A CASP-Based Approach to PDDL+ Planning

Marcello Balduccini
Drexel University
marcello.balduccini@gmail.com

Daniele Magazzeni
King's College London
daniele.magazzeni@kcl.ac.uk

Marco Maratea
University of Genoa
marco@dibris.unige.it

Abstract

PDDL+ is an extension of PDDL that makes it possible to model planning domains with mixed discrete-continuous dynamics. In this paper we present a new approach to PDDL+ planning based on the paradigm of Constraint Answer Set Programming (CASP), an extension of Answer Set Programming that supports efficient reasoning on numerical constraints. We provide an encoding of PDDL+ models into CASP problems. The encoding can handle non-linear hybrid domains, and represents a solid basis for applying logic programming to PDDL+ planning. As a case study, we consider an implementation of our approach based on CASP solver EZCSP and present very promising results on a set of PDDL+ benchmark problems.

1 Introduction

Planning in hybrid domains is a challenging problem that has found increasing attention in the planning community, mainly motivated by the need to model real-world domains. Indeed, in addition to classical planning, hybrid domains allow for modeling continuous behavior with continuous variables that evolve over time. PDDL+ (Fox and Long 2006) is the extension of PDDL that allows for modelling domains with mixed discrete-continuous dynamics, through continuous processes and exogenous events.

Various techniques and tools have been proposed to deal with hybrid domains (Penberthy and Weld 1994; McDermott 2003; Li and Williams 2008; Coles et al. 2012; Shin and Davis 2005). More recent works include (Bryce et al. 2015), which presents an approach based on Satisfiability Modulo Theory (SMT) and restricted to a subset of the PDDL+ features, and (Bogomolov et al. 2014; Bogomolov et al. 2015) that combines hybrid system model checking and planning, but is only limited to proving plan non-existence.

To date, the only approach able to handle the full PDDL+ is the *discretise and validate* approach implemented in UPMurphi (Della Penna et al. 2009). There, the continuous model is discretised and forward search is used to find a solution, which is then validated against the continuous model using VAL (Fox, Howey, and Long 2004). If the solution is not valid, the discretisation is refined and the process iterates. The main drawback of UPMurphi, though, is the lack of heuristics that strongly limits its scalability, and hence its applicability to real case studies.

This motivates the need for finding new ways to handle PDDL+. To this aim, in this paper we present a new approach to PDDL+ planning based on Constraint Answer Set Programming (CASP) (Baselice, Bonatti, and Gelfond 2005), an extension of Answer Set Programming (ASP) (Gelfond and Lifschitz 1991) supporting efficient reasoning on numerical constraints. We provide an encoding of PDDL+ models into CASP problems, which can handle linear and non-linear domains, and can deal with PDDL+ processes and events. This contribution represents a solid basis for applying logic programming to PDDL+ planning, and opens up the use of CASP solvers for planning in hybrid domains.

We describe how the different components of a PDDL+ domain can be encoded into CASP. In our encoding, continuous invariants are checked at discretised timepoints, and following the discretise and validate approach (Della Penna et al. 2009), VAL is used to check whether the found solutions are valid or whether more timepoints need to be considered. As a case study, we use the CASP solver EZCSP (Balduccini 2009). Experiments performed on PDDL+ benchmarks show that our approach outperforms the state-of-the-art PDDL+ planners dReal and UPMurphi.

The paper is structured as follows. We begin with preliminaries on PDDL+ planning and CASP. In Section 3, we present our encoding, followed by a discussion of the results of our experiments. Finally, in Section 6, we draw conclusions and discuss future directions of work.

2 Background

In this section, we provide background on the main topics covered by the paper. We first introduce PDDL+ planning, and then ASP and CASP.

Hybrid systems can be described as hybrid automata (Henzinger 1996), that are finite state automata extended with continuous variables that evolve over time. More formally, we have the following:

Definition 1 (Hybrid Automaton) *A* hybrid automaton *is a tuple* $\mathscr{H} = (Loc, Var, Init, Flow, Trans, I)$, *where*

- *Loc is a finite set of locations,* $Var = \{x_1, \ldots, x_n\}$ *is a set of real-valued variables,* $Init(\ell) \subseteq \mathbb{R}^n$ *is the set of initial values for* $x_1, \ldots, x_n$ *for all locations* ℓ.
- *For each location* ℓ, $Flow(\ell)$ *is a relation over the variables in Var and their derivatives of the form*

$$\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathscr{U},$$

  where $x(t) \in \mathbb{R}^n$, *A is a real-valued nxn matrix and* $\mathscr{U} \subseteq \mathbb{R}^n$ *is a closed and bounded convex set.*
- *Trans is a set of discrete transitions. A discrete transition* $t \in Trans$ *is defined as a tuple* (ℓ, g, ξ, ℓ') *where* ℓ *and* ℓ' *are the source and the target locations, respectively, g is the guard of t (given as a linear constraint), and* ξ *is the update of t (given by an affine mapping).*
- $I(\ell) \subseteq \mathbb{R}^n$ *is an invariant for all locations* ℓ.

An illustrative example is given by the hybrid automaton for a thermostat depicted in Figure 1. Here, the temperature is represented by the continuous variable x. In the discrete location corresponding to the heater being off, the temperature falls according to the flow condition $\dot{x} = -0.1x$, while when the heater is on, the temperature increases according to the flow condition $\dot{x} = 5 - 0.1x$. The discrete transitions state that the heater *may* be switched on when the temperature falls below 19 degrees, and switched off when the temperature is greater than 21 degrees. Finally, the invariants state that the heater can be on (off) *only* if the temperature is not greater than 22 degrees (not less than 18 degrees).

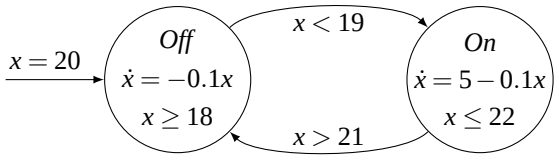

Figure 1: Thermostat hybrid automaton

Planning is an AI technology that seeks to select and organise activities in order to achieve specific goals (Nau, Ghallab, and Traverso 2004). A planner uses a domain model, describing the actions through their pre- and post-conditions, and an initial state together with a goal condition. It then searches for a trajectory through the induced state space, starting at the initial state and ending in a state satisfying the goal condition. In richer models, such as hybrid systems, the induced state space can be given a formal semantics as a timed hybrid automaton, which means that a plan can synchronise activities between controlled devices and external events.

2.1 PDDL+ Planning

Definition 2 (Planning Instance) *A planning instance is a pair* $I = (Dom, Prob)$, *where* $Dom = (Fs, Rs, As, Es, Ps, arity)$ *is a tuple consisting of a finite set of* function symbols *Fs, a finite set of* relation symbols *Rs, a finite set of (durative)* actions *As, a finite set of* events *Es, a finite set of* processes *Ps, and a function arity mapping all symbols in* $Fs \cup Rs$ *to their respective arities.*

The triple $Prob = (Os, Init, G)$ *consists of a finite set of* domain objects *Os, the* initial *state Init, and the* goal *specification G.*

Following (Bogomolov et al. 2014), for a given planning instance I, a *state* of I consists of a discrete component, described as a set of propositions P called *Boolean fluents*, and a numerical component, described as a set of real variables $\mathbf{v}$ called *numerical fluents*. Instantaneous actions are described through preconditions (which are conjunctions of propositions in P and/or numerical constraints over $\mathbf{v}$, and define when an action can be applied) and effects (which define how the action modifies the current state). *Instantaneous* actions and events are restricted to the expression of discrete change. Events have preconditions as for actions, but they are used to model exogenous change in the world, therefore they are triggered as soon as the preconditions are true. A process is responsible for the continuous change of variables, and is active as long as its preconditions are true. *Durative* actions have three sets of preconditions, representing the conditions that must hold when it starts, the invariant that must hold throughout its execution and the conditions that must hold at the end of the action. Similarly, a durative action has three sets of effects: effects that are applied when the action starts, effects that are applied when the action ends and a set of continuous numeric effects which are applied continuously while the action is executing.

Definition 3 (Plan) *A plan for a planning instance* $I = ((Fs, Rs, As, Es, Ps, arity), (Os, Init, G))$ *is a finite set of triples* $(t, a, d) \in \mathbb{R}^* \times As \times \mathbb{R}^*$, *where t is a timepoint, a is an action and d is the action duration.*

Note that processes and events do not appear in a plan, as they are not under the direct control of the planner.

2.2 Answer Set Programming

Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as in first-order logic. A literal is an atom a or its classical negation $\neg a$. A *rule* is a statement of the form:

$$h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \qquad (1)$$

where h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is

that a reasoner who believes $\{l_1,\ldots,l_m\}$ and has no reason to believe $\{l_{m+1},\ldots,l_n\}$, has to believe h. The formal semantics, defined in terms of models of a set of rules, is given later. We call h the *head* of the rule, and $\{l_1,\ldots,l_m,\text{not } l_{m+1},\ldots,\text{not } l_n\}$ the *body* of the rule. Given a rule r, we denote its head and body by $head(r)$ and $body(r)$, respectively. A rule with an empty body is called a *fact*, and indicates that the head is always true. In that case, the connective $\leftarrow$ is often dropped.

A *program* is a pair $\langle\Sigma,\Pi\rangle$, where Σ is a signature and Π is a set of rules over Σ. Often we denote programs by just the second element of the pair, and let the signature be defined implicitly.

A set A of literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A. A literal l is *satisfied* by a consistent set of literals A (denoted by $A \models l$) if $l \in A$. If l is not satisfied by A, we write $A \not\models l$. A set $\{l_1,\ldots,l_k\}$ of literals is satisfied by a set of literals A ($A \models \{l_1,\ldots,l_k\}$) if each l_i is satisfied by A.

Programs not containing default negation are called *definite*. A consistent set of literals A is *closed* under a definite program Π if, for every rule of the form (1) such that the body of the rule is satisfied by A, the head belongs to A. This allows us to state the semantics of definite programs.

Definition 4 *A consistent set of literals A is an* answer set *of definite program Π if A is closed under all the rules of Π and A is set-theoretically minimal among the sets closed under all the rules of Π.*

To define answer sets of arbitrary programs, we introduce the *reduct* of a program Π with respect to a set of literals A, denoted by Π^A. The reduct is obtained from Π by: **(1)** deleting every rule r such that $l \in A$ for some expression of the form *not* l from the body of r, and **(2)** removing all expressions of the form *not* l from the bodies of the remaining rules. The semantics of arbitrary ASP programs can thus be defined as follows.

Definition 5 *A consistent set of literals A is an* answer set *of program Π if it is an answer set of Π^A.*

To simplify the programming task, variables (identifiers with an uppercase initial) are allowed in ASP programs. A rule containing variables (a *non-ground* rule) is viewed as a shorthand for the set of its *ground instances*, obtained by replacing the variables by all possible ground terms. Similarly, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules.

There are also shorthands, which we introduce informally to save space. A rule whose head is empty is called *denial*, and states that its body must not be satisfied. A *choice rule* has a head of the form

$$\lambda\{m(\vec{X}) : \Gamma(\vec{X})\}\mu$$

where $\vec{X}$ is a list of variables, λ, μ are non-negative integers, and $\Gamma(\vec{X})$ is a set of literals that may include variables from $\vec{X}$. A choice rule intuitively states that, in every answer set, the number of literals of the form $m(\vec{X})$ such that $\Gamma(\vec{X})$ is satisfied must be between λ and μ. If not specified, λ, μ default, respectively, to 0, ∞. For example, given a relation q defined by $\{q(a),q(b)\}$, the rule:

$$1\{p(X) : q(X)\}2.$$

intuitively identifies three possible sets of conclusions: $\{p(a)\}$, $\{p(b)\}$, and $\{p(a),p(b)\}$.

2.3 Constraint ASP

CASP integrates ASP and Constraint Programming (CP) in order to deal with continuous dynamics. In this section we provide an overview of CP and of its integration in CASP.

The central concept of CP is the *Constraint Satisfaction Problem (CSP)* (Rossi, van Beek, and Walsh 2006), which is formally defined as a triple $\langle X,D,C\rangle$, where $X=\{x_1,\ldots,x_n\}$ is a set of variables, $D=\{D_1,\ldots,D_n\}$ is a set of domains, such that D_i is the domain of variable x_i, and C is a set of constraints. A *solution* to a CSP $\langle X,D,C\rangle$ is a complete assignment (i.e. where a value from the respective domain is assigned to each variable) satisfying every constraint from C.

There is currently no widely accepted, standardized definition of CASP. Multiple definitions have been given in the literature (Ostrowski and Schaub 2012a; Mellarkod, Gelfond, and Zhang 2008a; Baselice, Bonatti, and Gelfond 2005; Balduccini 2009). Although largely overlapping, these definitions are all somewhat distinct from each other.

To ensure generality of our results, we introduce a simplified definition of CASP, defined next, which captures the common traits of the above approaches. The main results of this paper will be given using our simplified definition of CASP. Later, in Section 4, we introduce a specific CASP language to discuss the use case and the experimental results.

Syntax. In order to accommodate CP constructs, the language of CASP extends ASP by allowing *numerical constraints* of the form $x \bowtie y$, where $\bowtie \in \{<,\leq,=,\neq,\geq,>\}$, and x and y are *numerical variables*[1] or standard arithmetic terms possibly containing numerical variables, numerical constants, and ASP variables. Numerical constraints are only allowed in the head of rules.

Semantics. Given a numerical constraint c, let $\tau(c)$ be a function that maps c to a syntactically legal ASP atom and τ^{-1} be its inverse. We say that an ASP atom a *denotes* a constraint c if $a=\tau(c)$. Function τ is extended in a natural way to CASP rules and programs. Note that, for every CASP program Π, $\tau(\Pi)$ is an ASP program.

[1] Numerical variables are distinct from ASP variables.

Finally, given a set A of ASP literals, let $\gamma(A)$ be the set of ASP atoms from A that denote numerical constraints. The semantics of a CASP program can thus be given by defining the notion of CASP solution, as follows.

Definition 6 *A pair $\langle A, \alpha \rangle$ is a* CASP solution *of a CASP program Π if-and-only-if A is an answer set of $\tau(\Pi)$ and α is a solution to $\tau^{-1}(\gamma(A))$.*

3 Encoding PDDL+ Models into CASP Problems

In this section we describe our encoding of PDDL+ problems in CASP. Our approach is based on research on reasoning about actions and change, and action languages (Gelfond and Lifschitz 1993; Reiter 2001; Chintabathina, Gelfond, and Watson 2005). It builds upon the existing SAT-based (Kautz and Selman 1992) and ASP-based planning approaches (Lifschitz 1999), and extends them to hybrid domains.

In reasoning about actions and change, the evolution of a domain over time is often represented by a *transition diagram* (or *transition system*) that represents states and transitions between states through actions. Traditionally, in transition diagrams, actions are instantaneous, and states have no duration and are described by sets of Boolean fluents. Sequences of states characterizing the evolutions of the domain are represented as a sequence of *discrete time steps*, identified by integer numbers, so that step 0 corresponds to the initial state in the sequence. We extend this view to hybrid domains according to the following principles:

- Similarly to PDDL+, a state is characterized by Boolean fluents and numerical fluents.
- The flow of actual time is captured by the notion of *global time* (Chintabathina, Gelfond, and Watson 2005). States have a duration, given by the global time at which a state begins and ends. Intuitively, this conveys the intuition that time flows "within" the state.
- The truth value of Boolean fluents only changes upon state transitions. That is, it is unaffected by the flow of time "within" a state. On the other hand, the value of a numerical fluent may change within a state.
- The global time at which an action occurs is identified with the end time of the state in which the action occurs.
- Invariants are checked at the beginning and at the end of every state in which durative actions and processes are in execution. Thus, in order to guarantee soundness we exploit a discretize and validate approach.

Next, we describe the CASP formalization of PDDL+ models. We begin by discussing the correspondence between global time and states, and the representation of the values of fluents and of occurrences of actions.

The global time at which the state at step i begins is represented by numerical variable $start(i)$. Similarly, the end time is represented by $end(i)$. The truth value of Boolean fluent f at discrete time step i is represented by literal $holds(f,i)$ if f is true and by $\neg holds(f,i)$ otherwise. For every *numerical fluent* n, we introduce two numerical variables, representing its value at the beginning and at the end of time step i. The variables are $v\_initial(n,i)$ and $v\_final(n,i)$, respectively. The occurrence of an action a at time step i is represented by an atom $occurs(a,i)$.

Additive fluents, whose value is affected by *increase* and *decrease* statements of PDDL+, are represented by introducing numerical variables of the form $v(contrib(n,s),i)$, where n is a numerical fluent, s is a constant denoting a source (e.g., the action that causes the increase or decrease), and i is a time step. The expression denotes the amount of the contribution to fluent n from source s at step i. Intuitively, the value of n at the end of step i (encoded by numerical variable $v\_final(n,i)$) is calculated from the values of the individual contributions. Next, we discuss the encoding of the domain portion of a PDDL+ problem.

3.1 Domain Encoding

In the following discussion, ASP variables I, $I1$, $I2$ denotes time steps.

Actions. The encoding of the preconditions of actions varies depending on their type. Preconditions on Boolean fluents are encoded by means of denials. For example, a denial:

$$\leftarrow holds(unavail(tk1),I), occurs(refuel\_with(tk1),I).$$

states that refuel tank $tk1$ must be available for the corresponding refuel action to occur. Preconditions on numerical fluents are encoded by means of numerical constraints on the corresponding numerical variables. For example, a rule

$$v\_final(height(ball),I) > 0 \leftarrow occurs(drop(ball),I).$$

states that, if $drop(ball)$ is selected to occur, then the height of the ball is required to be greater than 0 in the preceding state.

The effects of instantaneous actions on Boolean fluents are captured by rules of the form:

$$holds(f,I+1) \leftarrow occurs(a,I).$$

where f is a fluent and a is an action. The rule states that f is true at the next time step $I+1$ if the action occurs at (the end of) step I. The effects on numerical fluents are represented similarly, but the head of the rule is replaced by a numerical constraint. For example, the rule:

$$v\_initial(height(ball),I+1) = 10 \leftarrow occurs(lift(ball),I).$$

states the action of lifting the ball causes its height to be

10 at the beginning of the state following the occurrence of the action. If the action increases or decreases the value of a numerical fluent, rather than setting it, then a corresponding variable of the form $v(contrib(n,s),i)$ is used in the numerical constraint. The link between contributions and numerical fluent values is established by axioms described later in this section.

Durative actions. A durative action d is encoded as two instantaneous actions, $start(d)$ and $end(d)$. The start (end) preconditions of d are mapped to preconditions of $start(d)$ ($end(d)$). The overall conditions are encoded with denials and constraints, as described above in the context of preconditions. Start (end) effects are mapped to effects of $start(d)$ and $end(d)$ actions. Additionally, $start(d)$ makes fluent $inprogr(d)$ true. The continuous effects of d are made to hold in any state in which $inprogr(d)$ holds. For example, if a $refuel$ action causes the level of fuel in a tank to increase linearly with the flow of time, its effect may be encoded by:

$$v(contrib(flevel,refuel),I) = end(I) - start(I) \leftarrow holds(inprogr(d),I).$$

The above rule intuitively states that, at the end of any state in which d is in progress, the fuel level increases proportionally to the duration of the state. The value of the fluent is updated from its set of contributions S by the general constraint, shown next, which applies to every fluent F:

$$v\_final(F,I) = v\_initial(F,I) + \sum_{s \in S} v(contrib(F,s),I).$$

The fact that the value of numerical fluents stays the same by default throughout the time interval associated with a state is modeled by a rule:

$$v\_final(F,I) = v\_initial(F,I) \leftarrow \text{not } ab(F,I).$$

which applies to every numerical fluent F. Intuitively, this rule must not be applicable when the value of F is being changed by an action, process, or event. This is enforced by adding a rule that makes $ab(F,I)$ true. For example, for a durative action d that affects a numerical fluent f, the encoding includes a rule:

$$ab(f,I) \leftarrow holds(inprogr(d),I).$$

In a similar way, the contribution to a numerical fluent by every source is assumed to be 0 by default. This is guaranteed by the rule:

$$v(contrib(F,S),I) = 0) \leftarrow \text{not } ab(F,I).$$

To keep track of the duration of a durative action when the action spans multiple time steps, a rule records the global time at which d begun:

$$stime(d) = end(I) \leftarrow occurs(start(d),I).$$

Action $end(d)$ is modeled so that it is automatically triggered after $start(d)$. Finding the time at which the end action occurs, both in terms of time step and global time, is part of the constraint problem to be solved. The following rule:

$$1\{occurs(end(d),I2) : I2 > I1\}1 \leftarrow occurs(start(d),I1).$$

ensures that $end(d)$ will be triggered at some timepoint following $start(d)$. Finally, requirements on the duration of durative actions are encoded using numerical constraints: if the PDDL+ problem specifies that the duration of d is δ, the requirement is encoded by a rule:

$$end(I) - stime(d) = \delta \leftarrow occurs(end(d),I).$$

Intuitively, any CASP solution of the corresponding program will include a specification of when $end(d)$ must occur, both in terms of time step and global time.

Processes and Events. The encoding of processes and events follows the approach outlined earlier, respectively, for durative and instantaneous actions. However, their triggering is defined by PDDL+'s *must* semantics, which prescribes that they are triggered as soon as their preconditions are true. In CASP, this is captured by a choice rule combined with numerical constraints. Intuitively, when the Boolean conditions of the process are satisfied, the choice rule states the process will start unless it is inhibited by unsatisfied numerical conditions. Constraints enforced on the numerical conditions capture the latter case. Consider a process corresponding to a falling object, with preconditions $\neg held$ and $height > 0$. The choice rule:

$$1\{occurs(start(falling),I), is\_false(height > 0,I)\}1 \leftarrow holds(\neg held,I).$$

entails two possible, equally likely, outcomes: the object will either start falling, or be prevented from doing so by the fact that condition $height > 0$ is false. The second outcome is possible only if the height is indeed not greater than 0, which is enforced by the constraint:

$$v\_final(height,I) \leq 0 \leftarrow is\_false(height > 0,I).$$

Given an arbitrary process, the corresponding choice rule lists an atom $is\_false(\cdot,I)$ for every numerical condition, and the encoding includes a constraint on the value of $v\_final(n,I)$ corresponding to the complement of that condition. The treatment of events is similar.
The encoding is completed by the following statements:

$$start(I+1) = end(I).$$

$$v\_initial(F,I+1) = v\_final(F,I).$$

$$holds(F,I+1) \leftarrow holds(F,I), \text{not } holds(\neg F,I+1).$$
$$holds(\neg F,I+1) \leftarrow holds(\neg F,I), \text{not } holds(F,I+1).$$

The first rule ensures that there are no gaps between the time intervals associated with consecutive states. The others handle fluent propagation from a state to the next.

3.2 Problem Encoding

The problem portion of the PDDL+ problem is encoded as follows.

Initial state. The encoding of the initial state consists of a set of rules specifying the values of fluents in $P \cup v$ at step 0.

Goals. The encoding of a goal consists of a set of denials on Boolean fluents and of constraints on numerical fluents, obtained similarly to the encoding of preconditions of actions, discussed earlier.

Given a PDDL+ planning instance I, by $\Pi(I)$ we denote the CASP encoding of I. Next, we turn our attention to the planning task.

3.3 Planning Task

Our approach to planning leverages techniques from ASP-based planning (Lifschitz 2002; Balduccini, Gelfond, and Nogueira 2006). The planning task is specified by the planning module, M, which consists of the single rule:

$$\{occurs(A,I), occurs(start(D),I)\}.$$

where A,D are variables ranging over instantaneous actions and durative actions, respectively. The rule intuitively states that any action may occur (or start) at any time step.

It can be shown that the plans for a given maximum time step for a PDDL+ planning instance I are in one-to-one correspondence with the CASP solutions of $\Pi(I) \cup M$. The plan encoded by a CASP solution A can be easily obtained from the atoms of the form $occurs(a,i)$ and from the value assignments to numerical variables $start(i)$ and $end(i)$.

It is also worth noting the level of modularity of our approach. In particular, it is straightforward to perform other reasoning tasks besides planning (e.g., a hybrid of planning and diagnostics is often useful for applications) by replacing the planning module by a different one, as demonstrated for example in (Balduccini and Gelfond 2003b).

4 Case Study

For our case study, we have focused on a specific instance of CASP, called EZCSP (Balduccini 2009; Balduccini and Lierler 2013). In EZCSP, numerical constraints are encoded as arguments of the special relation *required*, e.g. $required(start(I+1) = end(I))$. Encodings of the *generator* (Bogomolov et al. 2014) and *car* domains (Bryce et al. 2015) were created as described above, and the architecture of the EZCSP solver was expanded to ensure soundness of the algorithm (see below). The complete encodings are omitted due to space considerations. Rather, to illustrate our approach, we present a fragment of the encoding of process *generate* from the *generator* domain, whose PDDL+ representation is shown in Figure 2. The fragment captures the invariants and the change of fuel level. The process has two continuous effects: it decreases the fuel level (the expression `(* #t 1)` states that the change is continuous and linear with respect to time) and increases the value of variable `generator_time`, which keeps track of how long the generator ran. The choice of *generate* was motivated by the fact that the representation of processes is arguably one of the most challenging aspects of encoding PDDL+ in CASP. The invariant on the maximum fuel level is encoded by two EZCSP rules (atom $tankcap(\cdot)$ determines the capacity of the tank):

$$required(v\_initial(fuel\_level, I) \leq TC) \leftarrow tankcap(TC).$$

$$required(v\_final(fuel\_level, I) \leq TC) \leftarrow tankcap(TC).$$

The (negative) contribution to the generator's fuel level is modeled by:

$$required(v(contrib(fuel\_level, generate), I) = -1 * (end(I) - start(I))) \leftarrow holds(inprogr(generate), I).$$

From an algorithmic perspective, the EZCSP solver

```
(:process generate
 :parameters (?g - generator)
 :condition
  (and
   (over all
    (>= (fuelLevel ?g) 0)
   )
   (over all
    (<= (fuelLevel ?g) (capacity ?g))
   )
  )
 :effect
  (and
   (decrease (fuelLevel ?g) (* #t 1))
   (increase (generator_time ?g)
             (* #t 1))
   )
 )
```

Figure 2: PDDL+ process from the Generator domain

computes CASP solutions of a program Π by iteratively (1) using an ASP solver to find an answer set A of Π, and (2) using a constraint solver to find the solutions of the CSP encoded by A. To account for the discretize and validate approach mentioned earlier, we have extended the EZCSP solver with a validation step. In the extended architecture, shown in Figure 3, if step (2) is successful, the tool VAL is called to validate the plan before returning it. If VAL finds the plan not to be valid, it returns which invariant was violated and at which timepoint. If that happens, the *expansion* process occurs, where the encoding is expanded with (1) new numerical variables that represent the value of the involved numerical flu-

ents at that timepoint, and (2) numerical constraints enforcing the invariant on them. The CASP solutions for the new encoding are computed again[2], and the process is iterated until no invariants are violated.

To illustrate the expansion process, let us consider a durative action d causing fluent f to increase by $\iota(\Delta)$, where Δ is elapsed time. Suppose invariant $f < c$ is violated at a timepoint t that falls within the time interval associated with time step i. The encoding is then expanded by:

$$required(v'(F,i) = \\ v\_initial(F,i) + v'(contrib(F,s),i)).$$

$$required(\\ v'(contrib(F,s),i)) = \iota(t - start(i)) \\ ) \leftarrow holds(inprog(d),i).$$

$$required(v'(F,i) < c).$$

5 Experimental Results

We performed an empirical evaluation of the performance achieved with our approach. The comparison was with the state-of-the-art PDDL+ planners dReal (Bryce et al. 2015) and UPMurphi. Although SpaceEx (Bogomolov et al. 2014) is indeed a related approach, it was not included in the preliminary comparison because it is focused on proving only plan non-existence. The experimental setup used a virtual machine running in VMWare Workstation 12 on a computer with an i7-4790K CPU at 4.00GHz. The virtual machine was assigned a single core and 4GB RAM. The operating system was Fedora 22 64 bit. The version of EZCSP used was 1.7.4[3], with gringo 3.0.5[4] and clasp 3.1.3[5] as grounding tool and ASP solver, and B-Prolog 7.5[6] and GAMS 24.5.7[7] as constraint solvers. The former was used for all linear problems and the latter for the non-linear ones. The other systems used were dReal 2.15.11[8], configured as suggested by its authors, and UPMurphi 3.0.2[9].

The experiments were conducted on the linear and non-linear versions of the *generator* and *car* domains.

The comparison with dReal was based on finding a single plan with a given maximum time step, as discussed in (Bryce et al. 2015). The results are summarized in Table 1. The comparison with UPMurphi was based on the cumulative times for finding a single plan by progressively increasing the maximum time step. The results are reported in Table 2. In the tables, entries marked "-" indicate a timeout (threshold 600 sec). Entries marked "*" indicate missing entries due to licensing limitations (see below). It should be noted that none of the instances triggered the expansion process described in the previous section, given that all plans were found to be valid by VAL. Next, we discuss the experimental results obtained for each domain.

Generator. Our encoding uses Torricelli's law ($v = \sqrt{2gh}$) to model the transfer of liquid. This is a more complex model than the one used in the dReal encoding, but is more physically accurate. The instances were generated by increasing the number of refuel tanks from 1 to 8. The CASP encoding was as discussed above, and included a single, encoding-level heuristic stating that action $start(generate)$ must occur during the first state transition and at timepoint 0. (dReal includes multiple heuristics that are hard-coded in the solver.)

The execution times for EZCSP for a fixed maximum time step (Table 1) ranged between 0.28 sec and 261.89 sec for the linear variant, and between 0.72 sec and 256.59 sec for the non-linear one. The non-linear variant was only tested up to instance 7 because of limitations of the free version of GAMS. In both the linear and non-linear case, the EZCSP encoding was substantially faster than dReal. Especially remarkable is the fact that, in both cases, dReal timed out on all instances except for the first one.

The cumulative times for EZCSP (Table 2) ranged between 0.89 sec and 292.22 sec for the linear case, with no timeouts. In the non-linear case, the times were between 1.44 sec and 267.11 sec, with a timeout in instance 8. UPMurphi did not scale as well. In the linear case, only instances 1-3 were solved, and resulted in times ranging between 2.02 sec and 91.80 sec. The speedup yielded by EZCSP reached about one order of magnitude before UPMurphi began to time out. In the non-linear case, UPMurphi timed out in all instances.

Car. The version of the car domain we used is the same that was adopted in (Bryce et al. 2015). In this domain, a vehicle needs to travel a certain goal distance from its start position. The vehicle is initially at rest. Two actions allow the vehicle to accelerate and to decelerate. The goal is achieved when the vehicle reaches the desired distance and its speed is 0. In the linear variant, accelerating increases the velocity by 1 and decelerating decreases it by 1. In the non-linear variant, accelerating increases the acceleration by 1, and similarly for decelerating. The velocity is influenced by the acceleration according to the usual laws of physics. The calculation also takes into account a drag factor equal to $0.1 \cdot v^2$. The instances were obtained by progressively increasing the range of allowed accelerations (velocities in the linear version) from $[-1,1]$ to $[-8,8]$. The CASP encoding leveraged no heuristics and, as discussed earlier, the underlying solvers are completely general-purpose.

[2] Only the solutions of the CSP need to be recomputed.

[3] http://mbal.tk/ezcsp/

[4] http://sourceforge.net/projects/potassco/files/gringo/

[5] https://sourceforge.net/projects/potassco/files/clasp/

[6] http://www.picat-lang.org/bprolog/

[7] http://www.gams.com/

[8] http://dreal.github.io/

[9] https://github.com/gdellapenna/UPMurphi/

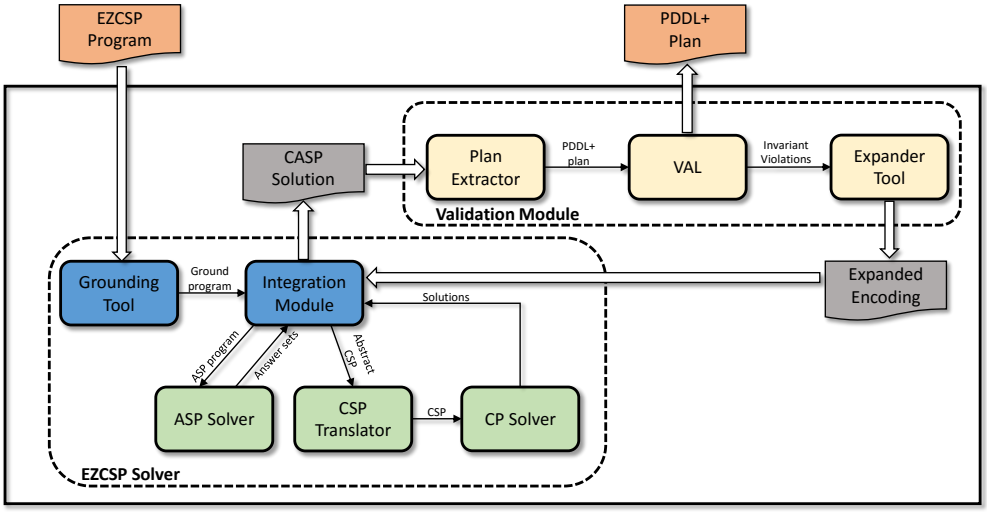

Figure 3: Extended Solver Architecture

Domain	Solver	1	2	3	4	5	6	7	8
Gen linear	EZCSP	0.28	1.03	4.21	7.25	27.08	43.42	54.83	261.89
	dReal	3.73	-	-	-	-	-	-	-
Gen non-linear	EZCSP	0.72	1.62	0.68	1.05	87.95	256.59	238.93	*
	dReal	8.18	-	-	-	-	-	-	-
Car linear	EZCSP	0.32	0.31	0.32	0.32	0.32	0.30	0.31	0.31
	dReal	1.11	1.11	1.15	1.14	1.19	1.13	1.14	1.19
Car non-linear	EZCSP	0.71	0.68	0.29	0.39	0.25	0.25	0.26	0.84
	dReal	58.21	162.60	-	-	-	-	-	-

Table 1: Fixed time step. Results in seconds. Problem instances refer to number of tanks (*generator*) and max acceleration (*car*).

Domain	Solver	1	2	3	4	5	6	7	8
Gen linear	EZCSP	0.89	1.92	5.46	9.93	30.79	50.25	67.97	292.22
	UPMurphi	2.02	12.75	91.80	-	-	-	-	-
Gen non-linear	EZCSP	1.44	2.44	13.10	53.70	88.58	267.11	250.03	-
	UPMurphi	-	-	-	-	-	-	-	-
Car linear	EZCSP	1.01	0.98	1.04	0.99	0.91	0.85	0.88	0.83
	UPMurphi	0.40	0.38	0.38	0.38	0.41	0.39	0.40	0.41
Car non-linear	EZCSP	2.32	1.49	1.14	1.85	1.14	1.18	1.06	2.13
	UPMurphi	184.88	-	-	-	-	-	-	-

Table 2: Cumulative times. Results in seconds. Problem instances refer to number of tanks (*generator*) and max acceleration (*car*).

As shown in Table 1, the execution times for EZCSP were around 0.30 sec for the linear case, and between 0.25 sec and 0.84 sec for the non-linear one. These times are about 3 times faster than dReal in the linear case and orders of magnitude better in the non-linear case, where dReal times out in instances 3-8. The scalability of EZCSP appears to be excellent, with no significant growth.

The comparison with UPMurphi on cumulative times shows some interesting behavior. In the linear case, EZCSP is, in fact, about 2.5 times slower than UPMurphi. The former has times ranging between 0.83 sec

and 1.04 sec, while UPMurphi's times are between 0.38 sec and 0.41 sec. On the other hand, EZCSP outperforms UPMurphi in the non-linear case, with all instances solved in times between 1.06 sec and 2.32 sec, while UPMurphi only solves the first instance with a time of 184.88 sec, i.e., nearly 2 orders of magnitude slower than EZCSP.

We believe the empirical results demonstrate the promise of our approach. From the perspective of the underlying solving algorithms, it is worth stressing that the better results of EZCSP over dReal are especially remarkable given that the latter employs planning-specific heuristics, while the EZCSP solver and its components are not specialized for a given reasoning task.

6 Conclusions

In this paper we have presented a new approach to PDDL+ planning based on CASP languages that provides a solid basis for applying logic programming to PDDL+ planning. Experiments on well-known domains, some involving non-linear continuous change, have shown that our approach outperforms comparable state-of-the-art PDDL+ planners.

Although other CASP solvers exist, EZCSP is, to the best of our knowledge, the only one supporting both non-linear constraints, required for modeling non-linear continuous change, and real numbers.

ACSOLVER (Mellarkod, Gelfond, and Zhang 2008b) implements an eager approach to CASP solving, where (in contrast to the lazy approach of EZCSP) ASP and CSP solving are tightly coupled and interleaved. It does not support non-linear or global constraints, but allows for real numbers.

CLINGON (Ostrowski and Schaub 2012b) is another tightly coupled CASP solver. The available implementation, however, is not broadly applicable to the kinds of problems considered in this paper. In fact, CLINGON does not support non-linear constraints and real numbers. On the other hand, differently from EZCSP, it allows for numerical constraints both in the head of rules and in their bodies.

A high level view of the languages and solving techniques employed by these solvers can be found in (Lierler 2014).

Finally, it is also worth noting that basing our approach on CASP makes it amenable to be expanded to handle uncertainty about the initial situation or the effects of actions (e.g., (Morales, Tu, and Son 2007)). Another interesting possibility is the use of PDDL+ domain descriptions, translated to CASP, for both planning and diagnosis, along the lines of the approach applied in (Balduccini and Gelfond 2003a) to ASP domain descriptions.

References

[Balduccini and Gelfond 2003a] Balduccini, M., and Gelfond, M. 2003a. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)* 3(4–5):425–461.

[Balduccini and Gelfond 2003b] Balduccini, M., and Gelfond, M. 2003b. Logic Programs with Consistency-Restoring Rules. In Doherty, P.; McCarthy, J.; and Williams, M.-A., eds., *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, 9–18.

[Balduccini and Lierler 2013] Balduccini, M., and Lierler, Y. 2013. Integration Schemas for Constraint Answer Set Programming: a Case Study. *Theory and Practice of Logic Programming (TPLP), On-line Supplement*.

[Balduccini, Gelfond, and Nogueira 2006] Balduccini, M.; Gelfond, M.; and Nogueira, M. 2006. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence* 47(1–2):183–219.

[Balduccini 2009] Balduccini, M. 2009. Representing Constraint Satisfaction Problems in Answer Set Programming. In *ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*.

[Baselice, Bonatti, and Gelfond 2005] Baselice, S.; Bonatti, P. A.; and Gelfond, M. 2005. Towards an Integration of Answer Set and Constraint Solving. In *Proceedings of ICLP 2005*.

[Bogomolov et al. 2014] Bogomolov, S.; Magazzeni, D.; Podelski, A.; and Wehrle, M. 2014. Planning as model checking in hybrid domains. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2228–2234.

[Bogomolov et al. 2015] Bogomolov, S.; Magazzeni, D.; Minopoli, S.; and Wehrle, M. 2015. PDDL+ planning with hybrid automata: Foundations of translating must behavior. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015), Jerusalem, Israel, June 7-11, 2015.*, 42–46.

[Bryce et al. 2015] Bryce, D.; Gao, S.; Musliner, D. J.; and Goldman, R. P. 2015. Smt-based nonlinear PDDL+ planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 3247–3253.

[Chintabathina, Gelfond, and Watson 2005] Chintabathina, S.; Gelfond, M.; and Watson, R. 2005. Modeling Hybrid Domains Using Process Description Language. In *Proceedings of ASP '05 – Answer Set Programming: Advances in Theory and Implementation*, 303–317.

[Coles et al. 2012] Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2012. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research* 44:1–96.

[Della Penna et al. 2009] Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. UPMurphi: A tool for universal planning on PDDL+ problems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI.

[Fox and Long 2006] Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27:235–297.

[Fox, Howey, and Long 2004] Fox, M.; Howey, R.; and Long, D. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, 294–301.

[Gelfond and Lifschitz 1991] Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

[Gelfond and Lifschitz 1993] Gelfond, M., and Lifschitz, V. 1993. Representing Action and Change by Logic Programs. *Journal of Logic Programming* 17(2–4):301–321.

[Henzinger 1996] Henzinger, T. A. 1996. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, 278–292.

[Kautz and Selman 1992] Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *ECAI*, 359–363.

[Li and Williams 2008] Li, H. X., and Williams, B. C. 2008. Generative planning for hybrid systems based on flow tubes. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E. A., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 206–213. AAAI.

[Lierler 2014] Lierler, Y. 2014. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence* 207:1–22.

[Lifschitz 1999] Lifschitz, V. 1999. *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, Berlin. chapter Action Languages, Answer Sets, and Planning, 357–373.

[Lifschitz 2002] Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138:39–54.

[McDermott 2003] McDermott, D. V. 2003. Reasoning about autonomous processes in an estimated-regression planner. In Giunchiglia, E.; Muscettola, N.; and Nau, D. S., eds., *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003)*, 143–152. AAAI.

[Mellarkod, Gelfond, and Zhang 2008a] Mellarkod, V. S.; Gelfond, M.; and Zhang, Y. 2008a. Integrating Answer Set Programming and Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*.

[Mellarkod, Gelfond, and Zhang 2008b] Mellarkod, V. S.; Gelfond, M.; and Zhang, Y. 2008b. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence* 53(1-4):251–287.

[Morales, Tu, and Son 2007] Morales, R.; Tu, P. H.; and Son, T. C. 2007. An Extension to Conformant Planning Using Logic Programming. In Veloso, M. M., ed., *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, 1991–1996.

[Nau, Ghallab, and Traverso 2004] Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Ostrowski and Schaub 2012a] Ostrowski, M., and Schaub, T. 2012a. ASP Modulo CSP: The Clingcon System. *Journal of Theory and Practice of Logic Programming (TPLP)* 12(4–5):485–503.

[Ostrowski and Schaub 2012b] Ostrowski, M., and Schaub, T. 2012b. ASP modulo CSP: the clingcon system. *TPLP* 12(4-5):485–503.

[Penberthy and Weld 1994] Penberthy, J. S., and Weld, D. S. 1994. Temporal planning with continuous change. In Hayes-Roth, B., and Korf, R. E., eds., *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, 1010–1015. AAAI Press / The MIT Press.

[Reiter 2001] Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dy- namical Systems*. MIT Press.

[Rossi, van Beek, and Walsh 2006] Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

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