approach is shown in Fig. 1. Given a first discretisation, the continuous model is relaxed into a discretised one. Then, a forward reachability analysis is used to perform universal planning on the discretised model. The generated plans are validated against the continuous model using the VAL [36] plan validator. Since VAL

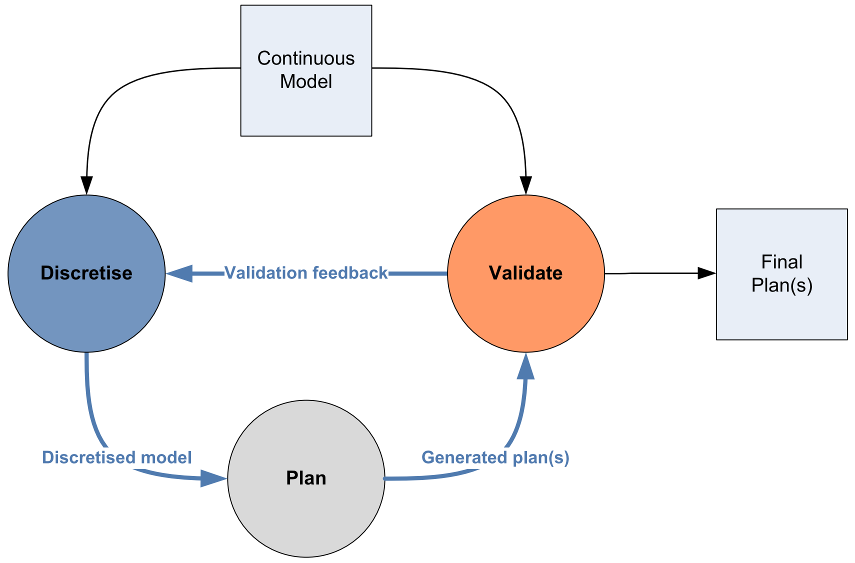


Fig. 1 The discretise and validate approach

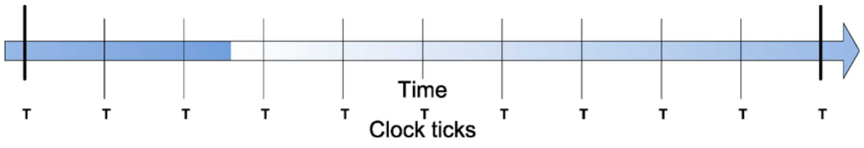


Fig. 2 Plan with discretised timeline

computes analytic solutions to the differential equations describing the system dynamics, its use is effective to prove the soundness of the discretised solutions. Moreover, the output of VAL is also used to determine whether a finer discretisation is required. The process loops until the validation returns a satisfying result.

In general, finding a suitable discretisation (i.e., a discretisation that gives rise to an approximation error which is less than a given threshold) is a hard and challenging task, which is out of the scope of the present paper. Therefore, in the following, we will always assume an initial coarse discretisation to be given: indeed, in many practical cases, it is easy to find out a suitable discretisation, or the discretisation itself is directly implied by the system specification (by means of hardware/mechanic limitations, etc.), as shown in the case studies (Sect. 5). However, in order to have an acceptable approximation of the continuous behaviour, a finer discretisation is often required. Since the finer is the discretisation, the bigger is the resulting state space, when the model is not trivial, the discretisation may result in a state space that is unavoidably huge, up to millions of states. To cope with this problem, the discretised problem is solved using a model checking based technique, which provides a very compact encoding of the state space, as described in Sect. 4.

Another important issue involved in the discretisation of a continuous domain is the discretisation of time. Indeed, in addition to continuous variables, also the dynamic behaviour of the model is discretised. When dealing with such a model, the planner should work on a discretisation of the timeline where events, actions, etc. can only happen (and are checked) at the beginning of each *clock tick*.

As shown in Fig. 2, where each clock tick is marked by T, in our framework we consider a discretised timeline using uniform discrete time steps. It is worth noting that the length of the clock tick is an important point, since it must be fine-tuned in order to avoid “hiding” or “stretching” happenings that may have a shorter length. Also in this case, the validation is used to detect if the clock tick must be shortened, for example by checking if all the events and processes captured by VAL are correctly handled also by the discretised model.

From a theoretical point of view, discretising the continuous dynamics of hybrid plants corresponds to moving from the *hybrid automata* semantics (which underlies the PDDL+ semantics) towards *finite state systems* semantics (which can be analysed through symbolic or explicit techniques such as model checking). From a practical point of view, this translation requires an encoding of the PDDL+ domain, as described in the rest of this section.

3.1 Discretisation of PDDL+ domains

One of the most important features provided by PDDL+ is the ability to model temporal behaviour through the use of processes which modify numeric values continuously. This allows modelling of domains involving continuous change. On the other hand, the discretised setting proposed in this paper requires a discrete modelling of time, where happenings are checked at the beginning of each discrete clock tick.

Note that the setting we are considering here goes beyond the capability of PDDL2.1, that is limited to time points associated with the start and end points of actions chosen by the planner. Instead, we consider time steps that do not depend on actions. In fact, we use a discretised timeline using uniform clock ticks that, if sufficiently short, provide a valid approximation of the continuous behaviour as expressed in PDDL+. Moreover, PDDL2.1 cannot handle exogenous events, while we want to take into account the PDDL+ events in order to model realistic problems.

To this aim, we designed and implemented the *PDDL+ to UPMurphi compiler*, which takes a PDDL+ domain/problem pair and generates a corresponding discretised model, that we refer to as *UPMurphi model*. The rest of this section describes the PDDL+ to UPMurphi model compilation process. In particular, for each main PDDL+ syntactic construct, we give the (possibly simplified) EBNF grammar, as reported in [29, 46], and the EBNF structure of the corresponding UPMurphi statements, highlighting similarities and differences. Examples of actual mappings are also given for each syntactic construct, using the well-known *Satellite* domain (described by Fox and Long [27]).

3.1.1 Notation

The UPMurphi tool and its input language, which is used to describe the discretised models, are deeply described in Sect. 4.

In the grammar notation, angle brackets are used to indicate the name of grammar nonterminal symbols, such as $\langle nonterminal \rangle$, and grammar rules have the form $\langle nonterminal \rangle ::= expansion$. It is worth noting that we indicate with the "upm_" prefix the nonterminals belonging to the UPMurphi grammar only (such as $\langle upm\_nonterminal \rangle$). All the other nonterminals are present, with the same meaning (but possibly with different expansions), in both PDDL+ and UPMurphi grammars.

We tried to maintain the highest possible similarity between the PDDL+ structures and the corresponding UPMurphi ones, to ease understanding the translation process and its correctness. However, since PDDL+ is a higher level language than the UPMurphi description language, the latter descriptions are naturally more complex than the former, and must explicitly implement some PDDL+ semantic rules. On the other hand, some trivial translation details will be omitted, such as the mapping of the PDDL+ (event, action) effects. Indeed, in this case, the expressive power of the corresponding sub-language (that includes arithmetic, assignments, function calls, etc.) is clearly comparable between PDDL+ and UPMurphi, so the translation description would reduce to a technical documentation of a cross-compilation process.

Moreover, both the grammars and the examples contained in this Section show a *plain* translation, whereas the implemented translator actually performs a variety of optimisations in order to create a model that is as much compact (and efficient) as possible. Some of these optimisations are automatic, whereas others can be manually enabled.

```
rule "time_passing"
    (true) ==> -- always executable
    begin
    apply_continuous_change();
    T := T + 1; -- let time flow
    end;
```

Fig. 3 `time_passing` rule in the UPMurphi model

3.1.2 Timeline discretisation

Each UPMurphi model contains a special rule called `time_passing`, which makes the time flow up to the next clock tick, as shown in Fig. 3. Moreover, the rule calls the `apply_ continuous_change` function, which is responsible of executing process updates, as discussed later in Sect. 3.1.7.

In this way, each clock tick in our plans is actually marked by the execution of a `time_passing`, as depicted in Fig. 4.

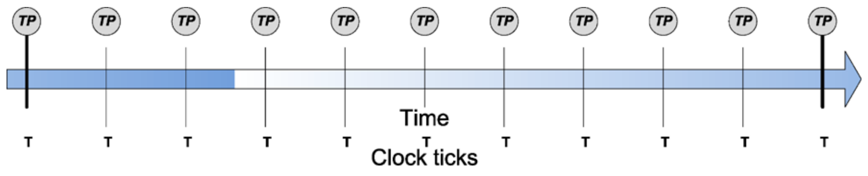
3.1.3 Types and objects

The *:types* keyword allows one to declare new types (in addition to the built-in ones), to be later used in the declaration of typed objects.

In the UPMurphi language, PDDL types become user-defined enumerative types, whose possible values are the names of all the declared objects of the corresponding type, as shown in Fig. 5.

As an example, Fig. 6 shows a fragment of the Satellite domain declaring the four types *satellite*, *direction*, *instrument* and *mode* and then using them to declare several objects. The UPMurphi equivalent of this fragment is also shown in Fig. 6: each type becomes an `Enum` construct having the corresponding objects as members.

Fig. 4 Plan timeline discretised through the `time_passing` rule



PDDL+ Grammar	UPMurphi Grammar
⟨types-def⟩ ::= **:types** *⟨typed-list (name)⟩* *⟨typed-list(x)⟩* ::= x^* | x^+ - *⟨type⟩⟨typed-list(x)⟩* *⟨type⟩* ::= *⟨name⟩* | (**either** *⟨type⟩*$^+$)	*⟨types-def⟩* ::= **type** *⟨typed-list (name)⟩* *⟨typed-list(x)⟩* ::= *x* **:***⟨type⟩* **;** *⟨type⟩* ::= **Enum {** *⟨upm_typed-obj-list (name)⟩* **};** | *⟨upm_fixed-type-def⟩* **;** *⟨upm_fixed-type-def⟩* ::= **real_type(***⟨integer⟩*, *⟨integer⟩***);** *⟨upm_typed-obj-list(x)⟩* ::= x^* | x^+ **,** *⟨upm_typed-obj-list(x)⟩*

Fig. 5 Comparison between PDDL+ and UPMurphi grammars for type definitions

PDDL+	UPMurphi

`(define (domain satellite)`
` (:requirements :typing)`
` (:types satellite`
` direction instrument mode)`
`...`
`(define (problem`
`satellite_prob)`
` (:domain satellite)`
` (:objects`
`  image1,spectrograph2,`
`   thermograph0 - mode`
`  star0,groundstation1,`
`   groundstation2,`
`   phenomenon3,phenomenon4,`
`  star5,phenomenon6`
`  - direction`
`  satellite0 - satellite`
`  instrument0,instrument1,`
`  instrument2 - instrument)`
`...`
	`type`
	` mode : Enum {`
	`  image1,`
	`  spectrograph2,`
	`  thermograph0};`
	` direction : Enum {`
	`  star0,`
	`  groundstation1,`
	`  groundstation2,`
	`  phenomenon3,`
	`  phenomenon4,`
	`  star5,`
	`  phenomenon6};`
	` satellite : Enum{`
	`  satellite0};`
	`  instrument : Enum {`
	`  instrument0,`
	`  instrument1,`
	`  instrument2};`

Fig. 6 Example of mapping for PDDL+ types

3.1.4 *Predicates and functions*

Model checking requires an explicit representation of the state in terms of *state variables*. On the other hand, the state of a PDDL+ domain is given by the current value of each function and predicate. Therefore, it is natural to map each PDDL+ function and predicate in an appropriately defined and typed (state) variable of the UPMurphi model.

In particular, each PDDL predicate (as declared by the *:predicates* keyword) is mapped on a global boolean variable of the UPMurphi model, which can be modified through appropriate (automatically generated) `get/set` helper functions. On the other hand, PDDL+ functions are mapped to real-typed variables, each with its own (private) type (*x*`_real_type`), which can be also *bounded*, if the user provides the translator with suitable bounds for the corresponding real number ranges.

Moreover, both predicates and functions can be *parametrised*. In this case, the translation process generates an array of boolean or real variables, indexed by the parameter type, as defined in Sect. 3.1.3. Predicates or functions with more than one parameter are mapped to multi-dimensional arrays.

PDDL+ Grammar	UPMurphi Grammar
⟨predicate-def⟩ ::= **(:predicates** *⟨atomic-formula-skeleton⟩*$^+$ *⟨atomic-formula-skeleton⟩* ::= *⟨predicate⟩ ⟨typed-list(variable)⟩* *⟨functions-def⟩* ::= **(:functions** *⟨atomic-function-skeleton⟩*$^+$ *⟨atomic-function-skeleton⟩* ::= *⟨function-symbol⟩* *⟨typed-list(variable)⟩*	*⟨predicate-def⟩* ::= *⟨predicate⟩* **:** *⟨upm_typed-arg-list (name)⟩** **;** *⟨upm_typed-arg-list(x)⟩* ::= **Array [** *x* **] of** *⟨upm_typed-arg-list(x)⟩* | **boolean** | *x***_real_type** *⟨function-def⟩* ::= *⟨function⟩* **:** *⟨upm_typed-arg-list (type_name)⟩** **;**

Fig. 7 Comparison between PDDL+ and UPMurphi grammars for predicate and function definitions

Figure 7 shows the grammar comparison between PDDL+ and UPMurphi for predicate and function definitions. It is worth noting how the PDDL+ parameter declarations, having the form *variable - type*, are mapped to subsequent `Array[type] of ...` expressions in the corresponding UPMurphi declaration. Thanks to the definition of types as enumerations discussed in Sect. 3.1.3, this actually creates an instance of the predicate/function for each possible combination of the given type values. The translation of predicate (and function) references within the domain is naturally implied by their definition.

As an example, in Fig. 8 we have a *on_board* predicate with two parameters of type *instrument* and *satellite*, respectively, and a *slew_time* function with two parameters both of type *direction*. The mapping of the three cited types has been already shown in Fig. 6. The UPMurphi equivalent declares a two-dimensional array of booleans to represent the *on_board* predicate, whose dimensions are indexed by the `instrument` and `satellite` enumerations, respectively. Thus, the first array dimension can have one of the values {`instrument0, instrument1, instrument2`}, whereas the second has `satellite0` as the only admissible value. On the other hand, the (not parametrised) *data_stored* function is declared as a real variable, with type `data-stored-real-type`.

3.1.5 *Actions*

PDDL+ actions are the only explicit elements of a plan, and are mapped on UPMurphi *rules*, which are used by the tool to perform the forward reachability analysis. In particular, the action precondition becomes the rule guard and the action effect is encoded in the rule code. Parametric actions are mapped into parametric rules using the `ruleset` construct.

Figure 9 compares the action grammars in PDDL+ and UPMurphi. It is worth noting that, since actions can modify the discrete component of the system state, possibly triggering an event in the same time instant, we place a call to the special procedure `event_check` after the effect code of all the action-rules, to check and trigger the event firing, as we will discuss in Sect. 3.1.6.

Actions are instantaneous, thus we allow more than one action to be performed at the same clock tick, i.e., time does

PDDL+	UPMurphi
(**define**	**var**
(*domain* satellite)	-- *predicates*
(:**predicates**	on_board : **Array** [instrument] **of Array** [satellite] **of boolean**;
(on_board ?i - instrument	canincrease : **boolean**;
?s - satellite)	-- *functions*
(canincrease))	slew_time : **Array** [direction] **of Array** [direction] **of** real_type;
(:**functions**	power_avail : **Array** [satellite] **of** slew_time_real_type;
(slew_time ?a ?b - direction)	data_stored : data_stored_real_type;
(power_avail ?s - satellite)	
(data-stored))	

Fig. 8 Example of mapping for PDDL+ predicates and functions

PDDL+ Grammar	UPMurphi Grammar
⟨*action-def*⟩ ::=	⟨*action-def*⟩ ::= **rule** ⟨*GD*⟩ ⇒
(**:action** ⟨*name*⟩ **:parameters**	**begin**
(⟨*typed list (variable)*⟩)	⟨*effect*⟩
:precondition ⟨*GD*⟩	**event_check();**
:effect ⟨*effect*⟩)	**end;**

Fig. 9 Comparison between PDDL+ and UPMurphi grammars for the action construct

PDDL+	UPMurphi
(:**action** heat	**ruleset** s:satellite **do**
:**parameters** (?s -	**rule** "heat"
satellite)	(energy[s] > 0 &
:**precondition** (*and*	heaterOff[s]) ==>
(> (energy ?s) 0)	**begin**
(heaterOff ?s))	set_heaterOn(s,**true**);
:**effect** (*and*	set_heaterOff(s,**false**);
(heaterOn ?s)	event_check();
(*not* (heaterOff ?s))	**end**;
)	**end**;
)	

Fig. 10 Example of mapping for a PDDL+ action

not advance due to the execution of an action-rule. As an example, Fig. 11 shows actions placed in the discretised plan timeline.

Figure 10 shows the action *heat* from the *Satellite* domain and the homonymous UPMurphi rule. Since the action has a parameter *s* of type *satellite*, the rule is enclosed in a corresponding `ruleset`. The rule guard (i.e., the action precondition) is given before the ==> symbol, and the rule body changes the values of two predicates using the corresponding `set` support functions.

3.1.6 Events

PDDL+ events are used to model exogenous changes in the domain. Differently from processes, events are instantaneous and may affect only discrete variables. However, an event may trigger *at the same time point* the activation of a process or a *cascading* sequence of other events [30]. In the mapping, this behaviour is restricted according to the *no moving target* definition given by [28], i.e., we suppose that no event can affect the parts of the state relevant to the preconditions of other events. This also implies that the execution of events is mutually-exclusive, and their order of execution is not relevant to the final outcome of the plan. Moreover, we impose that, at each *clock tick*, any event can be fired at most once, according to Howey et al. [37].

Figure 12 compares the event grammars in PDDL+ and UPMurphi. Basically, each event is mapped to a boolean function which returns `true` if the event was triggered.

As an example, Fig. 13 shows the UPMurphi mapping of the *freeze* event from the Satellite domain. The event parameters, preconditions and effects are trivially mapped in the corresponding function parameters and code.

The tool also generates an *event_check* procedure, that is called by the model whenever events may occur (e.g., after the execution of an action). The function tries to call all the possible instances of the available events (only one in the

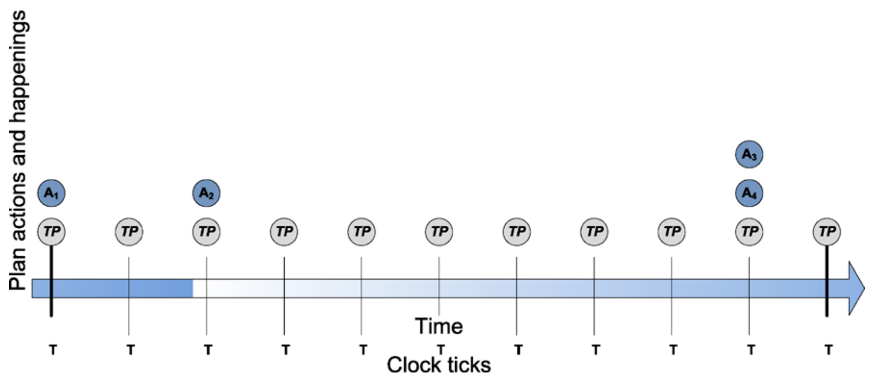


Fig. 11 Actions in the discretised plan timeline

example of Fig. 13), and continues until no event can be further triggered, avoiding multiple firing of the same event. Therefore, in the plan timeline, events are points (i.e., have zero duration) that appear as depicted in Fig. 14.

PDDL+ Grammar	UPMurphi Grammar

⟨*event-def*⟩ ::= **(:event** ⟨*name*⟩
:parameters (⟨*typed-list*⟩ **)**
:precondition ⟨*GD*⟩
:effect ⟨*effect*⟩ **)**
	⟨*event-def*⟩ ::= **function event_** ⟨*name*⟩
	(⟨*typed-list*⟩**)) : boolean ;**
	begin
	if(⟨*GD*⟩**) then** ⟨*effect*⟩
	return true;
	else return false;
	endif ;
	end ;

Fig. 12 Comparison between PDDL+ and UPMurphi grammars for the event construct

3.1.7 *Processes*

A PDDL+ *:process* is used to model *continuous* change and can be triggered by an event, an action or a durative action, based on its nature. In UPMurphi, processes become *procedures* that encode their behaviour. Process parameters are trivially mapped to procedure parameters.

Figure 15 shows a comparison between the procedure grammars in PDDL+ and UPMurphi. It is worth noting that in PDDL+ process effects contain *continuous update expressions* that refer to the special variable *#t* (time), and this is rendered in UPMurphi through the global constant `T` which defines the sampling time. Note that, in order to model such nontrivial continuous update expressions, the translator delegates them to external C language functions (see Sect. 4.3),

PDDL+

```
(:event freeze
 :parameters (?s - satellite
   ?i - instrument)
 :precondition (and
  (functional ?i)
  (onBoard ?s ?i)
  (<(temperature ?s)(safeValue ?i))
  )
 :effect (not (functional ?i))
)
```

UPMurphi

```
function event_freeze( s:satellite; i:instrument) : boolean;
begin
 if (functional[i] & onBoard[s][i] & (temperature[s] < safeValue[i])) then
  set_functional(i,false);
  return true;
 else return false;
 endif;

procedure event_check();
var
 freeze_event_triggered : Array[satellite] of Array[instrument] of boolean;
 event_triggered : boolean;
begin
 event_triggered := true;
 for s:satellite do
  for i:instrument do
   freeze_event_triggered[s,i]:=false;
  end;
 end;
 while(event_triggered) do
  event_triggered := false;
  for s:satellite do
   for i:instrument do
    if (!freeze_event_triggered[s,i]) then
    begin
     freeze_event_triggered[s,i] := event_freeze(s,i);
     event_triggered := event_triggered | freeze_event_triggered[s,i];
    endif;
   end;
  end;
 done;
end;
```

Fig. 13 Example of mapping for a PDDL+ event

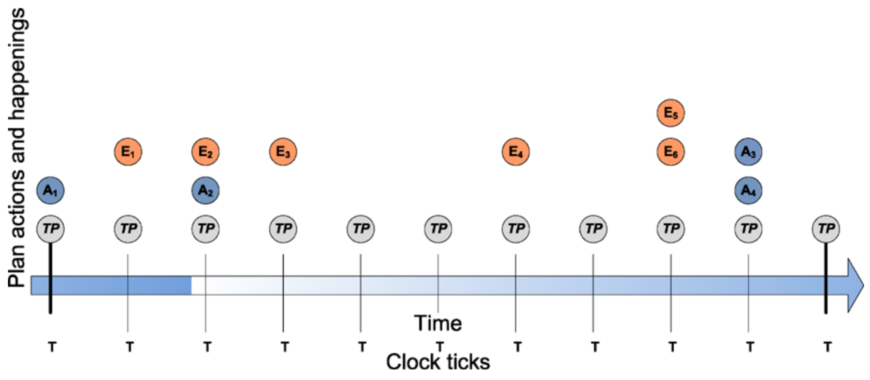


Fig. 14 Events in the discretised plan timeline

which allow to exploit the full power of the language and its math libraries to write the update code.

The process-procedures must be invoked at each clock tick, in order to approximate the continuous change in the discretised setting. Process updates may also trigger events, which in turn may activate other processes. To model this behaviour, the tool generates a support procedure, called `apply_continuous_change` which is executed *atomically* (w.r.t. a clock tick) within the `time_passing` rule (as described at the beginning of Sect. 3.1). Basically, this procedure works as follows: first, it calls all the process-procedures whose execution is currently enabled, then it checks if new events have been triggered and fires them. If some event firing occurred, the procedure checks the preconditions of all the processes to see if some new process has been enabled: if so, it repeats the whole sequence, otherwise it ends. Figure 17 shows how this technique allows to correctly represent the process behaviour across the plan timeline. In particular, in the figure, process P_1, after three clock ticks, triggers event E_3, which in turn activates process P_2.

PDDL+ Grammar	UPMurphi Grammar

⟨*process-def*⟩ ::= **(:process** ⟨*name*⟩
:parameters (⟨*typed-list*⟩ **)**
:precondition ⟨*GD*⟩
:effect ⟨*effect*⟩ **)**
	⟨*process-def*⟩ ::=
	procedure process_ ⟨*name*⟩
	(⟨*typed-list*⟩) **;**
	begin
	if(⟨*GD*⟩**) then** ⟨*effect*⟩
	endif ;
	end ;

Fig. 15 Comparison between PDDL+ and UPMurphi grammars for the process construct

PDDL+

```
(:process warming
 :parameters (?s - satellite)
 :precondition (and (heaterOn ?s)
   (> (energy ?s) 0))
 :effect (and
   (increase (temperature ?s)
     (* #t (heatRate ?s)))
   (decrease (energy ?s)
     (* #t (heaterConsumption ?s)))
     )
)
(:process cooling
 :parameters (?s - satellite)
 :precondition (inShade ?s)
 :effect (decrease (temperature ?s)
   (* #t (F(temperature ?s))))
)
```

UPMurphi

```
procedure process_warming(s:satellite);
begin
 if (heaterOn[s] & energy[s] > 0) then
  temperature[s] := temperature_update_warming(temperature[s],heaterRate[s
  ]);
  energy[s] := energy_update_warming(energy[s],heaterConsumption[s]);
 endif;
end;
procedure process_cooling(s:satellite);
begin
 if (inShade[s]) then
  temperature[s] := temperature_update_cooling(temperature[s]);
 endif;
end;
-- Extra procedure for process invocation
procedure apply_continuous_changes();
var
 process_warming_enabled : Array[satellite] of boolean;
 process_cooling_enabled : Array[satellite] of boolean;
 process_updated : boolean;
begin
 process_updated := false;
 for s:satellite do
 begin
  process_warming_enabled[s]=false;
  process_cooling_enabled[s]=false;
 end;
 while(true)
 for s:satellite do
   if (heaterOn[s] & energy[s] > 0 & process_warming_enabled[s]=false) then
   begin
    process_warming_enabled[s] := true;
    process_warming(s);
    process_updated := true;
   endif;
   if (inShade[s] & process_cooling_enabled[s]=false) then
   begin
    process_cooling_enabled[s] := true;
    process_cooling(s);
    process_updated := true;
   endif;
  end;
  if (!process_updated) then
  begin
   -- event checking code as in event_check
   -- breaks the loop if no event fired
  end;
 end;
end;
```

Fig. 16 Example of mapping for two PDDL+ processes

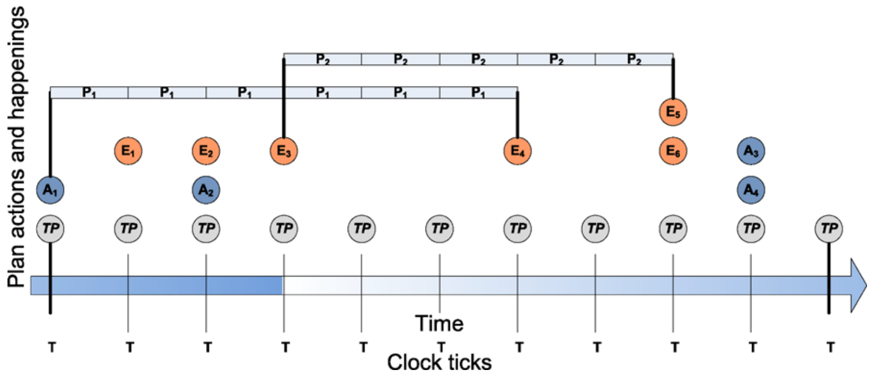


Fig. 17 Processes in the discretised plan timeline

Figure 16 shows the *warming* and *cooling* processes from the Satellite domain mapped in the UPMurphi model. The process preconditions become `if` guards on the effect code, which in turn delegates the continuous updates of *temperature* and *energy* to external functions.

It is worth noting that this translation setting makes also the modelling of concurrent processes easy. Indeed, in the given example both the warming and cooling processes could run in parallel and concurrently update the `temperature` variable. In particular, concurrent processes are treated as described by Fox and Long [29], that is all the continuous effects affecting the rate of change of a variable are combined by summing the contributions of the processes that are active in a clock tick.

3.1.8 Durative actions

The durative actions of PDDL+ offer an alternative to the continuous durative action model of PDDL 2.1 [28]. Basically, durative actions allow to represent periods of continuous change through a three-part structure, namely the *start-process-stop* model [29].

In this model, the continuous change is encoded in a process enabled by a special action, whose preconditions are the durative action's *at-start* conditions. The execution of such process can be later stopped by the effect of another special action, which becomes active after the durative action duration is reached, or by a failure event, which is triggered by the violation of the durative action's *over-all* condition.

The start-process-stop model can be easily implemented in UPMurphi using actions, events and processes, mapped as described in the previous sections. For example, Fig. 20 shows how a durative action is executed by means of a process P_3, that could possibly run in parallel with other processes, triggered by an action A_3 and stopped by another action A_4.

Figure 18 shows the durative action grammars in PDDL+ and UPMurphi. Note that, to better compare the two representations of the durative action, in the left column of Fig. 18 we also report a possible grammar for the PDDL+ durative action described according to the start-process-stop model. In particular, the ⟨*durative-action-def*⟩ is split into several sub-declarations, according to the start-process-stop model. We also need to split the conditions ⟨*da-GD*⟩ and effects ⟨*da-effect*⟩ to extract the start, end and overall conditions and effects, respectively. Moreover, according to the start-process-stop model, we introduce the variables ⟨*name*⟩**_clock**, ⟨*name*⟩**_clock_started** and **clock_count** which are used, respectively, to signal the activation of the durative action ⟨*name*⟩ (enabling the corresponding process), count the number of clock ticks passed from its activation (to calculate the time-dependant updates in the process), and count the total number of active durative actions.

In particular, *at start* conditions are mapped by the ⟨*upm_durative-action-def-start*⟩ nonterminal as a (parametric) action (see Sect. 3.1.5) that sets the `<name>_clock_started` variable to `true`, to enable the execution of a process which encodes the durative action *effect* and is mapped consequently (see Sect. 3.1.7) by the ⟨*upm_durative-action-def-process*⟩ nonterminal.

On the other hand, the *duration* is mapped in another action, which stops the process execution, through the ⟨*upm_durative-action-def-end*⟩ nonterminal. Finally, the failure of the *over all* conditions is checked by an event that is mapped (see Sect. 3.1.6) to the function `event_<name>_failure`, which terminates the durative action by resetting the `<name>_clock_started` variable.

Figure 19 shows an example of an UPMurphi mapping for a durative action extracted from the *Planetary Lander* case study described in Sect. 5.3. In the example, the durative action *observe* is mapped as follows: *at-start* conditions become the guard of a parametric (i.e., enclosed in one or more `ruleset` constructs) rule `observe_start`, which sets `observe_clock_started` to true to mark the beginning of the durative action. This enables the execution of the `process_observe` procedure, which maps the *effect* part of the durative action. After the given duration

PDDL+ Grammar	UPMurphi Grammar
⟨durative-action-def⟩ ::= (:durative-action ⟨name⟩ :parameters (⟨typed-list⟩) :duration ⟨duration-constraints⟩ :condition ⟨da-GD⟩ :effect ⟨da-effect⟩)	⟨durative-action-def⟩ ⟨upm_durative-action-def-start⟩ ⟨upm_durative-action-def-end⟩ ⟨upm_durative-action-def-overall⟩ ⟨upm_durative-action-def-process⟩
⟨action-def$_{at\ start}$⟩ ::= (:durative-action ⟨name⟩-start :parameters (⟨typed-list⟩) :precondition (and ⟨da-GD$_{at\ start}$⟩ (not(⟨name⟩-clock-started ⟨typed-list⟩))) :effect (and ⟨name⟩-clock-started ⟨typed-list⟩ (assign (⟨name⟩-clock) ⟨typed-list⟩ 0) (assign (⟨name⟩-duration) ⟨typed-list⟩ ⟨number⟩) (increase (⟨name⟩-clock-count) 1) ⟨da-effect$_{at\ start}$⟩))	⟨upm_durative-action-def-start⟩ ::= ⟨upm_structure-param-def⟩* rule ⟨upm_da-GD-start⟩ ⇒ begin ⟨name⟩_clock_started ⟨typed-statement(args)⟩* := true ; ⟨name⟩_clock ⟨typed-statement(args)⟩* := 0 ; clock_count := clock_count +1 ; ⟨upm_da-effect-start(x)⟩ end;
⟨action-def$_{at\ end}$⟩ ::= (:action ⟨name⟩-at end :parameters (⟨typed-list⟩) :precondition (and ⟨da-GD$_{at\ end}$⟩ (⟨name⟩-clock-started ⟨typed-list⟩) (= (⟨name⟩-clock ⟨typed-list⟩) (⟨name⟩-duration⟨typed-list⟩)) :effect (and not(⟨name⟩-clock-started ⟨typed-list⟩) (decrease (⟨name⟩-clock-count) 1) ⟨da-effect$_{at\ end}$⟩))	⟨upm_durative-action-def-end⟩ ::= ⟨upm_structure-param-def⟩* rule ⟨upm_da-GD-end⟩ ⇒ begin ⟨name⟩_clock_started ⟨typed-statement(args)⟩* := false ; ⟨name⟩_clock ⟨typed-statement(args)⟩* := 0 ; clock_count := clock_count -1 ; ⟨upm_da-effect-end⟩ end;
⟨event-def$_{overall}$⟩ ::= (:event ⟨name⟩-failure :parameters (⟨typed-list⟩) :precondition (and (⟨name⟩-clock-started ⟨typed-list⟩) not(= (⟨name⟩-clock ⟨typed-list⟩) (⟨name⟩-duration⟨typed-list⟩)) not(⟨da-GD$_{overall}$⟩)) :effect (and (assign (⟨name⟩-clock) ⟨typed-list⟩ + (⟨name⟩-duration⟨typed-list⟩ 1))	⟨upm_durative-action-def-overall⟩ ::= function event_⟨name⟩_failure ⟨upm_procedure-param-def⟩ : boolean ; begin if(not ⟨upm_da-GD-overall⟩) then ⟨name⟩_clock_started ⟨typed-statement(args)⟩* := false ; ⟨name⟩_clock ⟨typed-statement(args)⟩* := 0 ; clock_count := clock_count -1 ; return true; else return false; endif ; end ;
⟨process-def$_{overall}$⟩ ::= (:process ⟨name⟩ :parameters (⟨typed-list⟩) :precondition (⟨name⟩-clock-started ⟨typed-list⟩) :effect (increase (⟨name⟩-clock) ⟨typed-list⟩ (* #t 1)))	⟨upm_durative-action-def-process⟩ ::= procedure process_⟨name⟩ (⟨typed-list⟩) begin if(⟨name⟩_clock_started ⟨typed-statement(args)⟩*) then ⟨upm_da-effect-process⟩ endif ; end ;

Fig. 18 Comparison between PDDL+ and UPMurphi grammars for the durative-action construct

(`observationTime`), the process execution is stopped by the `observe_end` rule. Finally, if some *over all* condition fails, the checks in `event_observe_failure` immediately stop the process.

3.1.9 Problem

In PDDL+ a problem defines an instance of the domain, represented by an initial condition and a goal definition. These two elements simply become the `startstate` and the `goal` in the UPMurphi model, respectively. The corresponding grammars are shown in Fig. 21, and an example of a mapping is shown in Fig. 22.

4 The UPMurphi universal planner

The UPMurphi tool exploits model checking algorithms to perform universal planning. Model checking algorithms are typically divided in two categories: *symbolic* algorithms (e.g., [11]) and *explicit* algorithms (e.g., [24]). Symbolic algorithms have been successfully applied to classical planning [14, 23], however they do not work well on hybrid systems with nonlinear dynamics, due to the complexity of the state transition function. Therefore, explicit model checking performs better with the kind of systems we intend to approach. Also these algorithms are subject to the *state explosion problem*: however, the ability to build the system transition graph *on demand* and generate only the reachable states

PDDL+

```
(:durative-action observe
 :parameters (?r - recorder
    ?i - instrument
    ?o - observation)
 :duration (= ?duration (
 observationTime ?i ?o))
 :condition (and
   (at start (targetted ?i ?o))
   (over all (targetted ?i ?o))
   (over all (<= (data ?r)
     (capacity ?r))))
 :effect (increase (data ?r)
   (* #t (dataRate ?i)))
)
```

UPMurphi

```
-- START --
ruleset r:recorder do
 ruleset i:instrument do
  ruleset o:observation do
   rule "observe_start"
    (!observe_clock_started[r][i][o]) & (targetted[i][o]) ==>
   begin
    observe_clock_started[r][i][o] := true;
    observe_clock[r][i][o] := 0.0;
    clock_count := clock_count +1;
   end;
  end;
 end;
end;
-- END --
ruleset r:recorder do
 ruleset i:instrument do
  ruleset o:observation do
   rule "observe_end"
    (observe_clock_started[r][i][o]) &
    (observe_clock[r][i][o] = observationTime[i][o]) ==>
   begin
    observe_clock_started[r][i][o] := false;
    observe_clock[r][i][o] := 0.0;
    clock_count := clock_count -1;
   end;
  end;
 end;
end;
-- PROCESS --
procedure process_observe (r:recorder;i:instrument;o:observation);
begin
 if (observe_clock_started[r][i][o]) then
 data[r] := update_data(dataRate);
end;
-- FAILURE --
function event_observe_failure(r:recorder;i:instrument;o:observation) :
 Boolean;
begin
 if (observe_clock_started[r][i][o] &
 !(targetted[i][o] & data[r] <= capacity[r]) ) then
    observe_clock_started[r][i][o] := false;
    observe_clock[r][i][o] := 0.0;
    clock_count := clock_count -1;
    return true;
 else
    return false;
 endif;
end;
```

Fig. 19 Example of mapping for a PDDL+ durative action

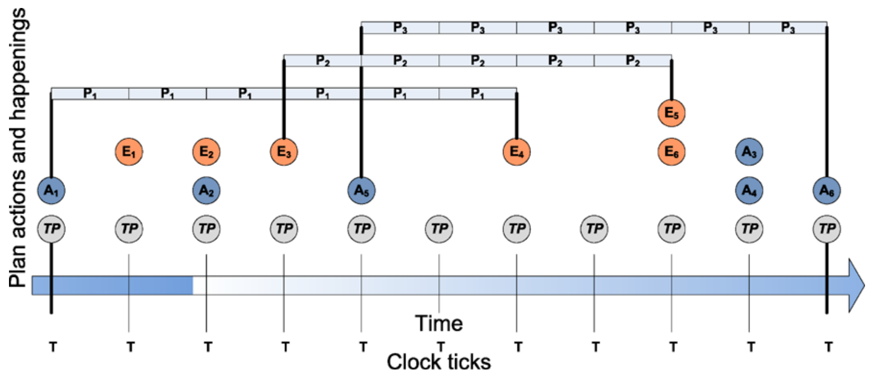


Fig. 20 Durative actions in the discretised plan timeline

of the system (through the *reachability analysis*), together with many space saving techniques (see, e.g., [18]), help to mitigate it.

Generally speaking, given a set E of *error states*, a model checker looks, via an exhaustive search, for a sequence of actions leading to an error state $e \in E$. If we look at error

PDDL+ Grammar	UPMurphi Grammar
⟨*init*⟩ ::= **(:init** ⟨*init-el**⟩ **)**	⟨*init*⟩ ::= **startstate** ⟨*upm_name*⟩ **begin** ⟨*init-el**⟩ **end ;**
⟨*goal*⟩ ::= **(:goal** ⟨*GD*⟩ **)**	⟨*goal*⟩ ::= **goal** ⟨*upm_name*⟩ ⟨*GD*⟩

Fig. 21 Comparison between PDDL+ and UPMurphi grammars for the init and goal constructs

PDDL+	UPMurphi
`(define (problem sat1)`	`startstate "start"`
`(:domain satellite)`	`begin`
`(...)`	` set_canincrease(true);`
`(:init`	` set_not_decreasing(true);`
` (canincrease)`	` set_not_increasing(true);`
` (not-decreasing)`	` set_not_flat(true);`
` (not-increasing)`	` set_powered(0);`
` (not-flat)`	` set_costof(image1,1);`
` (=(powered) 0)`	` set_costof(spectrograph2`
` (=(costof image1) 1)`	`  ,2.4);`
` (=(costof spectrograph2`	` set_costof(thermograph1`
` ) 2.4)`	`  ,3);`
` (=(costof thermograph1)`	` (...)`
`  3)`	`end;`
` (...)`	`goal "enjoy"`
`)`	` ((have_image[Phenomenon1`
`(:goal (and`	` ][thermograph0]) &`
` (have_image Phenomenon1`	` (have_image[star5][`
`  thermograph1)`	` spectrograph2]) &`
` (have_image Star5`	` (have_image[`
` spectrograph2)`	` GroundStation1][image1]`
` (have_image`	` &`
` GroundStation1 image1)`	` (have_image[`
` (have_image`	` GroundStation2][image1])`
` GroundStation2 image1)`	` );`
`))`	
`(...)`	
`)`	

Fig. 22 Example of mapping of PDDL+ init and goal constructs

states as *goal states*, and collect *all* the error sequences instead of the first one, we can use a model checker as a universal planner. This very simple fact allows one to use the model checking technology to automatically synthesise universal plans for complex systems.

4.1 Universal planning on finite state systems

In the following we formally define the universal planning problem on finite state systems, and we show an algorithm that solves this problem by means of a model checking derived algorithm.

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

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

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

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

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

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

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

Definition 4 (Reachable states) Let $s_I \in I$ be an initial state of the FSS $\mathcal{S} = (S, I, A, F)$. Then, we say that a state s' is *reachable* from s_I iff there exists a trajectory π in $\mathcal{S}$ such that $\pi_s(0) = s_I$ and $\pi_s(k) = s'$ for some $k >= 0$. We denote with *Reach(s)* the set of states reachable from s. Analogously, we denote with $Reach^{-1}(s)$ the set of states from which it is possible to reach the state s, that is $Reach^{-1}(s) = \{s' \in S | s \in Reach(s')\}$.

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

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

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

Algorithm 1 BUILD_GRAPH(UPP $\mathcal{P} = (\mathcal{S}, G, C, T)$)

```
1: let $\mathcal{S} \leftarrow (S, I, A, F)$
2: for all $s \in I$ do
3:   Enqueue($Q_S$, $s$)
4:   Insert(HT, $s$)
5:   if ($s \in G$) then
6:     Enqueue($Q_G$, $s$)
7:     HT[$s$].cost $\leftarrow 0$
8:   end if
9:   while (( $Q_S \neq \emptyset$) $\wedge$ (current_BFS_level $\leq$ T)) do
10:    $s \leftarrow$ Dequeue($Q_S$)
11:    for all $s' \in \{F(s, a) \mid (a) \in A\}$ do
12:      if ($s' \notin$ HT) then
13:        Insert(HT, $s'$)
14:        if ($s' \in G$) then
15:          Enqueue($Q_G$, $s'$)
16:          HT[$s'$].cost $\leftarrow 0$
17:        else
18:          Enqueue($Q_S$, $s'$)
19:        end if
20:      end if
21:      PT[$s'$] $\leftarrow$ PT[$s'$] $\cup \{s\}$;
22:    end for
23:  end while
24: end for
```

Algorithm 2 UPLAN_GENERATION

```
1: UPLAN $\leftarrow \emptyset$
2: $Q_S \leftarrow Q_G$ {this erases the previous content of $Q$}
3: while $Q_S \neq \emptyset$ do
4:   $s \leftarrow$Dequeue($Q_S$)
5:   prev_cost $\leftarrow$ HT[$s$].cost {0 if $s \in G$}
6:   for all ($\mathbf{s} \in$ PT[$s$]) {$\mathbf{s}$ is a predecessor of $s$} do
7:     local_cost $\leftarrow \min_{(a)\in A \mid F(\mathbf{s},a)=(s)} C(\mathbf{s}, a)$
8:     U $\leftarrow \{a \in A \mid F(\mathbf{s}, a) = s \wedge C(\mathbf{s}, a) =$ local_cost$\}$
9:     local_action $\leftarrow$ pick an action in U
10:    if (UPLAN[s]= $\emptyset$ $\vee$HT[s].cost $>$ prev_cost+local_cost) then
11:      UPLAN[s] $\leftarrow$ local_action
12:      HT[s].cost $\leftarrow$ prev_cost + local_cost
13:      Enqueue_in_Order($Q_S$, s)
14:    end if
15:  end for
16: end while
17: return UPLAN
```

4.2 Universal planning algorithm

Given a UPP, we solve it by means of an explicit algorithm, organised in two phases.

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

Note that, in the general theory of universal planning, the concept of start state is not present [55]. However, in the practice, the concept of reachable state implies such a start state. In other words, we need to start-up the universal planning with a set of start states, that we call a *start state cloud*. These states should be distributed in the system state space so that all the interesting states are reachable from at least one of them. However, a start state cloud can be also suitably prepared to concentrate the planning process on the most interesting state space regions, or to exclude hardly reachable states from the universal plan. Indeed, a complete universal plan could generally contain many rarely-used plans, whose computation requires however time and space. Therefore, an appropriate formulation of the start state cloud may help to minimise the universal plan generation effort and maximise its usefulness.

The procedure uses the hash table HT to store already visited states, while the queues Q_S and Q_G store the states to be expanded and the reached goal states (to be used in the next phase), respectively. This information is also used to detect and exploit trajectories intersections, so avoiding work duplication. Note that the computation of the successor states involves discretised values, i.e., continuous components of both s and s' in line 11 of Algorithm 1 are rounded according to the chosen discretisation. Finally, the predecessor table PT contains the immediate predecessors of each visited state. This structure is at the heart of the second phase of the algorithm, represented by the UPLAN_GENERATION procedure, whose pseudocode is given in Algorithm 2.

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

In particular, the check on line 10 of Algorithm 2, which updates the action associated to a state only if either no action has been defined yet or the current action leads to a better result, together with the ordered insertion in the queue Q_S, guarantee that the algorithm returns an *optimal solution* according to Definition 5.

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

4.3 The UPMurphi implementation

The UPMurphi tool is built on top of the CMurphi [12] model checker. In particular, UPMurphi consists of two main modules: the *PDDL+ to UPMurphi compiler*, illustrated in Sect. 3.1, which automatically discretises PDDL+ domains and problems and translates them into models written in the UPMurphi description language, and the *UP-*

Murphi engine, which is the core of the tool and implements the BUILD_GRAPH and UPLAN_GENERATION algorithms on the top of the CMurphi algorithms and data structures. Thanks to this, the universal planning algorithm presented in Sect. 4.2 can exploit all the CMurphi built-in state space optimisation techniques (such as bit compression [48], hash compaction [59], symmetry reduction [49], secondary memory storage and state space caching [18]) to handle large systems with huge state spaces.

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

In particular, the system state is defined in the UPMurphi through a set of *state variables*, which are suitably declared at the beginning of the model description. To this aim, the user can exploit the built-in data types provided by the tool or declare new user-defined types using data definition primitives such as arrays, structures and ranges.

The behavioural part of a UPMurphi model is a collection of *transition rules*. Each transition rule is a guarded command which consists of a condition (a boolean expression on global variables) and an action (a statement that can modify the variable values). It is also possible to write support functions and procedures and call them in the rule code to further modularise the specification.

The system initial states are declared through the `startstate` construct, which requires the user to suitably initialise all the model state variables. On the other hand, the `goal` construct is used to define the properties that a goal state must satisfy.

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

5 Case studies

In this section we show a number of PDDL+ case studies on which we applied the discretise and validate approach using UPMurphi to synthesise the universal plan.

We first present some experimental results for two benchmark domains, i.e., the continuous version of the well-known *Generator* domain as well as the *Cooling System* domain. Then, we present two significant case studies inspired by the real world specifications of complex systems, namely the *Planetary Lander* and the *Batch Chemical Plant*. Complete details on these case studies, including the UPMurphi code generated from the PDDL+ domains, can be found on the UPMurphi web site [40], together with more complete experiment results.

Table 1 Universal Plan statistics for the generator domain with time discretisation from 5.0 down to 1.0 seconds

Time discretisation (sec)	5.0	2.5	1.0
State space size	10^{15}	10^{16}	10^{18}
Reachable states	26,276	399,189	29,119,047
Generated plans	0	10,015	126,553
Total synthesis time	3.7	20.71	1,430.11
Valid	NO	NO	**YES**

5.1 Continuous generator domain

As a first example, we consider the continuous model of the *Generator* domain [35]. A generator is powered by a fuel tank with a limited capacity of 60 fuel units and consumes one fuel unit per second. During the generator activity (modelled by the *consume* durative action), two fuel tanks of 25 fuel units each can be used to refuel it (through the *refuel* durative action). The refuelling process has a variable duration (i.e., its duration must be decided by the planner) and is described by the Torricelli's law, which makes the system dynamics nonlinear. Moreover, the domain also involves concurrency, since the *consume* and *refuel* actions take place continuously and concurrently, and are modelled through continuous processes. The goal is to make the generator run for 100 seconds.

We defined a PDDL+ model of the generator and given it as input to UPMurphi. Table 1 summarises the results of the universal planning process: according to the discretise and validate approach, we first considered a time discretisation of 5.0 seconds, which resulted in an invalid solution, and then we iteratively refined it until we found a valid solution using a discretisation of 1.0 second. The final universal plan contains 126,553 plans, which is a small fraction of the near 30 million states that the system can reach, showing that there are many situations in which the goal cannot be achieved (i.e., a plan cannot be devised by the planner).

An example of VAL validation report for one of the generated plans is shown in Fig. 23. Here, we note that during the first 59 seconds the fuel level decreases linearly since no refuel action is performed. Then the generator is refuelled using tank 1 in the time interval [59, 84] and tank 2 in the time interval [75, 87] (thus in the time interval [75, 84] the generator is refuelled using both tanks). Finally, the generator uses the remaining fuel to complete the task.

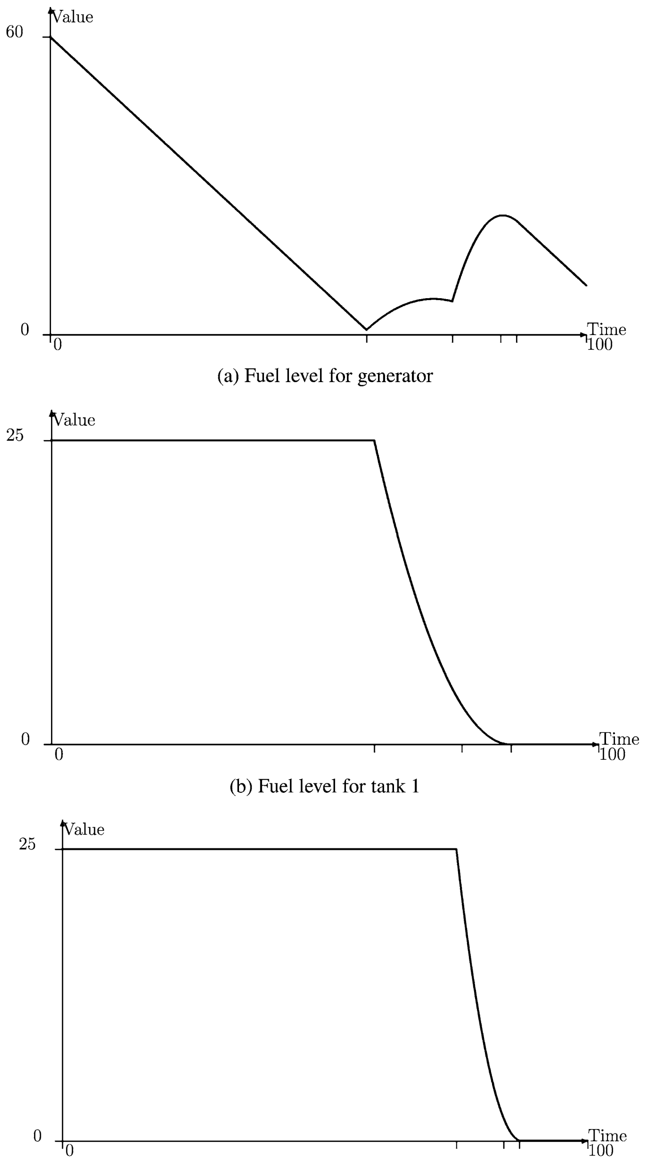


Fig. 23 Validation report for a single plan execution of the generator domain

5.2 Cooling system domain

In this case study we considered a classical open thermodynamic system that generates energy, part of which is lost in friction and hydraulic losses and transformed into heat. The system shown in Fig. 24 is composed by an external part, where a pump pours water continuously at a given rate into two hoses, and an internal part, composed by three water tanks which leak water at constant rate. The water passes through the pump and is poured by the hoses into two of the tanks at a time. We assume that hoses can be instantaneously repositioned on any tank. Moreover, we also consider that the water temperature raises when it passes through the pump, since it is heated by the pump engine.

Table 2 Cooling system constants

q	volume flow through the pump (m^3/s)	0.0006
P_s	brake power (kW)	0.095
μ	pump efficiency	0.05
c_p	specific heat capacity of the fluid (kJ/kg °C)	4.2
ρ	fluid density (kg/m^3)	1,000

The goal is to keep the amount of water in each of the three tanks above r_1, r_2 and r_3 liters, respectively, for 60 seconds. Moreover, we want the temperature T of the water passing through the pump to stay below 65 degrees.

Let v_i, with $i \in W = \{1, 2, 3\}$, denote the volume of water in Tank i and $v_i^{out} > 0$ denote the flow of water out of Tank i. Moreover, let v_j^{in}, with $j \in H = \{1, 2\}$, denote the flow of water introduced into the system through hose j, where $v_{total} = v_1^{in} + v_2^{in}$ denotes the water flow that passes through the pump.

The system is equipped with a controller that switches an hose to Tank i whenever $v_i \leq r_i$. The boolean variable $filling_{i,j}$ is true when tank i is filled through hose j. Therefore, the variation of the volume of the water in tank i is given by the following equation:

$$\frac{dv_i}{dt} = \begin{cases} v_j^{in} - v_i^{out} & \text{if } \exists j \in H | filling_{i,j} = 1 \\ v_i^{out} & \text{otherwise} \end{cases}$$

Finally, the water temperature rising can be computed as follows:

$$\frac{dT}{dt} = P_s(1 - \mu)/c_p q \rho$$

where the constant values are given in Table 2.

In the initial state of the system, the tanks are correctly filled and the water temperature respects the given constraint. However, the pump takes time to operate at full capacity, thus during the plan execution it increases its power by Δ_{power}, generating more heat. In this case, the planner may decide to increase the pump flow v_{total} by Δ_{rate} in order to mitigate the temperature rising (since the water flow cools down the pump), thus increasing also each v_i^{in} by $\frac{v_i^{in}}{v_{total}} \cdot \Delta_{rate}$, which may violate the constraint on the tank water level. On the other hand, the planner may leave v_{total} unchanged, risking to violate the maximum water temperature constraint.

Figure 25 shows the PDDL+ model of the cooling system domain, composed by a durative action *fill* that, given a tank `?t` and hose `?h`, starts the filling action of `?t` through `?h`, which is performed by the process *fill_tank*. Similarly, the tank leaking is modelled through process *leak_tank*. Note that the two processes may affect *concurrently* the same tank.

The event *over-range* is used to invalidate plans in which exists at least one tank where $v_i < r_i$ or $v_i > c_i$, while event

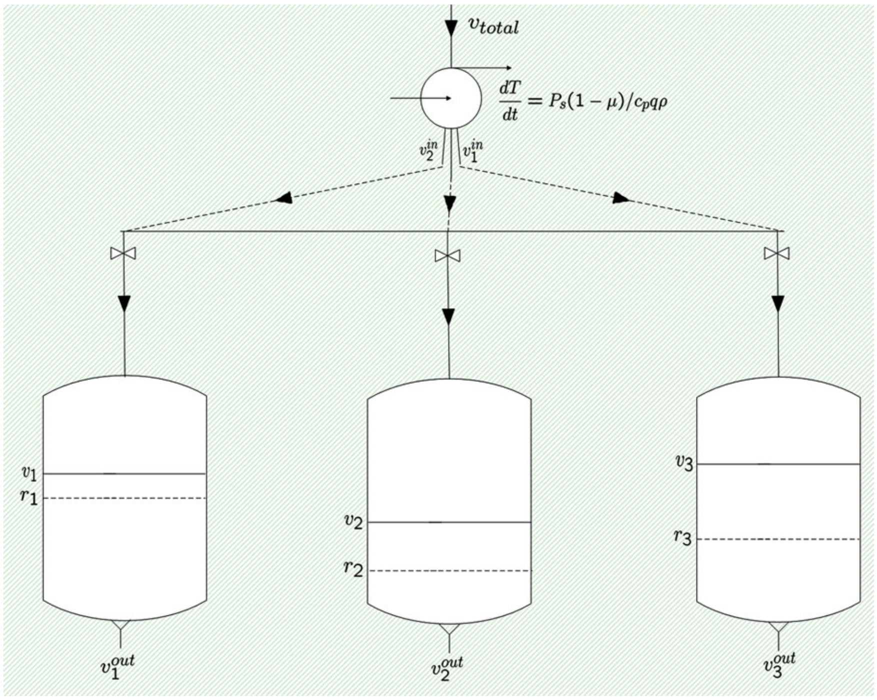


Fig. 24 A graphical representation of the cooling system domain

Table 3 Cooling system universal plan generation statistics with continuous variable rounding and time discretisation from 5.0 down to 1.0 seconds

	Time discretisation (sec)		
	5.0	2.5	1.0
State space size	10^{20}	$2 \cdot 10^{20}$	10^{21}
Reachable states	32	47,968	33,059,357
Generated plans	0	15,329	17,188,665
Total synthesis time	0.7	4.9	1,430.11
Valid	NO	NO	**YES**

overflow invalidates all plans for which a hose does not fill any tank. The event *pump-danger* is triggered when the temperature of the water passing through the pump is greater than the *danger_level*.

The process *system_activity* is used to measure the time elapsed since the beginning of the first *fill* action. Indeed, after a given amount of execution, the event *power-increasing* is triggered, increasing the pump's power consumption p_s by Δ_{power}. The effect of such event is to allow the execution of the action *increase_rate*, which in turn increases the pump flow v_{total} by Δ_{rate}, modifying the water flow in the hoses and changing the dynamics of the system from linear to nonlinear.

We used UPMurphi to generate the universal plan that controls the system activity for exactly one minute, starting from the initial condition shown in Fig. 26. The results in Table 3 show that, with a discretisation of 5.0, no plan was produced, whereas with 2.5 the generated plans were not valid. The valid universal plan was finally devised with a discretisation of 1.0.

In Fig. 27 we show an example of validation report for a complete plan starting from the start state given in Fig. 26. In particular, Figs. 27(a), 27(b), 27(c) describe the amount of water v_i in each tank i, Fig. 27(d) shows the evolution of v_j^{in} for each hose j, and Fig. 27(e) the water temperature rise.

5.3 The planetary lander

The planetary lander domain and problem are inspired by the specifications of the "Beagle 2" Mars probe [5], designed to operate on the Mars surface with tight resource constraints. In particular, we use the PDDL+ domain presented by [29], based on a simplified model of a solar-powered lander, the *Planetary Lander*. Table 4 shows an overview of the main domain elements with their preconditions.

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

The power needed to perform these operations comes from the probe solar panels. The energy generated by the panels (through the *generating* process) is influenced by the position of the sun, i.e., it is zero at night, rises until midday

PDDL+ durative action

```
(define (domain
cooling_system)
(:types tank hose pump)
(:predicates (fail)
  (system_start)
  (filling ?t - tank ?h -
  hose)
  (busy ?h - hose)
  (warning))
(:functions (v ?t -  tank)
  (v_out ?t - tank)
  (c ?t - tank)
  (v_in ?h - hose)
  (r ?t - tank)
  (system_counter)
  (temp ?p - pump)
  (p_s) (mu) (c_p) (q)
  (rho)
  (delta_rate)
  (delta_power)
  (plan_length)
  (danger_level))
(:durative-action fill
 :parameters (?t - tank ?h
  - hose)
 :duration (>= ?duration
 0)
 :condition  (at start (
 not(busy ?h)))
 :effect (and
  (at start (busy ?h))
  (at start (syste_start))
  (at start (filling ?t ?h
  ))
  (at end (not(filling ?t
  ?h)))
  (at end (not(busy ?h)))
 )
)
(:action increase_rate
 :parameters (?h1 ?h2 -
 hose)
 :precondition (warning)
 :effect (and
  (increase (v_in ?h1)
  (* 1000 (* (delta_rate)
  (/(v_in ?h1)(q)))))
  (increase (v_in ?h2)
  (* 1000 (* (delta_rate)
  (/(v_in ?h2)(q)))))
  (increase (q)
    (delta_rate)))
)
```

PDDL+ processes and events

```
(:event over-range
 :parameters (?t - tank)
 :precondition (or
  (< (v ?t) (r ?t))
  (> (v ?t) (c ?t)))
 :effect (fail)
)
(:event overflow
 :parameters (?h1 ?h2 -
 hose)
 :precondition (and
  (system_start)
  (or (not (busy ?h1))
  (not (busy ?h2))))
 :effect (fail)
)
(:event pump-danger
 :parameters (?p - pump)
 :precondition
  (> (temp ?p)
  (danger_level) )
 :effect (fail)
)
(:event power-increasing
 :parameters ()
 :precondition (and
  (>= (system_counter)
   (/ (plan_length) 2))
  (not(warning)) )
 :effect (and (warning)
  (increase (p_s) (
   delta_power)))
)
(:process fill_tank
 :parameters (?t - tank
  ?h - hose)
 :precondition (and
  (filling ?t ?h)
  (system_start))
 :effect (increase
  (v ?t) (* #t (v_in ?h)
  ))
)
(:process leak_tank
 :parameters (?t - tank)
 :precondition (and (
 system_start))
 :effect (decrease (v ?t
 )
  (* #t  (v_out ?t)))
)
(:process
system_activity
 :parameters ()
 :precondition (
 system_start)
 :effect (increase (
 system_counter)
  (* #t 1))
)
```

Fig. 25 The cooling system domain

PDDL+ problem

```
(define (problem
cooling_system_1)
(:domain cooling_system)
(:objects tank1 tank2 tank3
 - tank
 hose1 hose2 - hose pump1
  - pump)
(:init
  (= (v tank1) 0.2) ;liters
  (= (v tank2) 0.6)
  (= (v tank3) 0.9)
  (= (c tank1) 1.5) ;liters
  (= (c tank2) 1.5)
  (= (c tank3) 1.5)
  (= (r tank1) 0.1) ;liters
  (= (r tank2) 0.2)
  (= (r tank3) 0.2)
  (= (v_out tank1) 0.1) ;
  liters
  (= (v_out tank2) 0.3)
  (= (v_out tank3) 0.2)
  (= (v_in hose1) 0.3) ;
  liters
  (= (v_in hose2) 0.3)
  (= (system_counter) 0)
  (= (q) 0.0006) ; the sum
  of v_in in m^3
  (= (p_s) 0.13) ; kW
  (= (mu) 0.05)
  (= (c_p) 4.2 ) ; kJ/kg°C
  (= (rho) 1000 ) ; Kg/m^3
  (= (delta_power) 0.19) ;
  kW
  (= (delta_rate) 0.00005)
  ; in m^3
  (= (temp pump1) 60)
  (= (plan_length) 60)
  (= (danger_level) 65)
  (system_start)
  (not(fail))
)
  (:goal (and
    (not(fail))
    (= (system_counter) (
    plan_length))) )
  (:metric minimize (
  total-time)))
```

PDDL+ plan

```
000:(fill tank1 hose1)
 [001]
000:(fill tank1 hose2)
 [003]
001:(fill tank2 hose1)
 [059]
003:(fill tank3 hose2)
 [010]
013:(fill tank1 hose2)
 [005]
018:(fill tank3 hose2)
 [010]
028:(fill tank1 hose2)
 [005]
033:(fill tank3 hose2)
 [010]
043:(fill tank1 hose2 )
  [005]
048:(increase_rate
 hose1 hose2)
048:(fill tank3 hose2 )
  [008]
056:(fill tank1 hose2 )
  [004]
```

Fig. 26 The cooling system problem and one of the devised plans

and then returns to zero at dusk. Power coming from the solar panels is also used to charge a battery (the *charging* process), which is then discharged to give power to the lander (the *discharging* process) when the panels do not produce enough energy (e.g., at night). Moreover, the probe must always ensure a minimum battery level to keep its instruments warm.

The state of charge of the battery is therefore an important variable to monitor. Unfortunately, it follows a complex curve, since the charge/discharge process is nonlinear, and has several discontinuities, caused by the initiation and termination of the actions. Indeed, Table 5 shows the set of ordinary differential equations that are used to recalculate the values of the state variables *soc* (state of charge) and *supply* (solar panel generation). The symbols used in the equations have the following meaning: $s = soc$, $h = supply$, $d = demand$, $r = charge\_rate$, $sc = solar\_const$ and $D = daytime$. The equations clearly show the nonlinear dynamics of the system.

Obviously, the problem here is to find the best correct sequence of actions to achieve the probe goal in the shortest time possible, starting from any reasonable initial configuration. For sake of brevity, here we do not show the PDDL+ problem domain, which can be read in [29].

Table 4 A snapshot of the main PDDL+ domain elements for the planetary lander case study

Name	Type	Precondition
nightfall	Event	$(daytime >= dusktime)$ & day
daybreak	Event	$(daytime >= 0)$ & $\neg day$
charging	Process	$(supply >= demand)$ & day
discharging	Process	$(supply < demand)$
generating	Process	(day)
night-operation	Process	$(\neg day)$
fullprepare & prepareObs1 prepareObs2	Durative action	$\forall t \in ActionDuration$ $(battery >= safelevel)$
Observe1	Durative action	$readyForObs1$ & $\forall t \in ActionDuration$ $(battery >= safelevel)$
Observe2	Durative action	$readyForObs2$ & $\forall t \in ActionDuration$ $(battery >= safelevel)$

Table 5 PDDL+ events and processes for the planetary lander case study, with associated ordinary differential equations

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

5.3.1 Domain discretisation

The generation of the discretised UPMurphi model was initially performed rounding the continuous values to 0.5, whereas time was discretised to 0.5 time units (a Martian day is composed by 10 time units).

The start state cloud for the universal planning algorithm was selected by taking into account a set of reasonable configurations of the state variables *soc* and *daytime*. Note that it is realistic to consider *only* these parameters, since they define the environmental conditions to which the lander will be subject at the beginning of its mission. All the other domain parameters were fixed to the values inferred by looking at [5].

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

Table 6 Planetary lander universal plan generation statistics with continuous variable rounding and time discretisation to 0.5

State space size	10^{16}
Search depth limit	200 BFS levels
Reachable states	2,793,620
States to goal (generated plans)	699,595
Total synthesis time	82.32 seconds

Table 7 Planetary lander universal plan generation statistics with continuous variable rounding and time discretisation to 0.1

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

Therefore, the start state cloud will be defined as the set $\{(s, d) | s \in [90\%, 100\%] \wedge d \in [0, 8]\}$.

5.3.2 Universal planning

Given the domain variables and their ranges, we can easily calculate the state space size of the system, which is about of 10^{16} states. Thanks to the reachability analysis, UPMurphi generated an optimal solution for the universal planning problem, starting from the given start state cloud, and visiting only 2,793,620 reachable states, in 82.32 seconds on a 2.2 GHz CPU with 2 GB of RAM. The synthesis statistics are in Table 6.

However, the validation process revealed that the computed solutions were not valid for the original problem. Indeed, VAL did not validate the plans against the continuous domain. Thus, following the discretise and validate approach, the discretisation was iteratively refined, eventually rounding the continuous variables up to the first decimal and discretising the time to 0.1 time units, which produced a valid solution. With this approximation, the state space size of the system grows up to about 10^{24} states. Again, to generate the universal plan, UPMurphi visited only a small fraction of the state space, i.e., 31 million of reachable states, in less than 40 minutes. The synthesis statistics are in Table 7.

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

The generated solution contains more than 5 million plans, and thus it is able to bring to the goal more than

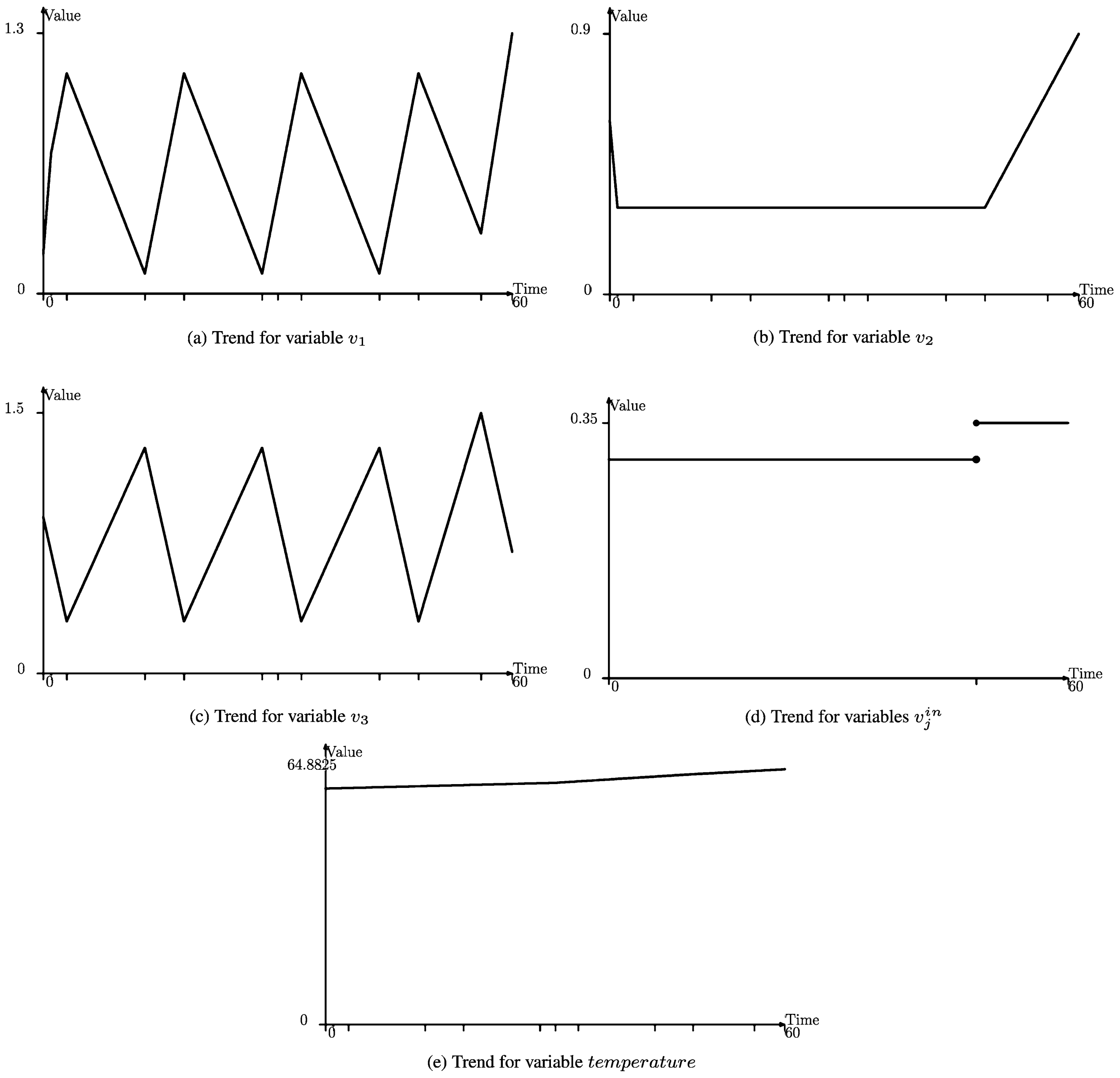


Fig. 27 Validation report for a single plan execution of the cooler system domain

Table 8 Normalised root mean squared error for variables *soc* and *supply* in the planetary lander case study, with continuous variable rounding and time discretisation to 0.1

	Min	Max	Avg
soc	0%	0.625392%	0.179329%
supply	0%	2.060061%	0.742575%

16% of the reachable states. Due to the exhaustive search performed by the tool, we can safely assert that, in the remaining 84% of the states, the lander could not complete its tasks and should therefore quit its mission or delay its initiation.

The solution passed the validation process (i.e., VAL validated the plans), thus the universal plan is correct and does not need any further discretisation refinement. However, to further estimate the precision of the plans, we compared the variable values computed by VAL during the validation process with the corresponding values output by the UPMurphi plan synthesis process, computing the normalised root mean squared error (NRMSE), as shown in Table 8. The NRMSE is at most 2% in all the generated plans for the nonlinear variable *soc*, at most 0.6% for nonlinear variable *supply* and always zero for the linear variable *daytime* (not shown in the table). Nevertheless, the average NRMSE is small: 0.179% for *soc* and 0.742% for *supply*, respectively.

5.4 The Batch chemical plant

The purpose of the batch chemical plant is to produce saline solution with a given concentration. If part of the product is not used, the plant can recycle it to restart another production cycle.

The plant (shown in Fig. 28) is composed of 7 tanks connected through a complex pipeline, whose flow is regulated by 26 valves and two pumps. In particular, tank 5 is provided with a heater, whereas tank 6 is connected to a condenser. Finally, tanks 6 and 7 are surrounded by a cooling circuit. A set of sensors provide information to the plant controller about the filling level of tanks 1, 2, 3 and 5, the pump pressure and the condenser status.

In the plant initial state, all the valves are closed, and the pumps, heaters and coolers are switched off. Tank 1 contains saline solution at a high concentration c_{high}, whereas tank 2 contains water.

If tank 1 does not contain enough solution, the plant enters the *startup phase*: water from tank 2 is moved to tank 3, where a suitable amount of salt is added manually to reach the required concentration, and finally pumped to tank 1. Note that tank 2 can be refilled with water at any time by opening the appropriate input valve.

When tanks 1 and 2 are appropriately filled, the plant can start the *production phase*. Tank 3 is partially filled with the solution from tank 1, which is then diluted using the water from tank 2 up to the requested concentration.

The resulting saline solution can be taken from the output valve of tank 3. If the product is not completely used, the plant recycles it in the next production cycle. To this aim, the solution in tank 3 is moved to tank 4 and then to tank 5. Here, the solution is boiled by the heater until it reaches the concentration c_{high}, and then moved to tank 7. The steam produced by this process is piped to the condenser that fills tank 6 with the resulting water. Finally, tanks 6 and 7 are cooled and their contents are pumped to tanks 2 and 1, respectively.

During the startup and production cycles the plant must obey some safety rules:

- pumps can be switched on only if all the valves in their pipeline are open,
- the heater cannot be switched on if tank 5 is empty, or the condenser is switched off, or if the valves involved in the heating/condensation process are closed,
- only two cooling circuits (including the one used by the condenser) can be switched on at the same time,
- tanks cannot be filled and emptied at the same time,
- the content of each tank must not exceed the corresponding capacity limitations [39], which are lower than the tank volume.

Table 9 Batch chemical plant constants

A_k	cross section of tank k
c_k	saline concentration in tank k
$c_{p,j}$	heat capacity of solution in tank j
$\Delta h_{vap,s}$	vaporisation enthalpy of solution s
h_k	filling level of tank k
$K_{k,l}$	volume flow from tank k to tank l
P_{heat}	heating power
$P_{cool,k}$	cooling power for tank k
ρ_j	density of solution in tank j
T_k	temperature of solution in tank k
$\dot{V}_i$	volume flow through valve i
$\dot{V}_{pk,l}$	volume flow through pump from tank k to tank l
$a_{k,l}$	section of pipe between tanks k and l
$H_{k,l}$	length of pipe between tanks k and l
$\zeta_{k,l}$	resistance of pipe between tanks k and l

The plant dynamics is described by Deparade [20] through a set of differential equations. In particular, given the constants and variables shown in Table 9, the following equations describe the variation of the filling level for tanks directly connected by a pipe with an open valve (and possibly a pump switched on) during the startup phase:

$$A_2\frac{dh_2}{dt} = \dot{V}_7$$

$$A_2\frac{dh_2}{dt} = -\dot{V}_9 = -K_{2,3}x; \quad x \in [1; x_{2,max}]$$

$$A_3\frac{dh_3}{dt} = \dot{V}_9 = K_{2,3}x; \quad x \in [1; x_{2,max}]$$

$$A_3\frac{dh_3}{dt} = -\dot{V}_{p3,1}$$

$$A_1\frac{dh_1}{dt} = \dot{V}_{p3,1}$$

whereas the following equations describe the same variation for tanks involved in the production phase:

$$A_1\frac{dh_1}{dt} = -\dot{V}_8 = -K_{1,3}x; \quad x \in [1; x_{1,max}]$$

$$A_3\frac{dh_3}{dt} = \dot{V}_8 = K_{1,3}x; \quad x \in [1; x_{1,max}]$$

$$A_2\frac{dh_2}{dt} = -\dot{V}_9 = K_{2,3}x; \quad x \in [1; x_{2,max}]$$

$$A_3\frac{dh_3}{dt} = \dot{V}_9 = K_{2,3}x; \quad x \in [1; x_{2,max}]$$

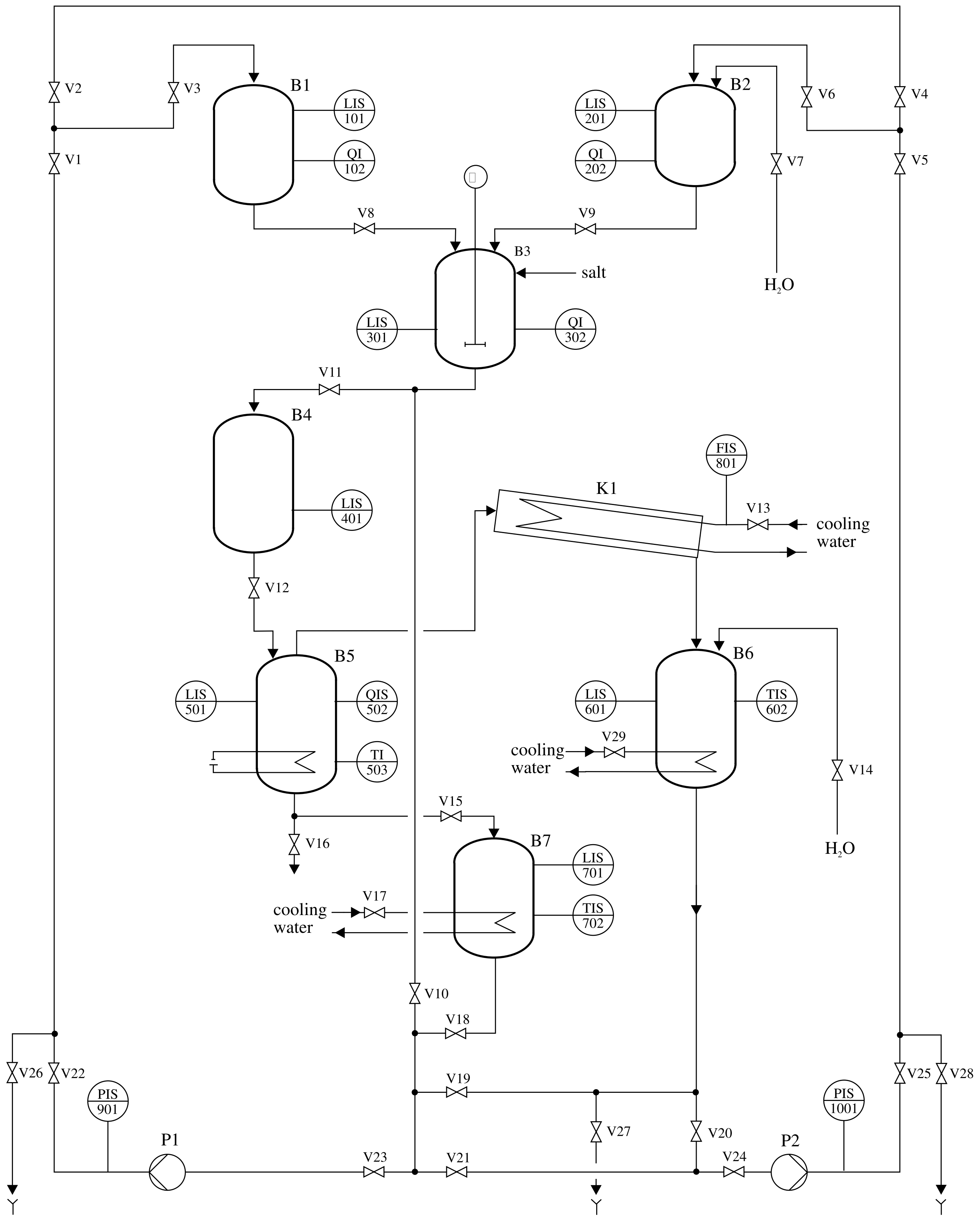


Fig. 28 Overall structure of the batch chemical plant

$$A_3\frac{dh_3}{dt} = -\dot{V}_{11} = -K_{3,4}x; \quad x \in [1; x_{3,max}]$$

$$A_4\frac{dh_4}{dt} = \dot{V}_{11} = K_{3,4}x; \quad x \in [1; x_{3,max}]$$

$$A_4\frac{dh_4}{dt} = -\dot{V}_{12} = -K_{4,5}x; \quad x \in [1; x_{4,max}]$$

$$A_5\frac{dh_5}{dt} = \dot{V}_{12} = K_{4,5}x; \quad x \in [1; x_{4,max}]$$

$$A_5\frac{dh_5}{dt} = -\dot{V}_{12} = -K_{5,7}x; \quad x \in [1; x_{5,max}]$$

$$A_7\frac{dh_7}{dt} = \dot{V}_{12} = K_{5,7}x; \quad x \in [1; x_{5,max}]$$

here, $K_{k,l} = \sqrt{\frac{2ga_{k,l}^2 H_{k,l}}{1+\zeta_{k,l}}}$ and $x = \sqrt{\frac{h_k}{H_{k,l}} + 1}$.

The variation of the filling level in tanks 5 and 6 is expressed differently, due to the effects of evaporation and condensation, respectively:

$$A_5\frac{dh_5}{dt} = \frac{\dot{m}_{vap}}{-\rho_{sol}}$$

$$A_6\frac{dh_6}{dt} = \frac{\dot{m}_{vap}}{\rho_w}$$

Equations are also given to calculate the variation of concentration and temperature in the tanks. The following equations compute the solution concentration in tanks 3 and 5:

$$A_3\left(c_3\frac{dh_3}{dt} + h_3\frac{dc_3}{dt}\right) = \dot{V}_9 c_2$$

$$A_5\left(c_5\frac{dh_5}{dt} + h_5\frac{dc_5}{dt}\right) = -\dot{m}_{vap} c_5$$

similarly, the temperature of tanks 5, 6, 7 is computed by the following equations:

$$c_{p,sol}\rho_{sol}A_5h_5\frac{dh_{T_5}}{dt} = P_{el}$$

$$T_5 c_{p,sol}\rho_{sol}A_5\frac{dh_{T_5}}{dt} = P_{el} - \dot{m}_{vap}(c_{p,sol}T_5 + \Delta h_{vap})$$

$$c_{p,lo}\rho_{lo}A_7h_7\frac{dT_7}{dt} = -P_{cool}$$

$$c_{p,w}\rho_w A_6h_6\frac{dT_6}{dt} = -P_{cool}$$

5.4.1 PDDL+ modelling

The most challenging and interesting aspect of the chemical plant specification is the production phase, so in the following we will focus only on the modelling of this phase.

This continuous, time-dependant domain is mainly modelled using processes, events and (flexible) durative actions.

```
; filling durative action (for tank 3)
(:durative-action B3_fill
:parameters ()
:duration (>= ?duration 0)
:condition (and
 (at start (not (V8))) (at start (= (B3_l) 0))
 (at start (>= (B1_l) 0)) (at start (not (V3)))
 (at start (not (V10))) (at start (not (V11)))
 (at start (not (B3_filled))) (at end (V8))
 (over all (>=(B1_l) 0)))
:effect (and
 (at start (B3_filling)) (at start (V8))
 (at end (not (V8))) (at end (B3_filled))
 (at end (not (B3_filling)))))
; filling process (for tank 3)
(:process B3_fill_process
:parameters ()
:precondition (B3_filling)
:effect (and
 (decrease (B1_l) (* #t (* (C_5_2) (sqrt (+ (/ (
 B1_l) (C_h_1_3)) 1 )))))
 (increase (B3_l) (* #t (* (C_5_2) (sqrt (+ (/ (
 B1_l) (C_h_1_3)) 1 )))))))
```

Fig. 29 Examples of durative actions and processes modelling the production phase of the batch chemical plant

```
(:process B3_fill_process
:parameters ()
:precondition (B3_filling)
:effect (and
 (decrease (B1_l)
   (* #t (* (C_5_2)( + (* -0.000415797 (* (B1_l) (
   B1_l) ) ) (+ (* (B1_l) 0.0424115 ) 1.00597 ))))
   )
 (increase (B3_l)
   (* #t (* (C_5_2)( + (* -0.000415797 (* (B1_l) (
   B1_l) ) ) (+ (* (B1_l) 0.0424115 ) 1.00597 ))))
   ))
)
```

Fig. 30 *B3_fill_process* with approximated square root

Indeed, Figs. 29, 31 and 32 show representative examples of such constructs extracted from the model (whose full source is available online in [19]), which contains a total of 59 predicates, 55 functions (14 of which represent real values), 19 events, 10 durative actions and 11 processes. In the figures, `B`*x*`_l`, `B`*x*`_c`, `B`*x*`_t` indicate the filling level, solution concentration and temperature for tank *x*, respectively, whereas `V`*y*, `P`*y* and `H`*y* indicate valve, pump and heater *y*, respectively. Finally, the value of a constant *k* taken from the problem specification is indicated with `C_`*k*.

In the following we describe the main elements of the PDDL+ model for the chemical plant production phase, highlighting their most interesting features. It is worth noting that the model has been written to adhere as much as possible to the formal specification given by Deparade [20]. However, to further check its correctness, we extracted from the universal plan generated by UPMurphi the single production policy corresponding to the initial conditions described by Kowalewski [39] and we verified that it was identical to the one (manually) devised by Kowalewski [39].

```
; pipeline flow failure (during B3 filling process)
(:event B3_flow_failure
:parameters ()
:precondition (and (or (V11) (V10)) (or (V8) (V9)))
:effect (not (correct_operation)))
; heater failure (on tank 5)
(:event H5_failure
:parameters ()
:precondition (or
  (and (H5) ( or (V12) (V15) (V16 )))
  (and (H5) (not(V13)))
  (and (H5) (not (>= (B5_l) (B5_l_safe)))))
:effect (not (correct_operation)))
; tank filling limit failure (on tank 3)
(:event B3_l_failure
:parameters ()
:precondition (or (< (B3_l) 0) (> (B3_l) (B3_l_max)
))
:effect (not (correct_operation)))
; pump (2) failure
(:event P2_failure
:parameters ()
:precondition (and (P2) (not(or
    (and (V25) (V28))
    (and (V25) (V5) (V6))
    (and (V25) (V5) (V4) (V2) (V1) (V3)))))
:effect (not (correct_operation)))
```

Fig. 31 Examples of failure events of the batch chemical plant

Production activities. The production activities, such as moving the solution from a tank to another, cool it down, etc., some of which can possibly be executed in parallel, are modelled using durative actions. However, the duration of these activities is not known *a priori*, thus the planner should determine the time point at which the tank capacity (or required concentration, or temperature) is reached. To achieve this, we use *duration inequalities* in the durative actions. On the other hand, continuous change to solution level, concentration and temperature in tanks are modelled through PDDL+ processes that update the corresponding model variables following the functions described by Deparade [20]. This modelling schema guarantees an immediate detection (i.e., triggering of failure events) of safety violations.

As an example, when tank 1 is nonempty, tank 3 is empty and some other conditions hold, the durative action *B3_fill* shown in Fig. 29 moves the solution from tank 1 to tank 3. The continuous update to the solution level in these tanks due to the durative action is performed by the process *B3_fill_process*, which is enabled by the durative action by setting to true the predicate *B3_filling*. The execution of this process may in turn trigger some events [28], e.g., *B3_l_failure* (shown in Fig. 31) that would invalidate the plan. At the end of the durative action (as chosen by the planner), *B3_filling* is set to false, and the filling process ends.

It is worth noting that the effects of *B3_fill_process* involve the calculation of a square root, which is currently not supported by PDDL+. Therefore, we have also created and tested an *approximated* model (available in [19]), where the square root is substituted by the second degree polynomial on the variable *B1_l* that best fits such function within

```
(:event production_end
:parameters ()
:precondition (and
 (B1_filled) (>= (B1_l) (B1_l_target_min))
 (< (B1_l) (B1_l_target_max))
 (= (B1_c) (B1_c_target)) (not(production_ended)))
:effect (and (production_complete)
 (production_ended)))
(:event production_success
:parameters ()
:precondition (and (not(success))
 (production_complete) (correct_operation)
 (not (or (V1) (V2) (V3) (V4) (V5) (V6) (V7) (V8) (
 V9) (V10) (V11) (V12) (V13) (V14) (V15) (V16) (
 V17) (V18) (V19) (V20) (V21) (V22) (V23) (V24) (
 V25) (V26) (V27) (V28) (V29) (P1) (P2))))
:effect (success))
```

Fig. 32 Cascading events triggering the goal of the batch chemical plant

the bounds deducible from the model dynamics. The corresponding approximated *B3_fill_process* is shown in Fig. 30.

Production events. The violation of one of the safety constraints listed in Sect. 5.4 should trigger an instantaneous change that invalidates the plan. Therefore, such failures have been modelled through PDDL+ events, whose effect is to falsify the invariant predicate *correct_operation*.

It is worth noting that, in the chemical plant model, discrete and continuous change are combined in the activation conditions of several events [37], making their checking more complex, but still very important since they may invalidate the plan [30]. As an example, event *H5_failure* in Fig. 31 shows the PDDL+ model of an exogenous event. Such event is activated when the heater is switched on (*H5* is true) and one of the valves 12, 15 or 16 is open (*or V12 V15 V16*), or valve 13 is closed (*not V3*), or the level of tank 5 is lower than the security level (*not (>= B5_l B5_l_safe)*).

Finally, the two events shown in Fig. 32 are used to trigger the end of the plan. In particular, event *production_end* is triggered when tank 1 contains a sufficient amount of solution with the required concentration, and its effect is to set the *production_complete* predicate to true. This, in turn, triggers a *cascading* event *production_success* that, if the plant has operated correctly (i.e., without violating any safety constraint) and all the valves and pumps have been correctly closed, sets the *success* predicate to true to indicate that the goal has been reached.

Production problem. The PDDL+ definition of the problem for the batch chemical plant production phase is quite straightforward. The domain is initialised by setting the function and predicate values to the ones obtained after the startup phase (see [20]), and the goal is to set the *success* predicate to true, minimising the *total-time*.

Table 10 Batch chemical plant startup phase universal plan generation statistics

State space size	10^{29}
Start state cloud size	81
Reachable states	3,092,112
States to goal (generated plans)	679,193
Synthesis time	530 sec
Peak of memory required	61 MB

Table 11 Batch chemical plant production phase universal plan generation statistics

State space size	10^{29}
Start state cloud size	23
Reachable states	29,968,861
States to goal (generated plans)	7,154,464
Synthesis time	6,319.8 sec
Peak of memory required	630 MB

5.4.2 Domain discretisation

We want to use UPMurphi to automatically perform universal planning on the startup and production phases, in order to generate a *set of policies* for the system.

To this aim, we fist discretise the PDDL+ model, as suggested by Brinksma and Mader [10], by rounding up the continuous variables up to the first decimal, and the time in steps of 10 seconds. Then, to generate the *start state clouds* used to initialise our universal planning engine, we proceed as follows.

As described in Sect. 5.4, the startup phase is triggered before a new production cycle if tank *B1* is empty or does not contain enough saline solution at concentration c_{high} (possibly recycled from the previous production phase). In this case, the startup phase must fill *B1* up to *B1_l_max*. Thus, the *start state cloud* for this phase considers all the values for *B1_l* in the range $[0, \mathit{B1\_l\_max}]$ with *B1_l_max* = 8 liters (as specified by Kowalewski [39]) and steps of 0.1 liters, i.e., 81 different start states.

On the other hand, the production phase, thanks to the startup postconditions, always starts working on a plant where *B1* and *B2* are completely filled. Here, the only parameter used to define the start state cloud is the amount of solution to be produced, that is *B3_l_target*. We vary this value in the range $[1.5, 3.7]$ liters with steps of 0.1, obtaining 23 different start states.

5.4.3 Universal planning

Table 10 shows the generation statistics for the startup phase universal plan. The plant state space is 10^{17}, however, starting from the given start state cloud, the planner found that only about 3 million of such states were actually reachable, and only for 22% of them is was possible to calculate a (optimal) policy to reach the goal.

On the other hand, the whole universal plan for the more complex production phase took about 6000 seconds to be generated, as shown in Table 11. Indeed, in this case there were about 30 million of reachable states (which are still sensibly less than the state space size), and for 24% of them UPMurphi was able to generate an optimal plan to the goal.

```
0.0: (B3_fill) [250]
260.0: (B3_dilution) [130]
400.0: (B4_fill) [290]
700.0: (B5_fill) [180]
890.0: (B5_evaporate) [750]
1650.0: (B7_fill) [130]
1790.0: (B7_cool) [270]
1800.0: (B6_cool) [160]
1970: (B2_fill) [120]
2070: (B1_fill) [80]
```

Fig. 33 A planned production policy for the batch chemical plant

The validation of the generated plans confirmed that the initial discretisation was fine enough to obtain correct results.

As an example, Fig. 33 shows one of the generated production policies, where *B3_level_target* = 3 liters. Figure 34 graphically shows the variation of *B1_l*, *B2_l*, *B3_l* and *B3_c*, respectively, as calculated by VAL during the validation of this plan. In particular, in the figure, letters A-F are used to indicate the time spans where the plant is performing particular tasks, i.e., A, B, C correspond to the activation of *B3_fill*, *B3_dilution* and *B4_fill*, respectively, D indicates the recycle phase, and E, F the activation of *B2_fill* and *B1_fill*, respectively.

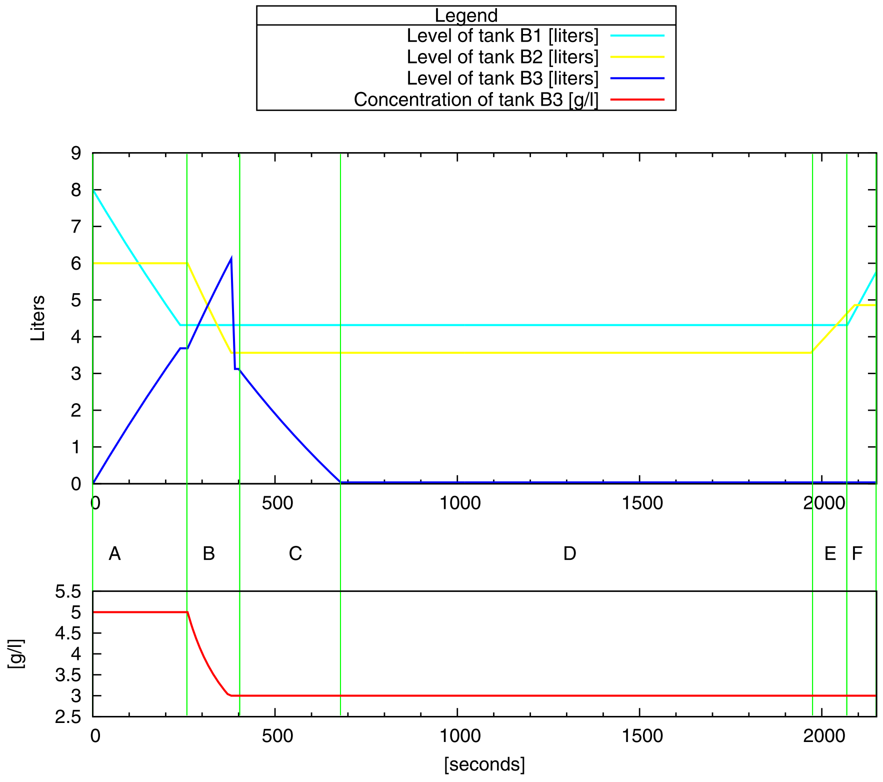
We see that the filling level of the first two tanks initially decreases due to the execution of the *B3_fill* (span A) and *B3_dilution* (span B) processes, respectively, while *B3* gets filled. On the other hand, the concentration *B3_c* remains stable on c_{max} during *B3_fill*, and rapidly decreases during the dilution (*B3_dilution*) up to c_{target}. Finally, part of the product is manually drained from *B3*, and the remaining solution is moved to other tanks (i.e., *B3_l* reaches zero, span C), where it is recycled (span D) and finally pumped back to *B1* (span E) and *B2* (span F).

6 Conclusions

In this paper we presented the UPMurphi tool, a universal planner based on the discretise and validate approach which is able to deal with PDDL+ domains modelling real world systems having a hybrid, nonlinear behaviour.

Indeed, planning with continuous nonlinear change is still an issue that is difficult to handle by the current state-of-the-art planners. However, thanks to an iteratively-refined

Fig. 34 Variation of filling levels computed by VAL for tanks *B1*, *B2*, *B3* and solution concentration in tank *B3* during the batch chemical plant production cycle described in Fig. 33



discretisation, our approach is able to relax a hybrid, nonlinear domain to obtain a finite state system, where universal planning can be performed by the UPMurphi tool. UPMurphi reads PDDL+ domain and problem specifications, generates their discretised model, and produces an optimal universal plan, which is validated against the original domain using VAL.

Moreover, UPMurphi exploits enhanced reachability analysis and state space compression algorithms, derived from model checking, to handle the huge number of states resulting from the approximation of the continuous part of the system dynamics.

The technique and the tool are being extensively experimented, and this paper presents a number of representative case studies, ranging from benchmark planning problems to realistic hybrid systems, namely a Martian lander and a chemical production plant, whose activities were successfully planned with UPMurphi. We feel that these case studies show that the discretise and validate approach, supported by UPMurphi, can be an effective and valuable universal planning technique for complex application domains.

The further development of UPMurphi will include the adoption of a new, more precise mathematical engine, to better process the complex dynamics of hybrid systems, and a tighter integration with VAL, to obtain more information from the validation process and use it as a guide to fully automatise the discretisation refinement.

Finally, since the complexity of hybrid systems could lead to state spaces which cannot be handled by UPMurphi, even with the help of reachability analysis and state space compression, we are developing new disk-based algorithms which, by making an intelligent use of secondary storage, could ensure a virtually unlimited space to store the state space, at the cost of a reasonable time penalty.

References

1. Aylett R, Soutter JK, Petley GJ, Chung PWH (1998) AI planning in a chemical plant domain. In: Prade H (ed) 13th European conference on artificial intelligence (ECAI), Brighton, UK, August 23–28. Wiley, New York, pp 622–626
2. Behrmann G, Cougnard A, David A, Fleury E, Larsen KG, Lime D (2007) UPPAAL-TIGA: Time for playing games! In: Damm W, Hermanns H (eds) 19th international conference on computer aided verification (CAV), July 3–7. Lecture notes in computer science, vol 4590. Springer, Berlin, pp 121–125
3. Bell KRW, Coles AJ, Coles AI, Fox M, Long D (2009) The role of AI planning as a decision support tool in power substation management. AI Commun 22(1):37–57
4. Bertoli P, Cimatti A, Pistore M, Roveri M, Traverso P (2001) MBP: a model based planner. In: Seventeenth international joint conference on artificial intelligence (IJCAI), workshop on planning under uncertainty and incomplete information (PRO-2). Seattle, Washington
5. Blake O, Bridges J, Chester E, Clemmet J, Hall S, Hannington M, Hurst S, Johnson G, Lewis S, Malin M, Morison I, Northey D, Pullan D, Rennie G, Richter L, Rothery D, Shaughnessy B, Sims M, Smith A, Townend M,

Waugh L (2004) Beagle2 Mars: mission report. Lander Operations Control Centre, National Space Centre, University of Leicester. http://www2.le.ac.uk/departments/physics/research/src/downloads/B2-Report.zip/at_download/file

6. Blondel VD, Tsitsiklis JN (2000) A survey of computational complexity results in systems and control. Automatica 36(9):1249–1274
7. Boddy MS, Johnson DP (2002) A new method for the global solution of large systems of continuous constraints. In: Global optimization and constraint satisfaction, first international workshop global constraint optimization and constraint satisfaction (COCOS). Lecture Notes in Computer Science, vol 2861. Springer, Berlin, pp 142–156
8. Bonet B, Geffner H (2001) Planning and control in artificial intelligence: A unifying perspective. Appl Intell 14:237–252. doi:10.1023/A:1011286518035
9. Borrelli F (2003) Constrained optimal control for hybrid systems. In: Constrained optimal control of linear and hybrid systems. Lecture notes in control and information sciences, vol 290. Springer, Berlin, pp 143–171. doi:10.1007/3-540-36225-8_8
10. Brinksma E, Mader A (2000) Verification and optimization of a PLC control schedule. In: 7th international SPIN workshop. Lecture notes in computer science, vol 1885. Springer, Stanford, pp 73–92
11. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
12. Cached Murphi Web Page (2006) http://www.dsi.uniroma1.it/~tronci/cached.murphi.html
13. Cimatti A, Roveri M, Traverso P (1998a) Strong planning in non-deterministic domains via model checking. In: Simmons Reid G, Manuela SS, Veloso M (eds) Fourth international conference on artificial intelligence planning systems (AIPS). AAAI Press, Pittsburgh, pp 36–43
14. Cimatti A, Pistore M, Roveri M, Traverso P (2003) Weak, strong, and strong cyclic planning via symbolic model checking. Artif Intell 147(1):35–84
15. Cimatti R, Roveri M, Traverso P (1998b) Automatic OBDD-based generation of universal plans in non-deterministic domains. In: Fifteenth national conference on artificial intelligence and tenth innovative applications of artificial intelligence conference (AAAI). AAAI Press / MIT Press, Madison, pp 875–881
16. Coles AI, Fox M, Long D, Smith AJ (2007) Planning with problems requiring temporal coordination. In: Twenty-second AAAI conference on artificial intelligence. AAAI Press, Vancouver, pp 415–420
17. Coles AJ, Coles AI, Fox M, Long D (2009) Temporal planning in domains with linear processes. In: Boutilier C (ed) 21st international joint conference on artificial intelligence (IJCAI), Pasadena, California, USA. pp 1671–1676
18. Della Penna G, Intrigila B, Melatti I, Tronci E, Venturini Zilli M (2004) Exploiting transition locality in automatic verification of finite state concurrent systems. Int J Softw Tools Technol Transf 6(4):320–341
19. Della Penna G, Intrigila B, Magazzeni D, Mercorio F (2009) Batch chemical plant PDDL+ model. http://www.di.univaq.it/gdellape/lamoka/go/?page=chemical
20. Deparade A (1999) A switched continuous model of VHS case study 1, draft. University of Dortmund. http://www-verimag.imag.fr/VHS/year1/cs11c.ps
21. Edelkamp S (2002) Mixed propositional and numeric planning in the model checking integrated planning system. In: Fox M, Coddington AM (eds) AIPS workshop on planning for temporal domains, Toulous, France, pp 47–55
22. Edelkamp S (2003) Taming numbers and durations in the model checking integrated planning system. J Artif Intell Res 20:195–238
23. Edelkamp S, Helmert M (2001) MIPS: The model-checking integrated planning system. AI Mag 22(3):67–72
24. Edelkamp S, Lafuente AL, Leue S (2001) Directed explicit model checking with HSF-SPIN. In: 8th international SPIN workshop on model checking of software. Springer, New York, pp 57–79
25. Edelkamp S, Jabbar S, Nazih M (2006) Large-scale optimal PDDL3 planning with MIPS-XXL. In: 5th international planning competition booklet. International conference on automated planning and scheduling, The English Lake District, Cumbria, UK, pp 28–31
26. Fourman M (2000) Propositional planning. In: Fifth international conference on artificial intelligence planning and scheduling—workshop on model theoretic approaches to planning (AIPS), Breckenridge, CO, USA, pp 10–17
27. Fox M, Long D (2002) PDDL+: modelling continuous time-dependent effects. In: 3rd international NASA workshop on planning and scheduling for space, Houston, Texas
28. Fox M, Long D (2003) PDDL2.1: An extension to PDDL for expressing temporal planning domains. J Artif Intell Res 20:61–124
29. Fox M, Long D (2006) Modelling mixed discrete-continuous domains for planning. J Artif Intell Res 27:235–297
30. Fox M, Howey R, Long D (2005) Validating plans in the context of processes and exogenous events. In: The twentieth national conference on artificial intelligence and the seventeenth innovative applications of artificial intelligence conference (AAAI/IAAI). AAAI Press / The MIT Press, Pittsburgh, pp 1151–1156
31. Gerevini A, Saetti A, Serina I (2004) Planning with numerical expressions in LPG. In: de Mántaras RL, Saitta L (eds) 16th European conference on artificial intelligence (ECAI). IOS Press, Valencia, pp 667–671
32. Giunchiglia F, Traverso P (2000) Planning as model checking. In: 5th European conference on planning: recent advances in AI planning. Springer, London, pp 1–20
33. Herrero J, Berlanga A, Molina J, Casar J (2005) Methods for operations planning in airport decision support systems. Appl Intell 22:183–206. doi:10.1007/s10791-005-6618-z
34. Holldobler S, Stor H (2000) Solving the entailment problem in the fluent calculus using binary decision diagrams. In: Chien S, Kambhampati S, Knoblock CA (eds) Fifth international conference on artificial intelligence planning systems (AIPS). AAAI Press, Breckenridge, pp 32–39
35. Howey R, Long D (2003) VAL's progress: the automatic validation tool for PDDL2.1 used in the international planning competition. Strathprints: The University of Strathclyde Institutional Repository [http://eprintscdlrstrathacuk/perl/oai2] (United Kingdom)
36. Howey R, Long D, Fox M (2004a) VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: 16th IEEE international conference on tools with artificial intelligence (ICTAI). IEEE Computer Society, Boca Raton, pp 294–301
37. Howey R, Long D, Fox M (2004b) Validating plans with exogenous events. In: 23rd workshop of the UK planning and scheduling special interest group. University College Cork, Ireland, pp 78–87
38. Jensen R, Veloso M (1999) OBDD-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. Artifi Intell Today, Recent Trends Dev 1600:213–248
39. Kowalewski S (1998) Description of VHS case study 1: "Experimental Batch Plant". http://astwww.chemietechnik.uni-dortmund.de/~vhs/cs1descr.zip
40. L'Aquila Model Checking Group (2010) UPMurphi web page. http://www.di.univaq.it/gdellape/lamoka/upmurphi
41. Léauté T, Williams BC (2005) Coordinating agile systems through the model-based execution of temporal plans. In: Veloso MM, Kambhampati S (eds) Twentieth national conference on artificial

intelligence and the seventeenth innovative applications of artificial intelligence conference (AAAI/IAAI). AAAI Press / The MIT Press, Pittsburgh, pp 114–120
42. Li HX, Williams BC (2008) Generative planning for hybrid systems based on flow tubes. In: Rintanen J, Nebel B, Beck JC, Hansen EA (eds) Eighteenth international conference on automated planning and scheduling (ICAPS). AAAI Press, Sydney, pp 206–213
43. Martin M, Geffner H (2004) Learning generalized policies from planning examples using concept languages. Appl Intell 20(1):9–19
44. McDermott D (2000) The 1998 AI planning systems competition. AI Mag 21(2):33–55
45. McDermott D (2003) Reasoning about autonomous processes in an estimated regression planner. In: Giunchiglia E, Muscettola N, Nau DS (eds) Thirteenth international conference on automated planning and scheduling (ICAPS). AAAI Press, Trento, pp 143–152
46. McDermott D the AIPS1998 planning competition committee (1998). PDDL: the planning domain definition language. Tech. rep., available at: www.cs.yale.edu/homes/dvm
47. Molineaux M, Klenk M, Aha DW (2010) Planning in dynamic environments: Extending HTNs with nonlinear continuous effects. In: Fox M, Poole D (eds) Twenty-fourth AAAI conference on artificial intelligence (AAAI). AAAI Press, Atlanta, pp 1115–1120
48. Murphi Web Page (2004) http://sprout.stanford.edu/dill/murphi.html
49. Norris-Ip C, Dill DL (1993) Better verification through symmetry. In: Agnew D, Claesen LJM, Camposano R (eds) Computer hardware description languages and their applications, proceedings of the 11th IFIP WG10.2 international conference on computer hardware description languages and their applications—CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, pp 97–111
50. Penberthy S, Weld D (1994) Temporal planning with continuous change. In: Twelfth national conference on artificial intelligence (AAAI). American Association for Artificial Intelligence, Menlo Park, pp 1010–1015
51. Reddy SY, Iatauro MJ, Kürklü E, Boyce ME, Frank JD, Jónsson AK (2008) Planning and monitoring solar array operations on the ISS. In: Eighteenth international conference on automated planning and scheduling (ICAPS), scheduling and planning applications workshop (SPARK), Sydney, Australia
52. Sapena O, Onaindía E (2008) Planning in highly dynamic environments: an anytime approach for planning under time constraints. Appl Intell 29:90–109. doi:10.1007/s10489-007-0083-x
53. Schmid U (2003) Inductive synthesis of functional programs: universal planning, folding of finite programs, and schema abstraction by analogical reasoning. Springer, New York
54. Schmid U, Wysotzki F (2000) Applying inductive program synthesis to macro learning. In: Chien S, Kambhampati S, Knoblock CA (eds) Fifth international conference on artificial intelligence planning systems (AIPS). AAAI Press, Breckenridge, pp 371–378
55. Schoppers M (1987) Universal plans of reactive robots in unpredictable environments. In: McDermott JP (ed) 10th international joint conference on artificial intelligence (IJCAI). Morgan Kaufmann, Milan, pp 1039–1046
56. Schuster HG (1988) Deterministic chaos: an introduction. Weinheim Physik
57. Shin JA, Davis E (2005) Processes and continuous change in a SAT-based planner. Artif Intell 166(1–2):194–253
58. Sontag E (1996) Interconnected automata and linear systems: a theoretical framework in discrete-time. In: Alur R, Henzinger T, Sontag E (eds) Hybrid systems III. Lecture notes in computer science, vol 1066. Springer, Berlin, pp 436–448
59. Stern U, Dill D (1995) Improved probabilistic verification by hash compaction. In: Camurati P, Eveking H (eds) Correct hardware design and verification methods. Lecture notes in computer science, vol 987. Springer, Berlin, pp 206–224
60. Wilson E, Karr C, Bennett J (2004) An adaptive, intelligent control system for slag foaming. Appl Intell 20:165–177. doi:10.1023/B:APIN.0000013338.39348.46

Giuseppe Della Penna obtained the master degree in Computer Science in 1998 at the University of L'Aquila, Italy and the Ph.D. in Computer Science in 2002 at the University of Rome "La Sapienza". He is currently a Researcher in the Computer Science Department of the University of L'Aquila, where he leads the *Model Checking and Applications* research group. His research interests include formal methods applied to web engineering, visual languages, and software systems control and verification. In these contexts, he published several peer-reviewed articles in international journals and conferences.

Daniele Magazzeni received the master degree in Computer Science in 2005 and the Ph.D. degree in Computer Science in 2009 at the University of L'Aquila, Italy. He is currently a Research Fellow in the Department of Sciences at the University of Chieti-Pescara, Italy. His research interests include formal methods and artificial intelligence, with particular focus on planning for temporal and metric domains, continuous planning, heuristic search and policy learning.

Fabio Mercorio is a third year Ph.D. student in Computer Science at the University of L'Aquila, Italy, where he received the master degree in Computer Science in 2008. His research interests include formal methods and artificial intelligence, with particular focus on continuous planning and the analysis and control of complex nondeterministic systems.