TLP **17** (4): 591–633, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000187 First published online 4 July 2017

CASP solutions for planning in hybrid domains

MARCELLO BALDUCCINI
Saint Joseph's University, Department of Decision and Support Services, 5600 City Avenue, Philadelphia, PA 19131, USA
(*e-mail:* marcello.balduccini@gmail.com)

DANIELE MAGAZZENI
King's College London, Department of Informatics, The Strand, London, WC2R 2LS, England
(*e-mail:* daniele.magazzeni@kcl.ac.uk)

MARCO MARATEA
University of Genoa, Italy
(*e-mail:* marco@dibris.unige.it)

EMILY C. LEBLANC
Drexel University, Computer Science Department, 3141 Chestnut Avenue, PA 19104, USA
(*e-mail:* ecl38@drexel.edu)

submitted 03 May 2017; revised 15 May 2017; accepted 29 May 2017

Abstract

Constraint answer set programming (CASP) is an extension of answer set programming that allows for numerical constraints to be added in the rules. PDDL+ is an extension of the PDDL standard language of automated planning for modeling mixed discrete-continuous dynamics. In this paper, we present CASP solutions for dealing with PDDL+ problems, i.e., encoding from PDDL+ to CASP, and extensions to the algorithm of the EZCSP CASP solver in order to solve CASP programs arising from PDDL+ domains. An experimental analysis, performed on well-known linear and non-linear variants of PDDL+ domains, involving various configurations of the EZCSP solver, other CASP solvers, and PDDL+ planners, shows the viability of our solution.

KEYWORDS: Answer set programming, planning, hybrid domains

1 Introduction

Constraint answer set programming (CASP) (Baselice *et al.* 2005) is an extension of answer set programming (ASP) (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991; Marek and Truszczynski 1999; Niemelä 1999; Baral 2003) that makes it possible to add numerical constraints to the rules of ASP programs, thus allowing to represent and reason on infinite state systems. In the automated planning community, the PDDL standard language for domain representation has been extended with constructs for modeling mixed discrete-continuous dynamics, thus with a similar objective of CASP development. The improved language of the planning community

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

is called PDDL+ (Fox and Long 2006). Current CASP algorithms and solvers are either based on an eager approach to CASP solving (Mellarkod *et al.* 2008; Ostrowski and Schaub 2012), where the numerical constraints are processed within the ASP search, or are based on a lazy approach (Balduccini 2009), where first an ASP solution is found, and then the numerical constraints involved in the ASP solution are checked for consistency by a constraint satisfaction problem (CSP) solver. Independently from the approach used, in order to be able to solve PDDL+ problems with CASP technology, there is the need to both define a suitable encoding from PDDL+ to CASP, and to extend the architecture of current CASP solvers.

In this paper, we present an approach for using CASP solving techniques to find plans for PDDL+ problems. In particular, first we design an encoding from PDDL+ to CASP, where we show how the various constructs and the overall planning problems can be expressed in CASP by building on work on reasoning about action and change, and previous work on ASP encoding of planning problems. Given that there is still no standard language for CASP, we will mainly rely on the language constructs of the EZCSP solver. This is not a severe limitation: variants of our encoding for other CASP languages, e.g., CLINGON, can be obtained by syntactic modifications. Second, we present an architecture suitable for finding PDDL+ plans for a PDDL+ problem via its translation to CASP and the use of a CASP solver.

The proposed encoding and solving techniques follow a discretize-and-validate approach (Della Penna *et al.* 2009) in which a discretization is applied during the generation of the CASP encoding. In order to select only those CASP solutions that correspond to valid plans, the basic CASP solving algorithm is extended with a further check that uses the plan validator VAL (Howey *et al.* 2004) to verify the validity of the solution found. If the test fails, another CASP solution is found, until a valid plan has been found. Although the architecture is given for the EZCSP solver, it can be extended to other CASP solvers.

The paper contains an extensive experimental analysis on well-known PDDL+ domains. The analysis includes (i) various configurations of the EZCSP solver, (ii) variants of the proposed encoding, exploiting both domain-specific and domain-independent heuristics, (iii) variants of the encoding for use with the CLINGON CASP solver, studying the ramifications of the added language expressivity of the solver, (iv) state-of-the-art PDDL+ planners dReal and UPMURPHI, and (v) two planning tasks, i.e., finding a plan at fixed (optimal) time step, and by progressively increasing the maximum time step. The experimental evaluation provides interesting results regarding the performance of CASP-based solutions versus other PDDL+ planning approaches. It also demonstrates that PDDL+ planning is an excellent source of CASP benchmarks, with results that are sometimes surprising. Overall, the results show that our solution employing EZCSP performs well across all domains and reasoning tasks, at the same time being the only solution able to deal with all cases presented.

The paper is structured as follows. Section 2 introduces needed preliminaries about PDDL+, ASP, and CASP. Section 3 presents the proposed architecture for planning in hybrid domains and discusses our encoding from PDDL+ to CASP, which is exemplified in Section 4 by means of a concrete PDDL+ problem. The experimental

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

analysis is provided in Section 5. Section 6 presents a more sophisticated extension of the solving architecture. The paper ends by discussing related work in Section 7, and by drawing conclusions in Section 8.

2 Preliminaries

In this section, we provide background knowledge on the main topics covered by the paper. We first introduce the domain of hybrid systems, followed by a brief introduction on AI planning and its application to hybrid domains. Finally, we briefly describe ASP, CSP, and CASP.

Hybrid systems can be described as hybrid automata (Henzinger 1996), which are finite state automata extended with continuous variables that evolve over time.

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 finite 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 $n \times n$ 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 $tr \in Trans$ being a tuple (ℓ, g, ξ, ℓ') where ℓ and ℓ' are the source and the target locations, respectively, g is the guard of tr (given as a linear constraint), and ξ is the update of tr (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°, and switched off when the temperature is greater than 21°. Finally, the invariants state that the heater can be on (off) *only* if the temperature is not greater than 22° (not less than 18°).

2.1 PDDL+ planning

Planning is an AI technology that aims at selecting and organizing activities in order to achieve specific goals (Nau *et al.* 2004). A planner uses a domain model, describing the actions through their pre- and post-conditions, and an initial state

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

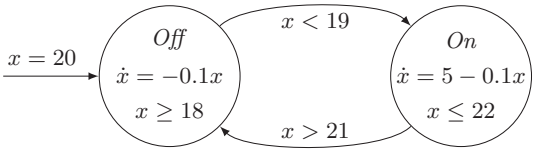


Fig. 1. Thermostat hybrid automaton.

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 synchronize activities between controlled devices and external events. PDDL+ is the planning formalism used to model hybrid system planning domains.

In the rest of this subsection, we introduce the concepts of planning instance and plan in the context of PDDL+ planning.

Definition 2 (*Planning instance*)
A planning instance is a pair $\mathscr{I} = (Dom, Prob)$, 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), given a planning instance $\mathscr{I}$, a *state* of $\mathscr{I}$ consists of a discrete component, described as a set of propositions P (the *Boolean fluents*), and a numerical component, described as a set of real variables $\mathbf{v}$ (the *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 specify how the action modifies the current state). The term "instantaneous" refers to the fact that instantaneous actions begin and end at the same timepoint. 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, respectively, the conditions that must hold when an action starts, the invariants 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 that are applied continuously while the action is executing.

Formally, the semantics of a PDDL+ planning instance (Definition 2) is given by hybrid automata (Definition 1).

The full syntax and semantics of PDDL+ can be found in the seminal paper (Fox and Long 2006). However, in the following, we provide a concrete example for

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

```
(:process off
 :parameters (?t - thermostat ?r - room )
 :condition (and (overall (>= (x ?r) 18)) (isOff ?t))
 :effect
  (and
   (decrease (x ?r) (* #t (* (x ?r) (-0.1)))))
 )
```

Fig. 2. Structure of a PDDL+ process.

```
(:action switchOff
 :parameters (?t - thermostat ?r - room )
 :precondition (and (isOn ?t) (> (x ?r) 21)))
 :effect
  (and (isOff ?t) (not (isOn ?t)))
 )
```

Fig. 3. Structure of a PDDL+ action.

the thermostat automaton in Figure 1, which also shows the main constructs of PDDL+.

Figure 2 shows the PDDL+ process representing the continuous change of the temperature when the thermostat is off (that corresponds to the *Off* mode in the hybrid automaton in Figure 1. Here, the parameters are the thermostat (variable `?t`) and the room (variable `?r`). The `condition` field specifies when the process is active, that is while the thermostat is off and the temperature of the room `(x ?r)` is greater than or equal to 18. The `effect` field specifies the continuous change that takes place while the process is active, and actually models the differential equation described in the hybrid automaton of Figure 1. A similar process can be defined for when the thermostat is on. Note that PDDL+ processes follow a *must* semantics, meaning that they are triggered as soon as the preconditions are true, and then become inactive as soon as they become false.

The other key element of a planning domain is the set of actions. Figure 3 shows an example of action for switching the thermostat off.

Here, the preconditions specify that the thermostat can be switched off only if it is currently on and if the temperature of the room is greater than 21. The action has two discrete effects that specify that the thermostat is now off and is no longer on. A similar action can be defined for switching the thermostat on. Note that, as opposed to processes, actions follow a *may* semantics, meaning that the planner may (or may not) decide to apply that action. Clearly, each action can only be applied if the precondition is satisfied.

For completeness, we also provide an example of *durative action* in Figure 4, from the mars rover, a typical domain in the AI planning community.

This durative action refers to the rover moving between two waypoints. The *duration* field specifies how long the action takes to complete, and in this case is function of the length of the path to be traversed. The condition field can specify conditions that must hold at start (i.e., before the action can actually start), at end (i.e., when the action terminates) and overall (i.e., throughout the duration of the

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

```
(:durative-action move
 :parameters (?r - rover ?wp1 ?wp2 - waypoint)
 :duration  (= ?duration  (* 2 (distance ?wp1 ?wp2))
 :condition (and (at start (at ?r ?wp1))
            (overall (> (batteryLevel ?r) 5)) (overall (visible ?wp1 ?wp2))
 :effect (and (at start (not (at ?r ?wp1)))
              (at end (at ?r ?wp2))
              (decrease (batteryLevel ?r) (* #t 0.4))
         )
)
```

Fig. 4. Structure of a PDDL+ durative action.

action). In this example, it is required that at start the rover is in the initial waypoint, and for the whole action the two waypoints are visible and there is enough battery charge. Similarly, the effects of the action can be applied when the action starts, when it terminates, or they can be continuous while the action is being applied. In this example, when the action starts the rover is no longer at waypoint wp1, it will be at waypoint wp2 when the action terminates, and while the action is being executed the battery level decreases continuously.

Once the planning instance has been defined, a planner can be used to find a *plan*, whose formal definition is given in the following.

Definition 3 (*Plan*)
A *plan* for a planning instance $\mathscr{I} = ((Fs, Rs, As, Es, Ps, arity), (Os, Init, G))$ 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 relation 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:

$$h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{2.1}$$

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. 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.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

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 (2.1) such that the body of the rule is satisfied by A, the head is satisfied by 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 is also a set of useful constructs, introduced informally in the following. 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

$$c_l \ \{m(\vec{X}) : \Gamma(\vec{X})\} \ c_u$$

where m is a relation symbol, $\vec{X}$ is a list of variables, $\Gamma(\vec{X})$ is a set of literals that include variables from $\vec{X}$, and c_l, c_u are non-negative integers. A choice rule intuitively states that, whenever the body is satisfied, the number of literals of the form $m(\vec{X})$ where $\Gamma(\vec{X})$ is satisfied must be between c_l and c_u. If not specified, the values of c_l and c_u are 0 and ∞, respectively. 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 satisfaction problems

A *CSP* (Smith 2006) is 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 (i.e., the

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

set of possible values that the variable can be assigned), and C is a set of constraints.[1] Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$, where σ is a vector of variables and ρ is a subset of the Cartesian product of the domains of such variables.

An *assignment* is a pair $\langle x_i, a \rangle$, where $a \in D_i$, whose intuitive meaning is that variable x_i is assigned value a. A *compound assignment* is a set of assignments to distinct variables from X. A *complete assignment* is a compound assignment to all the variables in X. A constraint $\langle \sigma, \rho \rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint.

Definition 6
A *solution* to a CSP $\langle X, D, C \rangle$ is a complete assignment satisfying every constraint from C.

Constraints can be represented either *extensionally*, by specifying the pair $\langle \sigma, \rho \rangle$, or *intensionally*, by specifying an expression involving variables, such as $x < y$. In this paper, we focus on constraints represented intensionally. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables (Katriel and van Hoeve 2006), such as $sum(x, y, z) < w$ and $all_different(x_1, \ldots, x_k)$.

One should notice that the mapping of an intensional constraint specification into a pair $\langle \sigma, \rho \rangle$ depends on the *constraint domain*. For example, the expression $1 \leqslant x < 2$ corresponds to the constraint $\langle \langle x \rangle, \{\langle 1 \rangle\} \rangle$ if the finite domain is considered, while it corresponds to $\langle \langle x \rangle, \{\langle v \rangle \mid v \in [1, 2)\} \rangle$ in a continuous domain. For this reason, and in line with the CLP Schema (Jaffar and Lassez 1987; Marriott *et al.* 2006), in this paper, we assume that a CSP includes the specification of the intended constraint domain.

2.4 Constraint ASP

CASP integrates ASP and CSP in order to deal with continuous dynamics. There is currently no standardized definition of CASP, and no standard language. Multiple definitions have been given in the literature (Baselice *et al.* 2005; Mellarkod *et al.* 2008; Balduccini 2009; Ostrowski and Schaub 2012). Although largely overlapping, these definitions and syntax differ from each other in some details.

To ensure generality of our results, in this paper, we introduce a simplified definition of CASP, which captures the common traits of the above approaches. The main results will be given using this simplified definition. Later (Section 4), we will introduce a specific CASP language and use it to describe a practical example.

Syntax. In order to accommodate CSP constructs, the language of CASP extends ASP by allowing *numerical constraints* of the form $x \bowtie y$, where $\bowtie \in \{<, \leqslant, =, \neq, \geqslant, >\}$, and x and y are *numerical variables*[2] or standard arithmetic terms possibly containing numerical variables, numerical constants, and ASP variables. Numerical constraints are only allowed in the head of rules.

[1] Strictly speaking, the use of the same index i across sets X and D in the above definition of the set of domains would require X and D to be ordered. However, as the definition of CSP is insensitive to the particular ordering chosen, we follow the approach, common in the literature on constraint satisfaction, of simply considering X and D as sets and abusing notation slightly in the definition of CSP.

[2] Numerical variables are distinct from ASP variables.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

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.

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 7
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 $\tau^{-1}(\gamma(A))$.

EZCSP *language.* In EZCSP, a program is a set of ASP rules written in such a way that their answer sets encode the desired CSPs. This is accomplished by using three types of special atoms:

(1) a *constraint domain declaration*, i.e., a statement of the form *cspdomain*($\mathscr{D}$), where $\mathscr{D}$ is a constraint domain such as fd, q, or r; informally, the statement says that the CSP is over the specified constraint domain (finite domains, rational numbers, real numbers), thereby fixing an interpretation for the intensionally specified constraints of the CSP. A program can contain only one *cspdomain* declaration;
(2) a *constraint variable declaration*, i.e., a statement of the form *cspvar*(x), where x is a ground term denoting a variable of the CSP;
(3) a *constraint statement*, i.e., a statement of the form *required*(γ), where γ is an expression that intensionally represents a constraint on (some of) the variables specified by the *cspvar* statements; intuitively, the statement says that the constraint represented by γ is required to be satisfied by any solution to the CSP.

For example, suppose that we are given integer variables $v(1)$, $v(2)$, $v(3)$ and we need to find values for them so that (i) each variable has a distinct value, and (ii) $v(1) + v(2) + v(3) \leqslant 6$ unless the constraint is blocked (atom *blocked*), in which case the constraint has no effect. A possible EZCSP encoding is

$$d(1).\ \ d(2).\ \ d(3). \qquad cspdomain(fd). \qquad cspvar(v(X)) \leftarrow d(X).$$
$$required(v(X) \neq v(Y)) \leftarrow d(X), d(Y), X \neq Y. \quad required(v(1) + v(2) + v(3) \leqslant 6) \leftarrow \text{not } blocked.$$

where X and Y are ASP variables used to describe compactly the uniqueness of values of $v(1)$, $v(2)$, $v(3)$.[3] The grounding of the program is

$$d(1).\ \ d(2).\ \ d(3). \qquad cspdomain(fd).$$
$$cspvar(v(1)) \leftarrow d(1). \qquad cspvar(v(2)) \leftarrow d(2). \qquad cspvar(v(3)) \leftarrow d(3).$$
$$required(v(1) \neq v(2)) \leftarrow d(1), d(2), 1 \neq 2. \quad required(v(1) \neq v(3)) \leftarrow d(1), d(3), 1 \neq 3. \quad \ldots$$
$$required(v(1) + v(2) + v(3) \leqslant 6) \leftarrow \text{not } blocked.$$

[3] For the interested reader, we note that such a constraint can be even more efficiently represented by a global constraint *all_different*, which is supported by EZCSP.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

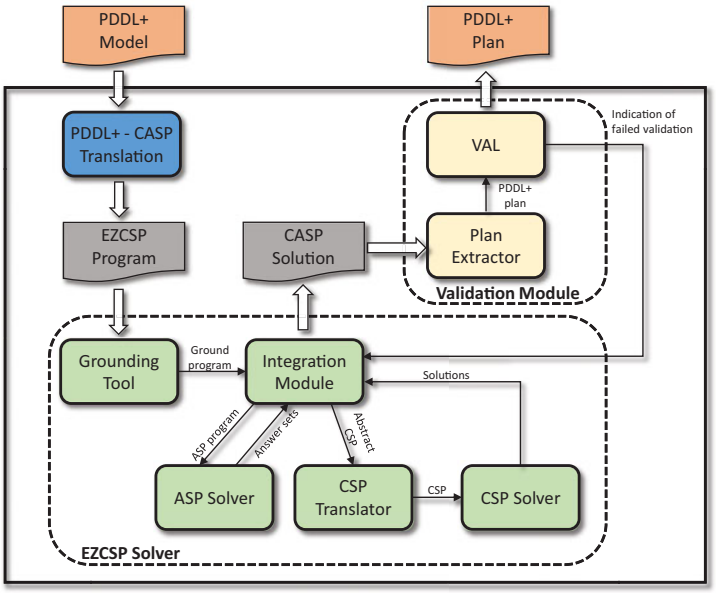


Fig. 5. CASP-based solution architecture.

The answer set of the program is $\{d(1), d(2), cspdomain(fd), cspvar(v(1)), cspvar(v(2)), cspvar(v(3)), required(v(1) \neq v(2)), required(v(1) \neq v(3), \ldots, required(v(1) + v(2) + v(3) \leqslant 6)\}$, which intuitively describes the numerical constraints given earlier. Note that constraint $v(1) + v(2) + v(3) \leqslant 6$ is included in the answer set due to the fact that *blocked* does not hold.

3 A CASP-based solution for planning in hybrid domains

In this section, we first present a CASP-based architecture for solving PDDL+ problems. Then, we focus on the CASP encoding of PDDL+ problems.

3.1 Architecture

Our approach is centered around the architecture outlined in Figure 5. A PDDL+ model is first translated to a CASP program Π. Then, the CASP solutions of Π are computed by a CASP solver. The architecture is independent of the CASP solver used, but, for presentation purposes, we focus on CASP solver EZCSP. In EZCSP, the Grounding Tool maps Π to its syntactically legal ASP counterpart, $\tau(\Pi)$, and grounds it, obtaining program Π_g. Next, the Integration Module carries out an iterative process in which

(1) an ASP solver is used to find an answer set A of Π_g;
(2) a CSP C is obtained from A by finding all numerical constraints that are denoted by some atom of A, i.e., $C = \{\tau^{-1}(a) \mid a \in \gamma(A)\}$;
(3) a CSP solver is used to find a solution, α, of C;

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

(4) if C has no solution, the ASP solver is used to find a new answer set of Π_g and the process is iterated; otherwise, the CASP solution $\langle A, \alpha \rangle$ is returned.

More details on the EZCSP algorithm can be found in Balduccini (2009). Details on other CASP solvers can be found in Section 7; Section 5 includes experiments that we conducted using CASP solver CLINGCON.

At this point, the Validation Module is executed. The module ensures that the CASP solution found represents a valid PDDL+ plan. As we will see later, in our encoding invariants are enforced at the beginning and at the end of every state in which durative actions and processes are in execution. In this sense, the continuous timeline is discretized. This may cause some plans found by the planner to be invalid: in fact, it is possible for an invariant to be satisfied at the boundaries of each state, and yet be violated within that interval. An example of this behavior is discussed in Section 6. To handle these situations, we follow the discretize-and-validate approach first proposed in Della Penna *et al.* (2009). In that approach, a plan found in the discretized setting is validated using the plan validator VAL (Howey *et al.* 2004), which is capable of carrying out the calculations needed to check the satisfaction of invariants within state boundaries. The Validation Module consists of two components, the Plan Extractor and VAL. The Plan Extractor component applies syntactic transformations to translate the CASP solution to a PDDL+ plan in the input language of VAL. The plan is then passed to VAL. If VAL finds it to be valid, the plan is returned and the process terminates. Otherwise, EZCSP is invoked again. Suitable constraints are added to the EZCSP program to rule out previously found CASP solutions and the process is iterated.

3.2 Encoding

This section is dedicated to discussing 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 *et al.* 2005). It builds upon the existing SAT-based (Kautz and Selman 1992) and ASP-based planning approaches 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 *et al.* 2005). States have a duration, given by the global time at which a state

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

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.

Next, we describe the CASP formalization of PDDL+ models. We begin by discussing the correspondence between global time and states; then, we present domain, problems, and planning task encodings.

3.2.1 Time, states, fluents

The global time at which the state at step i begins is represented by the numerical variable $tstart(i)$. Similarly, the end time is represented by $tend(i)$. A fluent literal is a fluent f or its negation $\neg f$. By $\bar{l}$ we denote Boolean complement over fluent literals, i.e., $\overline{f} = \neg f$ and $\overline{\neg f} = f$. If f holds at discrete time step i, in the encoding we write $holds(f, i)$; if $\neg f$ holds, i.e., f is false, we write $\neg holds(f, i)$. We also often use $\chi(l, i)$ as an abbreviation, corresponding to $holds(f, i)$ if $l = f$ and to $\neg holds(f, i)$ if $l = \neg f$. 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.

3.2.2 Domain encoding

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

Instantaneous actions. As we have seen, instantaneous actions are characterized by a set of preconditions and a set of effects. A Boolean precondition is a fluent literal l, representing a condition on the truth value of l. A numerical precondition is an inequality between mathematical expressions involving combinations of numerical fluents and numerical constants. The encoding of a precondition depends on its type. Thus, every Boolean precondition l of an action a is encoded by a denial:

$$\leftarrow \chi(\bar{l}, I), occurs(a, I). \tag{3.2}$$

Notice that a separate denial is used for every precondition. To illustrate the encoding, consider an example in which an action $refuel_with(tk1)$, representing the

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

refueling of some machine using refuel tank $tk1$, has a precondition that tank $tk1$ must be available, represented by $avail(tk1)$. The precondition is thus represented by

$$\leftarrow \neg holds(avail(tk1), I), occurs(refuel_with(tk1), I).$$

A numerical precondition γ is encoded by means of a numerical constraint γ^f obtained from γ by replacing every occurrence of a numerical fluent n by an expression $v_final(n, I)$. The latter represents the value of n at the end of time step I. Thus, for every precondition γ of an action a, the encoding includes a rule:

$$\gamma^f \leftarrow occurs(a, I). \tag{3.3}$$

Continuing on the previous example, if action $refuel_with(tk1)$ has a precondition $level(tk1) > 0$, intuitively meaning that the tank must not be empty, the precondition is encoded by a rule:

$$v_final(level(tk1), I) > 0 \leftarrow occurs(refuel_with(tk1), I).$$

Similarly to Boolean preconditions, Boolean effects are represented in PDDL+ by fluent literals. For every Boolean effect l of an action a, the encoding includes a rule:

$$\chi(l, I + 1) \leftarrow occurs(a, I). \tag{3.4}$$

The rule states that l is true at the next time step $I + 1$ if the action occurs at (the end of) step I. For the effects of actions on numerical fluents, we focus our presentation on the assignment of a value to a numerical fluent,[4] corresponding to PDDL+ expression (`assign n e`) where n is a numerical fluent and e is a mathematical expression possibly including numerical fluents. An effect of this form for an action a is represented by a rule:

$$v_initial(n, I + 1) = e' \leftarrow occurs(a, I), \tag{3.5}$$

where e' is obtained from e by replacing every occurrence of a numerical fluent n' by $v_initial(n', I + 1)$.

Durative actions. Recall that a durative action is characterized by two sets of conditions: the start conditions, which are analogous to the preconditions of instantaneous actions, and the *invariants*, i.e., conditions that must be true throughout the execution of the action. Syntactically, start conditions and invariants are represented like the preconditions of instantaneous actions, i.e., with fluent literals and numerical inequalities. The effects of a durative action are divided in at-start effects, at-end effects, and continuous effects. The at-start and at-end effects take place at the beginning and at the end of the action and are treated analogously to the effects of instantaneous actions. The continuous effects only apply to numerical fluents and describe a change of value that is a function of the time elapsed since the start of the action. That is, the continuous effects of durative action d can be viewed as a partial function associating to every numerical fluent n affected by d a mathematical expression $e_n(t)$ that is a function of time (and, possibly, of the

[4] The handling of additive fluents is discussed in the context of durative actions.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

value of other numerical fluents). This representation makes it possible to capture the behavior of additive fluents. Additionally, the duration of the action is specified by an inequality $\delta \bowtie e$, where δ represents the duration and e is an mathematical expression possibly including numerical fluents.

In our formalization, a durative action d is encoded by means of two instantaneous actions, $start(d)$ and $end(d)$. The start conditions of d are mapped to preconditions of $start(d)$ and encoded using (3.2) and (3.3). Note that a durative action typically affects multiple consecutive states. Our encoding introduces a special Boolean fluent $inprogr(d)$ to denote the states during which a durative action d is in progress. How this fluent is made true/false is discussed later. A Boolean invariant, l, of a durative action d is encoded by a denial:

$$\leftarrow \chi(\bar{l}, I), holds(inprogr(d), I). \tag{3.6}$$

It is worth noticing the similarity with (3.2). A numerical invariant γ is encoded by means of two rules:

$$\gamma^i \leftarrow holds(inprogr(d), I), \tag{3.7}$$

$$\gamma^f \leftarrow holds(inprogr(d), I), \tag{3.8}$$

where γ^f is defined as for (3.3) and γ^i is obtained from γ by replacing every occurrence of a numerical fluent n by an expression $v_initial(n, I)$. Note that these rules enforce the invariants only at the beginning and at the end of every state affected by the action. The task of checking whether the conditions hold at other timepoints is discussed later.

The start and end effects of a durative action d can be easily viewed as the effects of the $start(d)$ and $end(d)$ actions and encoded using (3.4) and (3.5). To account for the special Boolean fluent $inprogr(d)$, in our encoding the start and end effects of d are expanded to include, respectively, $inprogr(d)$ and $\neg inprogr(d)$.

To account for the effects of concurrent actions on additive fluents, our encoding considers separately the contributions from each action by introducing a special numerical variable $v(contrib(n, d), i)$ for every durative action d, numerical fluent n affected by d, and time step i. Recall that the continuous effect of d on n is described by a mathematical expression $e_n(t)$. This is encoded in CASP by means of a rule:

$$v(contrib(n, d), I) = e_n^i \leftarrow holds(inprogr(d), I), \tag{3.9}$$

where e_n^i is obtained from $e_n(t)$ by replacing every occurrence of a numerical fluent n' by $v_initial(n', I)$ and every occurrence of t by the expression $tend(I) - tstart(I)$. For example, if a $refuel$ action causes the level of fuel in a tank to increase linearly with time, its effect is encoded by

$$v(contrib(fuel_level, refuel), I) = 1 \cdot (tend(I) - tstart(I)) \leftarrow holds(inprogr(d), I).$$

The cumulative effects of durative actions are calculated by means of the following rule of the encoding:

$$v_final(N, I) = v_initial(N, I) + \textstyle\sum_{d \in D} v(contrib(N, d), I), \tag{3.10}$$

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

where N ranges over additive fluents and D over durative actions. Intuitively, for every numerical fluent n, the rule accumulates the effects of all durative actions on it and updates the value of n by the net result of such effects. For instance, in a domain in which $fuel_level$ is affected, possibly concurrently, by actions $refuel$ and $consume$, their cumulative effects are modeled by

$$v_final(fuel_level, I) = v_initial(fuel_level, I) + \sum_{d \in \{consume, refuel\}} v(contrib(fuel_level, d), I).$$

Note that implementing the summation from (3.10) in an actual CASP language may require the introduction of additional numerical variables and rules because of syntax restrictions of the implemented languages. In the next section, we show how (3.10) is implemented in the case of EZCSP.

The encoding also includes a rule:

$$v_final(N, I) = v_initial(N, I) \leftarrow \text{not } ab(N, I), \tag{3.11}$$

i.e., a default stating that every additive fluent maintains its value unless explicitly changed. Intuitively, atom $ab(n, i)$ states that fluent n is an exception to the default at time step i. Default (3.11) must be blocked in every state in which actions affect the fluent. This is accomplished by including in the encoding, for every action d affecting a fluent n, a rule:

$$ab(n, I) \leftarrow holds(inprogr(d), I). \tag{3.12}$$

Following the semantics of PDDL+, action $end(d)$ is automatically triggered after $start(d)$. For every durative action d, the task of finding a time step at which the end action occurs is accomplished by a choice rule:

$$1\{occurs(end(d), I2) : I2 > I1\}1 \leftarrow occurs(start(d), I1). \tag{3.13}$$

If time step i is selected for the occurrence of $end(d)$, the timepoint at which the action occurs is given by the value of $tend(i)$. Notice that, unlike the selection of a time step for $end(d)$, there is no need to state explicitly that a value must be selected for $tend(d)$. The semantics of CASP ensures that a value is selected for every numerical variable in the constraint problem.

Finally, the inequality $\delta \bowtie e$ on the duration of a durative action d is encoded by a rule:

$$(tend(I2) - tend(I1)) \bowtie e^f \leftarrow occurs(end(d), I2), occurs(start(d), I1), \tag{3.14}$$

where e^f is obtained from e by replacing every occurrence of a numerical fluent n by $v_final(n, I_1)$.

It is worth noting that the encoding extends to supporting multiple occurrences of the same durative action in a natural way. This is accomplished by adding an argument to the name of the action, i.e., instead of writing d we write $d(i)$. The argument represents the time step at which the action starts. For example, (3.14) becomes

$$(tend(I2) - tend(I1)) \bowtie e^f \leftarrow occurs(end(d(I)), I2), occurs(start(d(I)), I1).$$

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Intuitively, this approach yields multiple, and completely independent, "copies" of the durative action, whose effects and termination can be handled accordingly by the encoding presented. For simplicity of presentation, throughout the paper we make the assumption that durative actions occur only once and thus adopt the simpler writing d.

Processes and events. The encoding of processes and events builds upon the approach outlined above, respectively, for durative and instantaneous actions. That is, an event is encoded using (3.2)–(3.5), while a process is encoded using (3.6)–(3.14). However, recall that their triggering is defined by PDDL+'s *must* semantics (see Section 2). In our encoding, this is captured by a choice rule combined with numerical constraints. Let $l_1, \ldots, l_k$ be the Boolean preconditions of an event v and $\gamma_1, \ldots, \gamma_m$ be its numerical preconditions. The encoding includes a choice rule[5]:

$$1\{occurs(v,I), is_false(\gamma_1,I), \ldots, is_false(\gamma_m,I)\}1 \leftarrow \chi(l_1,I), \ldots, \chi(l_k,I). \tag{3.15}$$

Intuitively, when the Boolean conditions of the event are satisfied, the choice rule states that the event will be triggered unless it is inhibited by unsatisfied numerical conditions. Given a numerical condition γ_i, let $\overline{\gamma_i}$ denote its complement, e.g., $>$ is replaced by $\leqslant$ and $=$ by $\neq$. This part of the encoding is completed by introducing, for every γ_i, a rule:

$$\overline{\gamma_i}^{\,f} \leftarrow is_false(\gamma_i,I), \tag{3.16}$$

where operator ^{f} is defined as for (3.3). The rule intuitively states that if a numerical condition is hypothesized to be unsatisfied, then the corresponding numerical fluents must have values that falsify the condition. It is worth observing how the must semantics is obtained: as soon as all numerical conditions are satisfied by the current state i, (3.16) ensures that $is_false(\gamma_i, i)$ must be false for each of them. If all Boolean conditions are also satisfied, then (3.15) is forced to make $occurs(v,i)$ true, thus triggering the occurrence of the event. The triggering of a process p is handled in an analogous way, except that $start(p)$ is the event triggered by (3.15) rather than v. Also note that, while one ma1y be tempted to add a rule complementary to (3.16) such as

$$\gamma_i^f \leftarrow \text{not } is_false(\gamma_i,I), occurs(v,I),$$

this is made unnecessary by the presence of (3.3). To illustrate our approach, consider an example in which a process *generate* has start preconditions *enabled* and $fuel_level > 0$. The corresponding encoding is

$$\begin{array}{l} 1\{occurs(start(generate),I), is_false(fuel_level > 0, I)\}1 \leftarrow holds(enabled,I). \\ v_final(fuel_level,I) \leqslant 0 \leftarrow is_false(fuel_level > 0, I). \end{array} \tag{3.17}$$

[5] Due to the syntactic restrictions of CASP solvers, in practice $\gamma_1, \ldots \gamma_m$ are replaced by syntactically correct symbols that are in one-to-one correspondence with them.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

The encoding is completed by the domain-independent rules:

$$\begin{aligned} &tstart(I+1) = tend(I). \\ &v_initial(N, I+1) = v_final(N, I). \\ &holds(F, I+1) \leftarrow holds(F, I), \text{not } \neg holds(F, I+1). \\ &\neg holds(F, I+1) \leftarrow \neg holds(F, I), \text{not } holds(F, I+1). \end{aligned} \tag{3.18}$$

The first rule ensures that there are no gaps between the time intervals associated with consecutive states. The other rules handle propagation of fluent values from one state to the next.

3.2.3 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, where P and v are the sets of Boolean and numerical fluents, respectively.

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 $\mathscr{I}$, by $\Pi(\mathscr{I})$ we denote the CASP encoding of $\mathscr{I}$. Next, we turn our attention to the planning task.

3.2.4 Planning task

Our approach to planning leverages techniques from ASP-based planning (Lifschitz 2002; Balduccini *et al.* 2006). The planning task is specified by the planning module, M, which includes a choice rule of the form:

$$\lambda\{occurs(a_1, I), occurs(a_2, I), \ldots, occurs(a_k, I)\}\mu,$$

where $a_1 \ldots a_k$ are actions as defined by the PDDL+ specification and bounds λ, μ allow for control on concurrency of the actions. In practice, planning modules often also include heuristics for the purpose of increasing performance, also in terms of plan quality.

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

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 (2003).

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

4 An example of our encoding

In this section, we provide an encoding of a PDDL+ instance (a domain and one problem instance that will be used also in the experimental analysis) into a CASP program.

The specific domain is described in the first subsection, then its PDDL model is given, while the third subsection is devoted to the encoding into CASP.

4.1 The *generator* domain

The *generator* domain is well known across the planning community and has been a testbed for many planners. The problem revolves around refueling a diesel powered generator which has to run for a given duration without overflowing or running dry. This problem is interesting because the time at which the refuel action must be applied within the generate action is critical: the fuel level in the tank must not exceed the capacity of the tank so the planner must realize that the refuel action must be delayed at least X time units after the start of the generate action, but must also occur at least Y time units before the end of the generate action to prevent the fuel level falling below zero, where X and Y depend on the instance.

4.2 PDDL+ model

The PDDL+ domain of the generator is shown, in slightly simplified form, in Figure 6. The `generate` process models the generator running. Its invariant, specified by the `overall` condition, requires the fuel level in the generator to be no less than 0 at all times during the execution of the process. 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.

Durative action `refuel` formalizes the use of a tank for refueling the generator. The action has an invariant stating that the fuel level in the generator must not exceed the capacity of the tank, and is required to have a duration of 10 time units. The action has the effect of continuously increasing the fuel level of the generator's tank. Note that the rate at which the generator's tank is refilled is twice as high as the rate at which fuel is consumed. When both actions are being executed, their effects are combined. Given that the duration of the refuel action is fixed, the total contribution of a refuel tank is 20 units of fuel.

The PDDL+ problem models the instance, as shown in Figure 7. In this example, we have one tank, the initial fuel level in the generator is 990, and the capacity of the generator is 1,000 units. The goal for generator is to run for 1,000 time units.

4.3 The EZCSP encoding

The encoding assumes the existence of a relation *step*(·) defining the (finite) range of integers that represent the discrete steps in the evolution of the domain. Constant

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

```
(define (domain linear_generator)
 (:requirements :fluents :durative-actions :duration-inequalities :adl :typing)
 (:types generator)
 (:predicates (generator-ran) (refueling ?g - generator))
 (:functions (fuelLevel ?g - generator) (capacity ?g - generator))

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

 (:durative-action refuel
  :parameters (?g - generator ?t - tank)
  :duration  (= ?duration  10)
  :condition (overall (< (fuelLevel ?g) (capacity ?g)))
  :effect (increase (fuelLevel ?g) (* #t 2))
 )
)
```

Fig. 6. PDDL+ domain of the linear generator.

```
(define (problem linear-generator-prob)
        (:domain generator)
        (:objects gen - generator tank1 - tank)
        (:init
           (= (fuelLevel gen) 990)
           (= (capacity gen) 1000)
           (available tank1))
        (:goal (= generator_time 1000))
)
```

Fig. 7. PDDL+ problem of the linear generator.

last_step is set to the largest such integer, and determines the maximum number of steps in the plan, provided as an external parameter to the planner.

Domain-independent component. We begin the description of the encoding from its domain-independent portion. Note that EZCSP requires the explicit declaration of all numerical variables. Thus, the first rules we show capture the declarations of the variables that encode the start and end times of states, accompanied by constraints specifying their domain, i.e., they must be non-negative and have values such that the corresponding states have non-negative duration:

$$cspvar(tstart(I)) \leftarrow step(I). \qquad required(tstart(I) \geqslant 0) \leftarrow step(I).$$
$$cspvar(tend(I)) \leftarrow step(I). \qquad required(tend(I) \geqslant 0) \leftarrow step(I).$$
$$required(tend(I) \geqslant tstart(I)) \leftarrow step(I).$$

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

The next axioms encode the first and last set of rules from (3.18), namely stating that every state ends at the same time in which the next state begins, and that a Boolean fluent maintains its value unless forced to change. For convenience of representation, the remaining rule from (3.18) is written in a domain-dependent way, and thus introduced in the next paragraph:

$$required(tstart(I2) = tend(I1)) \leftarrow step(I1), step(I2), I2 = I1 + 1.$$

$$holds(F, I2) \leftarrow fluent(F), step(I1), step(I2), I2 = I1 + 1, holds(F, I1), not\ \neg holds(F, I2).$$
$$\neg holds(F, I2) \leftarrow fluent(F), step(I1), step(I2), I2 = I1 + 1, \neg holds(F, I1), not\ holds(F, I2).$$

Domain-dependent component. At the core of the domain encoding is the formalization of process *generate*, durative action *refuel*, and of how they affect the fuel level in the generator's tank. Let us start with *generate*.

The generator's fuel level is modeled by additive numerical fluent $fuel_level$. The next set of rules declares the numerical variables representing the value of the fluent at the beginning and at the end of each state, and sets limits on their values ($tankcap(\cdot)$ is part of CASP encoding of the problem instance, and indicates the capacity of the tank):

$$cspvar(v_initial(fuel_level, I)) \leftarrow step(I).$$
$$required(v_initial(fuel_level, I) \geqslant 0) \leftarrow step(I).$$
$$required(v_initial(fuel_level, I) \leqslant TC) \leftarrow step(I), tankcap(TC).$$

$$cspvar(v_final(fuel_level, I)) \leftarrow step(I).$$
$$required(v_final(fuel_level, I) \geqslant 0) \leftarrow step(I).$$
$$required(v_final(fuel_level, I) \leqslant TC) \leftarrow step(I), tankcap(TC).$$

The next axiom is an instantiation, to numerical fluent $fuel_level$, of the second rule from (3.18) and states that the value of $fuel_level$ at the end of a state coincides with its value at the beginning of the next:

$$required(v_initial(fuel_level, I2) = v_final(fuel_level, I1)) \leftarrow step(I1), step(I2), I2 = I1 + 1.$$

Next, we turn our attention modeling how the fluent changes over time, beginning with (3.10). As mentioned earlier, contributions to additive fluents are represented by means of numerical variables of the form $v(contrib(\cdot), \cdot)$. Because the sum operator from (3.10) is not supported directly by the syntax of EZCSP, the EZCSP encoding includes auxiliary rules and variables, which calculate, separately, the total positive and negative contributions from the sum of the relevant contributions. The positive and negative contributions are distinguished by an extra argument (*incr* or *decr*) in the name of variable $v(contrib(\cdot), \cdot)$:

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

$cspvar(v(contrib(fuel_level, decr), I)) \leftarrow step(I).$
$required(v(contrib(fuel_level, decr), I) \leqslant 0) \leftarrow step(I).$

$decr(I, v(contrib(fuel_level, decr, D), I)) \leftarrow step(I), cspvar(v(contrib(fuel_level, decr, D), I)).$
$required(sum([decr(I)/2], =, v(contrib(fuel_level, decr), I))) \leftarrow step(I).$

$cspvar(v(contrib(fuel_level, incr), I)) \leftarrow step(I).$
$required(v(contrib(fuel_level, incr), I) \geqslant 0) \leftarrow step(I).$

$incr(I, v(contrib(fuel_level, incr, D), I)) \leftarrow step(I), cspvar(v(contrib(fuel_level, incr, D), I)).$
$required(sum([incr(I)/2], =, v(contrib(fuel_level, incr), I))) \leftarrow step(I).$

Above, ASP variable D ranges over all possible durative actions that may cause a contribution. Numerical variables $v(contrib(fuel_level, decr), I)$ and $v(contrib(fuel_level, incr), I)$ capture the negative and positive contributions at a given step. Relations $decr(\cdot, \cdot)$ and $incr(\cdot, \cdot)$ are used to link group the contributions at each time step. For example, a fact $decr(1, v(contrib(fuel_level, decr, generate), 1))$ states that $v(contrib(fuel_level, decr, generate), 1))$ is one of the (negative) contributions to the fuel level at step 1. The constraint $required(sum([decr(I)/2], = , v(contrib(fuel_level, decr), I)))$ sums all of the negative contributions for time step I and assigns the total to numerical variable $v(contrib(fuel_level, decr), I)$. The summation of the positive contributions works in a similar way. The EZCSP counterpart of (3.10) is a rule stating that the value of $fuel_level$ is the sum of the total positive and negative contributions:

$$required(v_final(fuel_level, I) = v_initial(fuel_level, I) + v(contrib(fuel_level, incr), I) - v(contrib(fuel_level, decr), I)) \leftarrow step(I).$$

Next, we focus on the *generate* process. The next rule defines Boolean fluent $inprogr(generate)$, which indicates whether the process is in progress:

$$fluent(inprogr(generate)).$$

Next, we define a numerical variable formalizing *generate*'s contribution to the fuel level of the generator's tank:

$$cspvar(v(contrib(fuel_level, decr, generate), I)) \leftarrow step(I).$$
$$required(v(contrib(fuel_level, decr, generate), I) \geqslant 0) \leftarrow step(I).$$

The contribution provided by *generate* while the process is in progress is calculated as per (3.9):

$$required(v(contrib(fuel_level, decr, generate), I) = 1 * (tend(I) - tstart(I))) \leftarrow step(I), holds(inprogr(generate), I).$$

The next set of rules follows (3.11) and (3.12), expressing the fact that, by default, the contribution by *generate* is 0. This is elegantly represented by means of default

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

negation:

$$ab(contrib(fuel_level, decr, generate), I) \leftarrow step(I), holds(inprogr(generate), I).$$

$$required(v(contrib(fuel_level, decr, generate), I) = 0) \leftarrow step(I), \text{not } ab(contrib(fuel_level, decr, generate), I).$$

Next, we model the two (instantaneous) actions that trigger the start and the end of the process. The actions are defined by

$$action(start(generate)). \quad action(end(generate)).$$

The following rule is the counterpart of (3.13) and ensures that action $end(generate)$ is triggered at some time step following $start(generate)$:

$$1\{occurs(end(generate), I2) : step(I2) : I2 > I1 : I2 < last_step\}1 \leftarrow step(I), occurs(start(generate), I1).$$

The next set of rules formalizes the effect of starting and stopping the process.

$$holds(inprogr(generate), I2) \leftarrow step(I1), step(I2), I2 = I1 + 1, occurs(start(generate), I1).$$

$$\neg holds(inprogr(generate), I2) \leftarrow step(I1), step(I2), I2 = I1 + 1, occurs(end(generate), I1).$$

The preconditions and the triggering of action $start(generate)$, which follows the *must* semantics, are encoded according to (3.3), (3.15), and (3.16):

$$required(v_final(fuel_level, I) \geqslant 0) \leftarrow step(I), occurs(start(generate), I).$$

$$1\{occurs(start(generate), I), is_false(fuel_level \geqslant 0, I)\}1 \leftarrow step(I), I < last_step, \neg holds(inprogr(generate), I).$$

$$required(v_final(fuel_level, I) < 0) \leftarrow step(I), is_false(fuel_level \geqslant 0, I).$$

Durative action $refuel$ (instantiated for each refuel tank) is modeled along the same lines. First of all, fluents and numerical variables are defined to represent whether the action is in progress and the action's contribution to the generator's fuel level (variable TK ranges over the refuel tanks from the problem):

$$fluent(inprogr(refuel(TK))) \leftarrow refuel_tank(TK).$$

$$cspvar(v(contrib(fuel_level, incr, refuel(TK)), I)) \leftarrow step(I), refuel_tank(TK).$$
$$required(v(contrib(fuel_level, incr, refuel(TK)), I) \geqslant 0) \leftarrow step(I), refuel_tank(TK).$$

The contribution to $fuel_level$ while $refuel$ is in progress is defined similarly to that of $generate$:

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

$$required(v(contrib(fuel_level, incr, refuel(TK)), I) = 2 * (tend(I) - tstart(I))) \leftarrow step(I), refuel_tank(TK), holds(inprogr(refuel(TK)), I).$$

$$ab(contrib(fuel_level, incr, refuel(TK)), I) \leftarrow step(I), refuel_tank(TK), holds(inprogr(refuel(TK)), I).$$

$$required(v(contrib(fuel_level, incr, refuel(TK)), I) = 0) \leftarrow step(I), refuel_tank(TK), not\ ab(contrib(fuel_level, incr, refuel(TK)), I).$$

Next, the instantaneous actions that correspond to the start and to the end of the durative action are introduced, and their effects defined:

$$action(start(refuel(TK))) \leftarrow refuel_tank(TK).$$
$$action(end(refuel(TK))) \leftarrow refuel_tank(TK).$$

$$1\{occurs(end(refuel(TK)), I2) : step(I2) : I2 > I1 : I2 < last_step\}1 \leftarrow step(I), refuel_tank(TK), occurs(start(refuel(TK)), I1).$$

$$holds(inprogr(refuel(TK)), I2) \leftarrow step(I1), step(I2), I2 = I1 + 1, refuel_tank(TK), occurs(start(refuel(TK)), I1).$$

$$\neg holds(inprogr(refuel(TK)), I2) \leftarrow step(I1), step(I2), I2 = I1 + 1, refuel_tank(TK), occurs(end(refuel(TK)), I1).$$

Finally, the requirement on the duration of *refuel* from the PDDL+ specification is captured according to (3.14), i.e., a rule stating that, whenever *refuel* ends, the time elapsed between its start and its end must equal the duration specified (variable RT represents the duration of the refuel action; for convenience of representation, the duration is parameterized by relation $duration(\cdot, \cdot)$ rather than being expressed directly by a constant):

$$required(tend(I2) - tend(I1)) = RT) \leftarrow step(I1), step(I2), refuel_tank(TK), duration(refuel(TK), RT), occurs(end(refuel(TK)), I2), occurs(start(refuel(TK)), I1).$$

Planning module. Action selection is achieved by a choice rule stating that refuel actions are allowed to occur at any time:

$$0\{occurs(start(refuel(TK)), I) : step(I)\}1 \leftarrow refuel_tank(TK).$$

In order to improve performance, the planning module includes a heuristic saying that at least an action must be executed at every time step:

$$some_action(I) \leftarrow occurs(A, I).$$
$$\leftarrow step(I), I < last_step, \text{not } some_action(I).$$

Problem encoding. Relation $step(\cdot)$ is thus defined by

$$step(0..last_step).$$

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

The capacity of the generator's tank is given by

$$tankcap(1,000).$$

The set of available fuel tanks and the duration of the *refuel* action are specified by

$$refuel_tank(tank1).$$
$$duration(refuel(TK),10) \leftarrow refuel_tank(TK).$$

The initial state is specified in the following, with a set of rules that specifies the initial fuel level, states that no process or durative action is initially in progress, and sets the clock to 0:

$$required(v_initial(fuel_level,0) = 990).$$
$$\neg holds(inprogr(generate),0).$$
$$\neg holds(inprogr(refuel(TK)),0) \leftarrow refuel_tank(TK).$$
$$required(tstart(0) = 0).$$

Finally, the PDDL+ goal (= *generator_time* 1000) is formalized by

$$duration(generate,1000).$$

$$required(tend(I2) - tend(I1) = D) \leftarrow step(I1), step(I2), duration(generate,D), occurs(end(generate),I2), occurs(start(generate),I1).$$

EZCSP *solution.* A CASP solution produced by EZCSP for the above encoding of domain and problem is shown next. For clarity, we focus on the restriction of the solution to relation *occurs* and to numerical variable $tend(\cdot)$:

$$occurs(start(generate),0)$$
$$occurs(start(refuel(tank1)),0)$$
$$tend(0) = 0.000$$

$$occurs(end(refuel(tank1)),1)$$
$$tend(1) = 10.000$$

$$occurs(end(generate),2)$$
$$tend(2) = 1000.000$$

This solution corresponds to the PDDL+ plan in which *generate* begins immediately (time 0.000) and lasts for 1000.000 time units, and refuel begins immediately and lasts for 10.0 time units:

$$0.000 : \ generate \quad [1000.000]$$
$$0.000 : \ refuel(tank1) \quad [10.000]$$

5 Experiments on PDDL+ planning domains

We performed an empirical evaluation of the run-time performance of our approach. The comparison was with the state-of-the-art PDDL+ planners dReal (Bryce *et al.*

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

2015) and UPMurphi. Although SpaceEx (Bogomolov *et al.* 2014) is indeed a related approach, it was not included in the 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 4.00 GHz. 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,[6] with gringo 3.0.5[7] and clasp 3.1.3[8] as grounding tool and ASP solver, and B-Prolog 7.5[9] and GAMS 24.5.7[10] as constraint solvers. The former was used for all linear problems and the latter for the non-linear ones. It should be noted that GAMS acts as a front-end to a large set of constraint solvers. The underlying constraint solver to be used by GAMS is selected at run-time. We performed a thorough evaluation of the underlying solvers on the domains they support.

In an attempt to evaluate the role of the encoding and of the solver in the resulting performance, we also created variants of our encodings, discussed later, suitable for CLINGCON, and studied their performance. For this part of the experiments, we used CLINGCON 2.0.3.[11] Note that CLINGCON supports only integer variables and linear constraints, and was thus applied only to the linear variants of the domains considered. The other systems used were dReal 2.15.11,[12] configured as suggested by its authors, and UPMurphi 3.0.2.[13]

The experiments were conducted on the linear and non-linear versions of the *generator* and *car* domains. The CASP encodings were created manually as described earlier. Because of the different ways in which dReal and UPMurphi operate, the comparison with dReal was based on finding a single plan with a given maximum time step (called here *fixed-time step* experiments), as discussed in Byrce *et al.* (2015). The results are summarized in Tables 1–3. On the other hand, the comparison with UPMurphi was based on the cumulative times for finding a single plan by progressively increasing the maximum time step (referred to as *cumulative-time* experiments). The results are reported in Tables 4–6.

In the tables, entries marked "-" indicate a timeout (threshold 600 s). Entries marked "*" indicate missing entries due to licensing limitations (see below). Remarkably, we found that, in all instances, VAL determined that the CASP solutions returned represented valid PDDL+ plans. More information on the topic of validation, as well as an example in which validation fails, can be found in Section 6. Next, we discuss the experimental results obtained for each domain.

[6] http://mbal.tk/ezcsp/
[7] http://sourceforge.net/projects/potassco/files/gringo/
[8] https://sourceforge.net/projects/potassco/files/clasp/
[9] http://www.picat-lang.org/bprolog/
[10] http://www.gams.com/
[11] https://sourceforge.net/projects/potassco/files/clingcon/
[12] http://dreal.github.io/
[13] https://github.com/gdellapenna/UPMurphi/

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Table 1. *Car domain with fixed time step. Results in seconds. Problem instances refer to max acceleration*

Domain	Solver	1	2	3	4	5	6	7	8
Car linear	EZCSP (B-Prolog)	0.32	0.31	0.32	0.32	0.32	0.30	0.31	0.31
	CLINGCON	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	dReal	1.11	1.11	1.15	1.14	1.19	1.13	1.14	1.19
Car non-linear	EZCSP (GAMS/couenne)	1.41	0.38	0.49	1.1	0.42	0.52	0.43	2.62
	EZCSP (GAMS/knitro)	0.71	0.68	0.29	0.39	0.25	0.25	0.26	0.84
	dReal	58.21	162.60	-	-	-	-	-	-

Table 2. *Linear variant of generator with fixed time step. Results in seconds. Problem instances refer to number of tanks*

Encoding	Solver	1	2	3	4	5	6	7	8
Basic	EZCSP (B-Prolog)	5.82	2.19	41.77	74.51	114.86	424.80	164.95	-
	CLINGCON	16.88	62.62	-	-	-	-	-	52.43
	CLINGCON/Opt	17.15	62.47	51.76	-	-	-	-	-
Heuristic	EZCSP (B-Prolog)	0.28	1.03	4.21	7.25	27.08	43.42	54.83	261.89
	CLINGCON	16.89	61.39	49.40	-	-	-	-	-
	CLINGCON/Opt	16.72	61.28	50.91	-	-	-	-	-
Estimator	EZCSP (B-Prolog)	0.27	0.73	1.64	25.64	77.38	303.75	-	-
	CLINGCON	25.10	4.87	82.78	-	-	-	-	-
	CLINGCON/Opt	-	-	-	-	-	-	-	-
	dReal	3.73	-	-	-	-	-	-	-

5.1 Car

The version of the car domain we used is the same that was adopted in Byrce *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 $[-8, 8]$.

EZCSP was run on the encoding described earlier. The CLINGCON encoding was obtained by a straightforward syntactic transformation. As illustrated by Tables 1 and 4, both CASP encodings solved all instances in approximately constant time – even in the non-linear case – demonstrating excellent scalability. The experimental evaluation yielded a number of other interesting results, discussed next. We begin from the outcomes of the fixed-time step experiments (Table 1), and later address the cumulative-time experiments.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Table 3. *Non-linear variant of generator with fixed time step. Results in seconds. Problem instances refer to number of tanks*

Encoding	Solver	1	2	3	4	5	6	7	8
Basic	EZCSP (GAMS/conopt)	1.24	3.25	18.49	-	147.94	-	-	*
	EZCSP (GAMS/couenne)	0.99	4.4	24.7	-	192.71	-	-	*
	EZCSP (GAMS/ipopt)	1.18	6.2	37.54	-	218.8	-	-	*
	EZCSP (GAMS/ipopth)	1.1	5.37	30.55	-	-	-	-	*
	EZCSP (GAMS/knitro)	0.82	3.62	20.2	-	166.64	-	-	*
	EZCSP (GAMS/lindo)	0.78	3.55	19.2	-	150.89	-	-	*
	EZCSP (GAMS/lindoglobal)	0.81	3.42	19.14	-	148.08	-	-	*
	EZCSP (GAMS/minos)	0.78	3.3	18.27	-	143.19	-	-	*
	EZCSP (GAMS/snopt)	0.8	3.41	18.78	-	148.43	-	-	*
Heuristic	EZCSP (GAMS/conopt)	0.72	1.62	0.68	1.05	87.95	256.59	238.93	*
	EZCSP (GAMS/couenne)	0.32	2.47	1.48	2.05	120.88	329.82	309.68	*
	EZCSP (GAMS/ipopt)	0.33	2.88	1.68	1.93	131.95	-	-	*
	EZCSP (GAMS/ipopth)	0.33	2.44	1.07	1.59	123.97	-	-	*
	EZCSP (GAMS/knitro)	0.31	1.84	0.79	1.13	126.80	449.79	566.50	*
	EZCSP (GAMS/lindo)	0.34	1.73	0.81	1.07	127.09	470.66	314.35	*
	EZCSP (GAMS/lindoglobal)	0.33	3.67	0.69	1.15	121.79	329.97	280.94	*
	EZCSP (GAMS/minos)	0.31	1.71	0.74	1.05	121.27	296.70	294.98	*
	EZCSP (GAMS/snopt)	0.29	1.64	0.66	0.98	104.46	-	-	*
Estimator	EZCSP (GAMS/conopt)	0.81	1.25	0.49	1.19	93.10	50.50	-	*
	EZCSP (GAMS/couenne)	0.33	1.84	0.76	2.65	132.22	73.10	-	*
	EZCSP (GAMS/ipopt)	0.38	1.98	0.79	2.20	145.62	-	-	*
	EZCSP (GAMS/ipopth)	0.30	1.73	0.65	1.70	134.29	-	-	*
	EZCSP (GAMS/knitro)	0.31	1.37	0.55	1.46	112.19	59.41	-	*
	EZCSP (GAMS/lindo)	0.30	1.22	0.50	1.12	97.75	53.66	-	*
	EZCSP (GAMS/lindoglobal)	0.28	1.14	0.51	1.12	96.20	52.71	-	*
	EZCSP (GAMS/minos)	0.34	1.16	0.52	1.16	94.26	51.12	-	*
	EZCSP (GAMS/snopt)	0.35	1.21	0.57	1.19	99.59	51.34	-	*
	dReal	8.18	-	-	-	-	-	-	-

Table 4. *Car domain with cumulative times. Results in seconds. Problem instances refer to max acceleration*

Domain	Solver	1	2	3	4	5	6	7	8
Car linear	EZCSP (B-Prolog)	1.01	0.98	1.04	0.99	0.91	0.85	0.88	0.83
	CLINGCON	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	UPMurphi	0.40	0.38	0.38	0.38	0.41	0.39	0.40	0.41
Car non-linear	EZCSP (GAMS/couenne)	2.59	1.78	1.90	2.50	1.91	2.00	1.84	3.96
	EZCSP (GAMS/knitro)	2.32	1.49	1.14	1.85	1.14	1.18	1.06	2.13
	UPMurphi	184.88	-	-	-	-	-	-	-

EZCSP *versus* CLINGCON. CLINGCON outperformed EZCSP in all experiments on the linear variant of *car*, although the small magnitude of both sets of run-times (negligible time for CLINGCON and about 0.30 s for EZCSP) prevents general claims.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Table 5. *Linear variant of generator with cumulative times. Results in seconds. Problem instances refer to number of refuel tanks*

Encoding	Solver	1	2	3	4	5	6	7	8
Basic	EZCSP (B-Prolog)	1.14	2.71	8.56	12.79	25.90	151.94	96.40	279.81
	CLINGCON	17.71	63.75	-	-	-	-	-	117.91
	CLINGCON/Opt	21.41	83.23	-	-	-	-	-	143.95
Heuristic	EZCSP (B-Prolog)	0.89	1.92	5.46	9.93	30.79	50.25	67.97	292.22
	CLINGCON	17.30	62.60	50.58	-	-	-	-	-
	CLINGCON/Opt	17.34	62.80	51.43	-	-	-	-	-
Estimator	EZCSP (B-Prolog)	0.83	1.55	3.19	26.27	82.32	318.98	-	-
	CLINGCON	31.44	6.51	103.79	-	-	-	-	-
	CLINGCON/Opt	22.83	71.26	-	-	-	-	-	110.70
	UPMurphi	2.02	12.75	91.80	-	-	-	-	-

Table 6. *Non-linear variant of generator with cumulative times. Results in seconds. Problem instances refer to number of refuel tanks*

Encoding	Solver	1	2	3	4	5	6	7	8
Basic	EZCSP (GAMS/conopt)	2.30	4.36	42.11	-	152.53	-	-	-
	EZCSP (GAMS/couenne)	2.79	10.08	81.56	-	387.30	-	-	-
	EZCSP (GAMS/ipopt)	2.00	7.58	80.99	-	226.92	-	-	-
	EZCSP (GAMS/ipopth)	1.84	6.77	67.04	-	-	-	-	-
	EZCSP (GAMS/knitro)	1.52	4.75	46.69	-	175.50	-	-	-
	EZCSP (GAMS/lindo)	1.46	4.57	44.07	-	157.69	-	-	-
	EZCSP (GAMS/lindoglobal)	1.46	4.50	43.62	-	155.14	-	-	-
	EZCSP (GAMS/minos)	1.49	4.50	46.98	-	153.10	-	-	-
	EZCSP (GAMS/snopt)	1.49	4.40	41.15	-	156.87	-	-	-
Heuristic	EZCSP (GAMS/conopt)	1.44	2.44	13.10	53.70	88.58	267.11	250.03	-
	EZCSP (GAMS/couenne)	1.39	5.72	21.07	77.78	244.31	-	-	-
	EZCSP (GAMS/ipopt)	0.97	3.77	24.43	78.26	136.53	-	-	-
	EZCSP (GAMS/ipopth)	0.93	3.30	20.31	78.17	126.66	-	-	-
	EZCSP (GAMS/knitro)	0.89	2.64	15.06	64.25	109.14	318.25	299.07	-
	EZCSP (GAMS/lindo)	0.84	2.45	13.20	54.29	89.98	265.14	254.92	-
	EZCSP (GAMS/lindoglobal)	0.91	2.74	14.57	58.92	99.72	278.55	266.06	-
	EZCSP (GAMS/minos)	0.87	2.36	14.55	56.95	93.78	285.23	259.65	-
	EZCSP (GAMS/snopt)	0.92	2.64	14.87	56.22	92.22	271.60	258.83	-
Estimator	EZCSP (GAMS/conopt)	0.88	1.89	12.66	54.95	96.47	55.28	-	-
	EZCSP (GAMS/couenne)	1.46	5.29	20.99	81.85	261.31	413.99	-	-
	EZCSP (GAMS/ipopt)	1.02	2.89	24.96	82.59	153.27	-	-	-
	EZCSP (GAMS/ipopth)	1.04	2.68	20.82	82.26	141.18	-	-	-
	EZCSP (GAMS/knitro)	0.96	2.17	15.82	72.33	118.26	69.32	-	-
	EZCSP (GAMS/lindo)	0.90	2.03	13.94	58.64	101.62	60.48	-	-
	EZCSP (GAMS/lindoglobal)	0.89	2.04	13.47	58.65	101.43	59.94	-	-
	EZCSP (GAMS/minos)	0.98	2.04	13.54	56.20	97.59	58.03	-	-
	EZCSP (GAMS/snopt)	0.85	1.91	13.15	55.46	95.75	57.28	-	-
	UPMurphi	—	-	-	-	-	-	-	-

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

GAMS solvers. Given the GAMS limitations in terms of sizes of the formulas that can be analyzed with the free license, only two GAMS solvers can be employed in this domain. Of those, KNITRO, which implements state-of-the-art interior-point and active-set methods for dealing with non-linear problems, is the best performer.

EZCSP *versus dReal.* EZCSP substantially outperformed dReal: EZCSP solved all non-linear instances in less than one second, while dReal only solved the two smallest instances, with times that peaked at 3 orders of magnitude larger than the EZCSP times.

Cumulative times (Table 4). As one might expect, given previous results, CLINGCON outperformed EZCSP on the cumulative-time experiments as well. The absolute times were still rather small – negligible for CLINGCON and between 0.83 s and 1.04 s for EZCSP. Once again, GAMS's KNITRO solver had the best performance among the GAMS solvers. Finally, the comparison with UPMurphi shows some interesting results. In the linear case, EZCSP is, in fact, about 2 to 3 times slower than UPMurphi, although the absolute times are small in both cases. On the other hand, EZCSP outperformed UPMurphi by a much larger margin in the non-linear case, with all instances solved in times between 1.78 s and 3.96 s, while UPMurphi only solved the first instance, and with a time of 184.88 s, i.e., nearly 2 orders of magnitude slower than EZCSP. Reasons for the performance of UPMurphi seem to be (i) the unoptimized implementation of the discretize-and-validate approach, and (ii) the fact that it performs blind search.

5.2 Generator

Our encoding models the transfer of liquid according to Torricelli's law:

$$v = \sqrt{2gh}.$$

It should be noted that this is different from the approach used in Bryce *et al.* (2015), where a simpler, but less physically accurate model was used. For a fair comparison with Bryce *et al.* (2015), the simpler model was used in reproducing the results for dReal. The instances were generated by increasing the number of refuel tanks from 1 to 8.

The complexity of this domain makes it amenable to studying various optimizations of the CASP encodings, mostly aimed at improving performance of the encoding with respect to the treatment of the must semantics.[14]

In the following, the CASP encoding presented throughout this paper is referred to as "Basic." The encoding referred to as "Heuristic" leverages the observation, based on straightforward considerations on this domain, that the *generate* process must start at timepoint 0. Thus, "Heuristic" extends "Basic" by a single heuristic stating that action *start*(*generate*) must occur immediately. The expectation is that "Heuristic" will outperform "Basic" in most cases. We are also interested

[14] It is not difficult to see, from our presentation of the CASP encoding, that the must semantics may significantly affect performance.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

in contrasting the effects of this domain-specific, encoding-level heuristic with those of the sophisticated, algorithm-level, and yet domain-independent, heuristics used in dReal.

The encoding labeled "Estimator" takes the observation about the *generate* process one step further, replacing the domain-specific heuristic with rules that, in some conditions, can be used to estimate the value of numerical fluents without calling the constraint solver. This is expected to enable earlier pruning of candidate plans directly within the ASP solver, and result in performance potentially comparable with that of "Heuristic." The encoding is based on the following idea. As we have seen, the conditions that trigger a process can be divided in Boolean and numerical. In the CASP encoding, the former are directly listed in the body of choice rule (3.15), while the latter are captured by numerical constraints, such as (3.16). Because these constraints are checked by the constraint solver, from an algorithmic perspective, a candidate plan that violates (3.16) is detected only after it has been fully computed by the ASP solver and passed to the constraint solver. This may obviously lead to unnecessary computations. A more economical approach seems to be that of checking as many numerical conditions as possible during the ASP computation. To accomplish that we maintain extra Boolean fluents that capture the value of numerical fluents, i.e., $has_val(f, v)$ states that numerical fluent f has value v. Keeping track of all changes to numerical fluents is obviously unfeasible on the ASP side, but it is indeed possible to keep track of direct assignments of values to them. For example, for every rule of the form

$$v_initial(f, I) = v \leftarrow \Gamma,$$

where I is as above and v is an integer constant,[15] we introduce a rule:

$$holds(has_val(f, v), I) \leftarrow \Gamma.$$

The value of such fluents is propagated by a rule:

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

The last element in the body of the rule enables blocking the propagation when needed. Specifically, for every rule $h(I) \leftarrow \Gamma$ of the original encoding whose head is a numerical constraint over the value of f (at step I) that is not in the form of the assignment considered above, we add to the encoding a rule:

$$ab(f, I) \leftarrow \Gamma.$$

It is worth noting that, compared to dReal and to "Heuristic," the "Estimator" encoding is both encoding-level and domain-independent. Furthermore, while dReal's heuristics are specific to the PDDL+ planning task, this approach is task-independent.

We also studied the performance of CLINGCON. To do so, we created translations of all of the three encodings, "Basic," "Heuristic," and "Estimator." These translations

[15] The approach can be extended to non-integer constants.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

are labeled "CLINGCON" in the tables. Note that, due to CLINGCON's limitations, the "CLINGCON/Opt" translations are applicable only to linear instances. Another CLINGCON variant that we considered, labeled "CLINGCON/Opt," is specifically aimed at leveraging additional features of CLINGCON. Differently from EZCSP, the language of CLINGCON allows for numerical constraints in both the head and body of rules. We attempted to take advantage of this and achieve a more compact encoding of the must semantics. We replaced rules (3.15) and (3.16) by a single rule, in which numerical constraints occur in the body and can directly trigger the process, as long as the Boolean conditions are satisfied. For example, (3.17) becomes

$$occurs(generate, I) \leftarrow holds(enabled, I), v_final(fuel_level, I) > 0. \tag{5.19}$$

In order to understand our interest in "CLINGCON/Opt," note that (5.19) is a regular rule, rather than a choice rule. With (3.15) and (3.16), the ASP solver will tend to generate multiple CSPs in an uninformed way, so that each can be checked by the constraint solver. With the adoption of (5.19), it is conceivable that the triggering of the process may be achieved earlier in the computation and improve performance, thanks to the tighter integration of the ASP and constraint solver in CLINGCON, whose algorithm is capable of interleaving the processes of solving the qualitative part of the problem and the numerical one.

Next, we discuss the experimental results. As for the *car* domain, we begin by addressing the outcomes of the fixed-time step experiments (Tables 2 and 3), and later analyze the cumulative-time experiments.

EZCSP *versus* CLINGCON. EZCSP outperformed CLINGCON in all experiments on the linear variant of *generator*, except for a single outlier (instance 8 of the "Basic" encoding). This is a remarkable result, given that in all previous analyses that the authors are aware of, the tighter coupling between ASP solver and constraint solver featured by CLINGCON led to better performance, see, e.g., Balduccini and Lierler (2013). The result is surprising in particular because the seemingly strong interdependency between the qualitative and numerical components of *generator* would have been expected to advantage tight coupled solvers.

*"*CLINGCON*" versus "*CLINGCON/*Opt."* Another surprising result comes from the comparison of the "CLINGCON" and "CLINGCON/Opt" translations. Recall that the latter is designed specifically to take advantage of CLINGCON's language features and solving algorithm. Yet, "CLINGCON/Opt" did not yield the performance improvements one might have expected. In the case of "Heuristic," "CLINGCON/Opt" and "CLINGCON" had similar performance; for the "Basic" encoding, "CLINGCON/Opt" was able to find a solution for an instance (instance 3) in which "CLINGCON" timed out, but at the same time the former failed to yield a solution for instance 8, which the latter was able to solve; finally, in the case of the "Estimator" encoding, "CLINGCON/Opt" timed out on every instance, yielding substantially worse performance than "CLINGCON."

GAMS solvers. GAMS's CONOPT solver, which implements a fast method for finding a first feasible solution, and thus can be efficient also on constraint problems

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

without an objective function like ours, exhibited better scalability than all other GAMS solvers.

Advanced encodings. Focusing on EZCSP, and on the GAMS CONOPT solver for the non-linear cases, there was no absolute winner among the "Basic," "Heuristic," and "Estimator" encodings. As one might expect, "Heuristic" yielded the best performance in many cases. On the other hand, "Estimator" was able to tie or beat "Heuristic" in a number of situations – most notably, non-linear instance 6. In the linear instances, "Basic" had worse performance than "Estimator" in most cases, but scaled better, solving instance 7 while the latter encoding timed out, suggesting a possible negative impact on scalability due to a somewhat larger encoding for "Estimator." It is also worth noting that "Estimator" surprisingly exhibited worse scalability in the linear case, where 6 instances were solved, than in the non-linear case, where 7 instances were solved, with the 8th giving a GAMS's license error rather than a timeout.

EZCSP *versus dReal.* Overall, EZCSP substantially outperformed *dReal*. Particularly interesting is the comparison between "Estimator" and dReal, since the former uses declarative, task-independent heuristics, while the latter employs algorithm-level, task-dependent ones. The execution times for EZCSP for this case (Tables 2 and 3) ranged between 0.27 s and 303.75 s for the linear variant, with 6 instances solved, and between 0.72 s and 256.59 s for the non-linear one (with GAMS's CONOPT solver), with 7 instances solved. On the other hand, *dReal* was only able to solve the first instance in both cases (3.73 s for the linear one and 8.18 s for the non-linear one).

Cumulative times (Tables 5 and 6). If we contrast the results based on solvers used, CLINGCON beat EZCSP in absolute performance on the largest instance (110.70 s versus 279.81 s in instance 8), but lost when overall scalability is considered: the EZCSP versions of "Basic" and "Estimator" solve all instances while the CLINGCON translations solve 3 instances at best. "CLINGCON/Opt" yielded mixed performance results: while the "CLINGCON" and "CLINGCON/Opt" translations solved the same number of instances overall, the latter often had worse performance – up to one order of magnitude for instance 2 when using the "Estimator" encoding, but typically 20%–30% worse. On the other hand, the "CLINGCON/Opt" translation of the "Estimator" encoding was able to solve instance 8. Interestingly, both CLINGCON translations of the "Heuristic" encoding timed out, while the CLINGCON translations of the "Basic" encoding found solutions. This is surprising given that "Heuristic," when compared to "Basic," introduces heuristic knowledge that simplifies the problem.

Looking at the non-linear variant of the domain, we see that, once again, GAMS's CONOPT solver was the one with best performance among the GAMS solvers. Focusing on EZCSP (and GAMS's CONOPT), "Basic" was slightly better than "Heuristic" and "Estimator" in some cases (e.g., linear instance 8), but overall lost to both – especially in the non-linear instances, where "Basic" solves 3 and 2 fewer instances, respectively. Remarkably, "Estimator" featured marginally better performance than "Heuristic" on most easy instances (instances 1–3), but did not scale equally well. Finally, UPMurphi had substantially worse performance than EZCSP in all cases: it solved only 3 linear instances (versus 8 for EZCSP) and no

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

non-linear instances (versus 7 for EZCSP[16]). The speedup yielded by EZCSP reached about one order of magnitude before UPMurphi began to time out.

5.3 Summary

Considering the entire set of experiments, we can draw a number of important conclusions. First of all, the experiments demonstrate that our CASP-based solution substantially outperforms both dReal and UPMurphi, showing better scalability. EZCSP is also more scalable than CLINGCON on challenging problems from the *generator* domain. This is a remarkable result, which goes against previous comparisons (see, e.g., Balduccini and Lierler 2013) of the performance of tightly coupled hybrid solving algorithms versus their loosely coupled counterparts. Note that such comparisons were conducted on sets of benchmarks different from the ones considered here. This indicates that PDDL+ planning problems may be a new and useful class of valuable benchmarks for the CASP community.

Looking specifically at encoding variants and the corresponding techniques, the "Heuristic" variant of the encoding was overall the best, while "Estimator" worked well in a number of cases, but did not equal its performance. These results corroborate the considerations that led to creation of these variants. Also, "CLINGCON/Opt" failed to yield consistent performance improvements in spite of its leveraging the more advanced features of the CLINGCON language and solver, which are not available in EZCSP.

Overall, we believe the empirical results demonstrate the promise of our approach.

6 Extended architecture

As we mentioned, validation through VAL always succeeds in the previous experiments. In this section, we discuss how the architecture from Section 3 can be extended to handle more efficiently cases in which the validation does not succeed. Specifically, we present a variant of the architecture from Figure 5 aimed at making better use of the information provided by VAL in case of a failed validation.

The extended architecture is shown in Figure 8. While the figure refers to EZCSP, the approach is not difficult to extend to CLINGCON and other CASP solvers.

Our approach leverages the fact that, when VAL finds a plan not to be valid, it returns information about which invariants were violated and in which timepoint intervals. Consider a variant of instance 1 of the non-linear generator from Section 5, modified so that the capacity and initial level of the generator's tank are 100 units. A possible plan for this instance is

$$0.000 : \mathit{generate}\ [100.000]$$
$$12.500 : \mathit{refuel}(\mathit{tank}1)\ [12.500]$$

[16] It should be noted that the experiments on cumulative times do not result in GAMS licensing errors, because the corresponding instances time out before reaching the point where the error would be generated.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

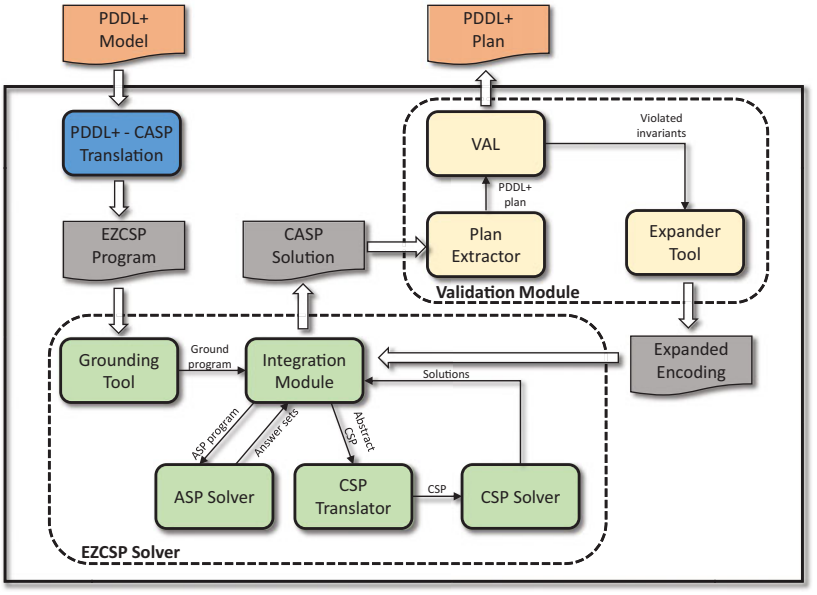


Fig. 8. Extended EZCSP solver architecture.

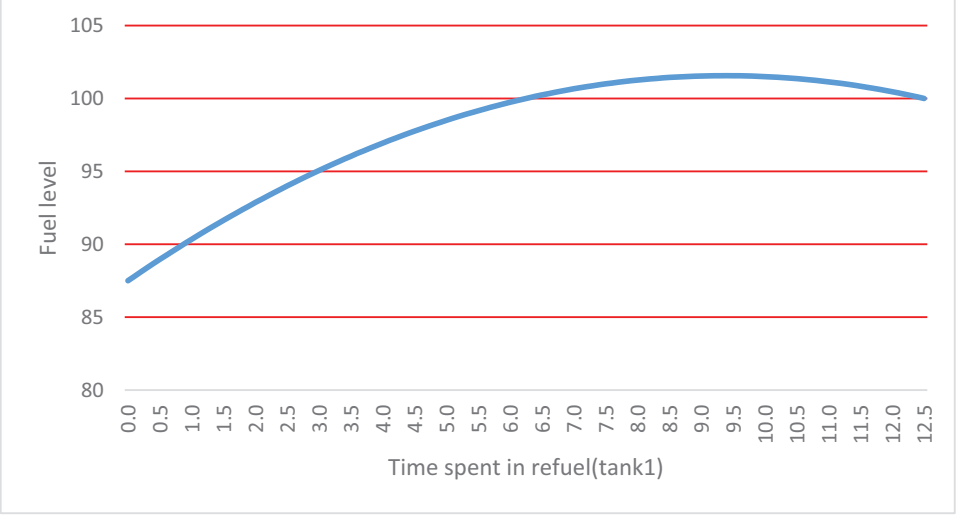


Fig. 9. Fuel level during refuel(tank1). Capacity 100.00. Spill @ 6.25 s from the start of the action.

When validating this plan, VAL detects that the invariant

```
(< (fuelLevel ?g) (capacity ?g))
```

is violated during the execution of $refuel(tank1)$ – specifically, starting at 6.25 time units into the execution of the action, corresponding to timepoint range $[18.75, 25]$. See Figure 9 for an illustration. Such feedback is leveraged by the *Expander Tool* from Figure 8. The intuition behind the process is to select suitable timepoints in

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

the offending interval, and introduce in the encoding (i) new numerical variables that capture the value of the relevant numerical fluents at those timepoints, and (ii) corresponding numerical constraints that enforce the invariants on those variables. The idea is made precise by Algorithms 1 and 2. Function EXPAND (Algorithm 1) takes in input the CASP encoding and a set of pairs $\langle inv, range \rangle$, where *inv* is a violated invariant and *range* is the timepoint range in which it is violated. For each such pair, the algorithm selects a set of timepoints within the range (step 4) via user-supplied function SELECT_TIMEPOINTS and stores them in Δ. These are the timepoints on which the invariant will be enforced by means of additional constraints. In simple cases, SELECT_TIMEPOINTS may select the timepoints so that they are uniformly distributed within the range, but more complex options are possible. Next (step 5), the algorithm identifies the time step, *s*, at which the violation of the invariant occurred. For simplicity of presentation, we assume that a single time step corresponds to each range. (If the range extends over multiple time steps, one can always pre-process the output of VAL, splitting each violation into violations whose range corresponds to a single step.) The loop that begins at step 6 iterates over every timepoint δ from Δ and (step 7) considers each numerical fluent, *n*, that occurs in the invariant. For each of them, EXPAND_FLUENT (Algorithm 2) is called. The function, which will be described in more detail next, expands the encoding with constraints that calculate the value of *n* at timepoint δ. Next, steps 10–13 insert in the encoding a final constraint that instantiates the offending invariant for timepoint δ. At the core of the process is the instantiation of the constraint from the head of the rule of the form (3.8) for *inv*, which occurs at step 12. In the first part of the process, every occurrence of ASP variable *I* is replaced by the time step, *s*, identified earlier, at which the invariant was violated. Next, for every numerical fluent *n*, the occurrences of $v_final(n, s)$ are replaced by $v_final_{\delta}(n, s)$. This ensures that the constraint considers the values of the fluents at timepoint δ, as calculated by the constraints added by EXPAND_FLUENT, as opposed to the end of the time step.

Function EXPAND_FLUENT is given in input the program Π, a numerical fluent *n*, a timepoint δ and time step *s*. First of all, the function expands the encoding by creating a rule of the form (3.10) in which $v_final(\cdot, \cdot)$ and $v(contrib(\cdot), \cdot)$ are replaced by new variables $v_final_{\delta}(\cdot, \cdot)$ and $v_{\delta}(contrib(\cdot), \cdot)$. These variables capture the value of *n* at timepoint δ and the contributions to such value. Next, (steps 3-7) variants of the rules of the form (3.9) are introduced. The new rules calculate the contributions to the value of *n* at timepoint δ for each durative action and process, *d*. Note that, in rule (3.9), such calculation depends on the duration of the action. However, in calculating the contribution of *d* from its start and up to timepoint δ, the time components of the calculation must be scaled accordingly. This is accomplished by OFFSET (Algorithm 2), which normalizes, w.r.t. $[0, 1]$, the position of δ within the interval of execution of *d*, and by step 6 of EXPAND_FLUENT, which scales the time component accordingly. Recall that e^i_n from (3.9) represents the value of the contribution to *n* from a particular source. Thus, at the core of step 6 is the construction of a new expression, e', which represents the contribution up to timepoint δ. Such expression is obtained by instantiating *I* to the time step at

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Algorithm 1 Expansion step

```
1: function EXPAND(Π, V)
   input: Π: CASP encoding; V: set of pairs ⟨inv, range⟩
2:     Π′ := ∅
3:     for ⟨inv, range⟩ ∈ V do
4:         Δ := SELECT_TIMEPOINTS(range)
5:         s := time step at which inv is violated
6:         for δ ∈ Δ do
7:             for n ∈ inv do
8:                 Π′ := Π′ ∪ EXPAND_FLUENT(Π, n, s, δ)
9:             end for
10:            d := durative action or process for which inv is specified
11:            γ^f := constraint from the head of the rule of form (3.8) for inv
12:            γ^f(s, δ) := replace in γ^f : { 1. I by s
                                             { 2. v_final(n, s) by v_final_δ(n, s)  for every numerical fluent n
13:            Π′ := Π′ ∪ {γ^f(s, δ) ← holds(inprogr(d), s).}
14:        end for
15:    end for
16:    return Π′
17: end function
```

which the invariant was violated and by scaling down, by a factor δ_d, the duration of the interval under consideration.

Next, we illustrate the algorithms with an example, which we carry out using the language of EZCSP. Continuing with the output of VAL described earlier, EXPAND begins by selecting timepoints at which the invariant should be further checked. For sake of illustration, let us select 3 uniformly distributed timepoints, 18.75, 21.875, and 25. The expansion of the encoding for the negative contribution and timepoint 18.75 is (below, $\iota(t)$ is the formula that calculates fuel transferred for a duration of time t; the actual calculation is omitted for space considerations):

$$required(v_{18.75}(contrib(fuel_level, incr, refuel(TK)), 2) = \iota(\tfrac{18.75-12.5}{25-12.5} * (tend(2) - tstart(2)))) \leftarrow holds(inprog(refuel(TK)), 2).$$

$$required(v_final_{18.75}(fuel_level, 2) = v_initial(fuel_level, 2) + v_{18.75}(contrib(fuel_level, incr), 2) - v_{18.75}(contrib(fuel_level, decr), 2)) \leftarrow step(I).$$

$$required(v_final_{18.75}(fuel_level, 2) \leqslant TC) \leftarrow tankcap(TC), holds(inprog(refuel(TK)), 2).$$

The expansion for the positive contributions to the fuel level and for timepoints 21.875 and 25 is similar. Once the encoding has been expanded, EZCSP is executed again. The added constraints ensure that any solution that is returned does not violate the invariants at those timepoints. The process is iterated as needed until the validation succeeds. In our example, for the expanded encoding described above, the planner returns the plan:

$$0.000 : generate\ [100.000]$$
$$14.063 : refuel(tank1)\ [12.500]$$

This plan is successfully validated by VAL, and the search terminates.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Algorithm 2 Expansion of a numerical fluent

function EXPAND_FLUENT(Π, n, s, δ)
input: Π: CASP encoding; n: numerical fluent; s: time step; δ: timepoint
  $\Pi' := \{v_final_\delta(n,s) = v_initial(n,s) + \sum_{d \in D} v_\delta(contrib(n,d),s).\}$
  for durative action or process d **do**
    for every rule of Π of the form $v(contrib(n,d),I) = e_n^i \leftarrow holds(inprogr(d),I)$ **do**
      $\delta_d := \text{OFFSET}(\delta, d)$
      $e' :=$ replace in e_n^i: $\begin{cases} 1. & I & \text{by} & s \\ 2. & tend(s) - tstart(s) & \text{by} & \delta_d \cdot (tend(s) - tstart(s)) \end{cases}$
      $\Pi' := \Pi' \cup \{v_\delta(contrib(n,d),I) = e' \leftarrow holds(inprogr(d),s).\}$
    end for
  end for
  return Π'
end function

function OFFSET(δ, d)
input: δ: timepoint; d: durative action or process
  $\langle s, e \rangle :=$ start, end timepoints of d in the plan
  if $\delta > e$ **then return** 1
  if $\delta < s$ **then return** 0
  return $\frac{\delta - s}{e - s}$
end function

The approach has been implemented in prototypical form. When applied to the problem above, the prototype correctly detects that the original plan is not validated by VAL. Then, using VAL's output, the expanded encoding is generated as above, and EZCSP is used to find a new solution. Using the "Heuristic" encoding, EZCSP, and GAMS's CONOPT solver, the computation of the first plan takes 0.37 s and that of the second plan takes 0.40 s, for a total of 0.77 s. The time taken for the validation by VAL is negligible. For "Basic" and "Estimator," the total times are 1.08 s and 0.84 s, respectively.

7 Related work

In this section, we analyze related approaches and solvers. We start with PDDL+, then we move to CASP. Last, we relate our approach to research on planning in hybrid domains based on action languages.

PDDL+ algorithms and solvers. Various techniques and tools have been proposed to deal with hybrid domains (Penberthy and Weld 1994; McDermott 2003; Shin and Davis 2005; Li and Williams 2008; Coles *et al.* 2012). Nevertheless, none of these approaches are able to handle the full set of PDDL+ features, namely, non-linear domains with processes and events.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

More recent works include Bryce *et al.* (2015), which presents an approach based on SMT for handling hybrid domains. However, dReach does not use PDDL+, and cannot handle exogenous events.

From the model checking and control communities, a number of works based on timed and hybrid automata have been proposed to handle hybrid systems. Some examples include Tabuada *et al.* (2002), Maly *et al.* (2013), Cavada *et al.* (2014), Henzinger and Otop (2014), Liu and Ozay (2014), Cimatti *et al.* (2015), and Bae *et al.* (2016), and sampling-based planners (Karaman *et al.* 2011; Lahijanian *et al.* 2014). Similarly, *falsification* of hybrid systems tries to guide the search towards the error states, that can be easily cast as a planning problem (Cimatti *et al.* 1997; Plaku *et al.* 2013). However, while all these works aim to address hybrid systems, they cannot be used to handle PDDL+ models. Some recent works (Bogomolov *et al.* 2014, 2015) are trying to define a formal translation between PDDL+ and standard hybrid automata, but so far only an over-approximation has been defined, that allows the use of those tools only for proving plan non-existence.

UPMurphi (Della Penna *et al.* 2009, 2012) is the only tool able to handle the full set of PDDL+ features, although it is very limited in scalability as it performs blind search.

PDDL+ has been used to model a number of planning applications (Della Penna *et al.* 2010; Fox *et al.* 2011; Vallati *et al.* 2016) but then specific domain-dependent heuristics or tools have been used to find plans (Piacentini *et al.* 2016; Fox *et al.* 2012).

The approach proposed in this paper is similar to the encoding used in TM-LPSAT (Shin and Davis 2005). However, TM-LPSAT assumes linear continuous change, given it uses a linear solver to manage the continuous constraints, while we are solving problems with non-linear dynamics. TM-LPSAT exploits the linearity by checking conditions only at the end-points of intervals of continuous change. Moreover, as reported by Shin and Davies, the experiments showed that TM-LPSAT was not performant and finally the code is not available; hence, TM-LPSAT remains a largely theoretical (though valuable) contribution.

Finally, some very recent works on PDDL+ planning have been proposed (Cashmore *et al.* 2016; Piotrowski *et al.* 2016), but a proper comparison with them will be addressed in future work.[17]

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

ACSOLVER (Mellarkod *et al.* 2008) 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.

[17] At the time of submission, these papers were not published yet, and the corresponding planners not released.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

CLINGON (Ostrowski and Schaub 2012) 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, unlike EZCSP, it allows for numerical constraints both in the head of rules and in their bodies, and is characterized by a tighter coupling of ASP and CSP solvers.

A high level view of the languages and solving techniques employed by these solvers can be found in Lierler (2014). There, by relying on the framework of *abstract solvers*, i.e., a graph-based representation of solving algorithms, similarities and differences among these solvers are formally stated by means of comparison of the related graphs.

Hybrid domains and action languages. Action language *H* was introduced in Chintabathina *et al.* (2005) as an extension to previous well-known action languages for modeling hybrid domains. Then, in Chintabathina (2013), *H* has been used specifically to model planning and scheduling tasks in hybrid domains, and reasoning is done via CASP language and solver EZCSP as in our paper. However, that approach suffers from some shortcomings, which our work overcomes, e.g., it (i) does not take into account PDDL+ as target language, despite it being the standard language for planning in hybrid domains, (ii) does not consider additive fluents, non-linear numerical constraints, and the triggering of processes and events via the must semantics, and (iii) does not include an experimental evaluation of the approach. Another action language *ADP* (Baral *et al.* 2002) was introduced earlier to allow for the specification of, e.g., actions with duration and continuous effects.

Other lines of research that have dealt with extensions involving primitives for dealing with continuous change, processes, and (macro-)events include the Event Calculus in, e.g., Evans (1990), Shanahan (1990), Miller and Shanahan (1996), and Cervesato and Montanari (2000).

8 Conclusions

PDDL+ and CASP languages extend PDDL and ASP to reason with mixed discrete-continuous dynamics. PDDL+ is the standard language for the automated planning community, with a number of interesting domains being represented as PDDL+ models. In this paper, we have presented a new approach for solving PDDL+ problems by means of an encoding into CASP problems, and extension to the EZCSP solving architecture for planning in hybrid domains. Our solution can deal with both linear and non-linear variants of the domains. An experimental analysis, performed on well-known PDDL+ domains, involving some variants of our approach, other CASP solvers and PDDL+ planners on two reasoning tasks, showed the viability of our approach.

References

BAE, K., ÖLVECZKY, P. C., KONG, S., GAO, S. AND CLARKE, E. M. 2016. SMT-based analysis of virtually synchronous distributed hybrid systems. In *Proc. of 19th International Conference*

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

on Hybrid Systems: Computation and Control (HSCC 2016), A. Abate and G. E. Fainekos, Eds. ACM, 145–154.

Balduccini, M. 2009. Representing constraint satisfaction problems in answer set programming. In *Proc. of ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2009)*.

Balduccini, M. and Gelfond, M. 2003. Logic programs with consistency-restoring rules. In *Proc. of International Symposium on Logical Formalization of Commonsense Reasoning*, P. Doherty, J. McCarthy and M.-A. Williams, Eds., AAAI 2003 Spring Symposium Series. AAAI Press, 9–18.

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, M. and Lierler, Y. 2013. Integration schemas for constraint answer set programming: A case study. *Theory and Practice of Logic Programming 13,* 4–5, On-line Supplement.

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.

Baral, C., Son, T. C. and Tuan, L. 2002. A transition function based characterization of actions with delayed and continuous effects. In *Proc. of 8th International Conference on Principles and Knowledge Representation and Reasoning (KR 2002)*, D. Fensel, F. Giunchiglia, D. L. McGuinness and M. Williams, Eds. Morgan Kaufmann, 291–302.

Baselice, S., Bonatti, P. A. and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proc. of 21st International Conference on Logic Programming (ICLP 2005)*, M. Gabbrielli and G. Gupta, Eds., Lecture Notes in Computer Science, vol. 3668, 52–66.

Bogomolov, S., Magazzeni, D., Minopoli, S. and Wehrle, M. 2015. PDDL+ planning with hybrid automata: Foundations of translating must behavior. In *Proc. of 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, R. I. Brafman, C. Domshlak, P. Haslum and S. Zilberstein, Eds. AAAI Press, 42–46.

Bogomolov, S., Magazzeni, D., Podelski, A. and Wehrle, M. 2014. Planning as model checking in hybrid domains. In *Proc. of 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, C. E. Brodley and P. Stone, Eds. AAAI Press, 2228–2234.

Bryce, D., Gao, S., Musliner, D. J. and Goldman, R. P. 2015. SMT-based nonlinear PDDL+ planning. In *Proc. of 29h AAAI Conference on Artificial Intelligence (ICAPS 2015)*, B. Bonet and S. Koenig, Eds. AAAI Press, 3247–3253.

Cashmore, M., Fox, M., Long, D. and Magazzeni, D. 2016. A compilation of the full PDDL+ language into SMT. In *Proc. of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, A. J. Coles, A. Coles, S. Edelkamp, D. Magazzeni and S. Sanner, Eds. AAAI Press, 79–87.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M. and Tonetta, S. 2014. The nuXmv symbolic model checker. In *Proc. of 26th International Conference on Computer Aided Verification (CAV 2104)*, A. Biere and R. Bloem, Eds., Lecture Notes in Computer Science, vol. 8559. Springer, 334–342.

Cervesato, I. and Montanari, A. 2000. A calculus of macro-events: Progress report. In *Proc. of 7th International Workshop on Temporal Representation and Reasoning (TIME 2000)*. IEEE Computer Society, 47–58.

Chintabathina, S. 2013. Planning and scheduling in hybrid domains using answer set programming. URL: http://arxiv.org/pdf/1301.1389v1 [Accessed on 02/05/2016].

Chintabathina, S., Gelfond, M. and Watson, R. 2005. Modeling hybrid domains using process description language. In *Proc. of ASP '05: Answer Set Programming: Advances in Theory and Implementation*, 303–317.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Cimatti, A., Giunchiglia, E., Giunchiglia, F. and Traverso, P. 1997. Planning via model checking: A decision procedure for AR. In *Recent Advances in AI Planning*. Springer, 130–142.

Cimatti, A., Griggio, A., Mover, S. and Tonetta, S. 2015. HyComp: An SMT-based model checker for hybrid systems. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (ETAPS 2015)*, 52–67.

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, G., Intrigila, B., Magazzeni, D. and Mercorio, F. 2010. A PDDL+ benchmark problem: The batch chemical plant. In *Proc. of 20th International Conference on Automated Planning and Scheduling (ICAPS 2010)*, R. I. Brafman, H. Geffner, J. Hoffmann and H. A. Kautz, Eds. AAAI Press, 222–225.

Della Penna, G., Magazzeni, D. and Mercorio, F. 2012. A universal planning system for hybrid domains. *Applied Intelligence 36,* 4, 932–959.

Della Penna, G., Magazzeni, D., Mercorio, F. and Intrigila, B. 2009. UPMurphi: A tool for universal planning on PDDL+ problems. In *Proc. of 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, A. Gerevini, A. E. Howe, A. Cesta and I. Refanidis, Eds. AAAI Press.

Evans, C. 1990. The macro-event calculus: Representing temporal granularity. In *Proc. of Pacific Rim International Conference on Artificial Intelligence (PRICAI 1990)*. IOS Press.

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

Fox, M., Long, D. and Magazzeni, D. 2011. Automatic construction of efficient multiple battery usage policies. In *Proc. of 21st International Conference on Automated Planning and Scheduling (ICAPS 2011)*, F. Bacchus, C. Domshlak, S. Edelkamp and M. Helmert, Eds. AAAI Press.

Fox, M., Long, D. and Magazzeni, D. 2012. Plan-based policies for efficient multiple battery load management. *Journal of Artificial Intelligence Research 44*, 335–382.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, R. A. Kowalski and K. A. Bowen, Eds., 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Gelfond, M. and Lifschitz, V. 1993. Representing action and change by logic programs. *Journal of Logic Programming 17,* 2–4, 301–321.

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

Henzinger, T. A. and Otop, J. 2014. Model measuring for hybrid systems. In *Proc. of 17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week) (HSCC 2014)*, M. Fränzle and J. Lygeros, Eds. ACM, 213–222.

Howey, R., Long, D. and Fox, M. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proc. of 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*. IEEE Computer Society, 294–301.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *Proc. of 14th ACM Symposium on Principles of Programming Languages (POPL 1987)*. ACM Press, 111–119.

Karaman, S., Walter, M. R., Perez, A., Frazzoli, E. and Teller, S. J. 2011. Anytime motion planning using the RRT. In *IEEE International Conference on Robotics and Automation (ICRA 2011)*. IEEE, 1478–1483.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Katriel, I. and van Hoeve, W.-J. 2006. Global constraints. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek and T. Walsh, Eds. Foundations of Artificial Intelligence, Chapter 6. Elsevier, 169–208.

Kautz, H. A. and Selman, B. 1992. Planning as satisfiability. In *Proc. of 10th European Conference on Artificial Intelligence (ECAI 1992)*, B. Neumann, Ed. 359–363.

Lahijanian, M., Kavraki, L. E. and Vardi, M. Y. 2014. A sampling-based strategy planner for nondeterministic hybrid systems. In *Proc. of IEEE International Conference on Robotics and Automation (ICRA 2014)*. IEEE, 3005–3012.

Li, H. X. and Williams, B. C. 2008. Generative planning for hybrid systems based on flow tubes. In *Proc. of 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, J. Rintanen, B. Nebel, J. C. Beck and E. A. Hansen, Eds. AAAI Press, 206–213.

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

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

Liu, J. and Ozay, N. 2014. Abstraction, discretization, and robustness in temporal logic control of dynamical systems. In *Proc. of 17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week) (HSCC 2014)*, M. Fränzle and J. Lygeros, Eds. ACM, 293–302.

Maly, M. R., Lahijanian, M., Kavraki, L. E., Kress-Gazit, H. and Vardi, M. Y. 2013. Iterative temporal motion planning for hybrid systems in partially unknown environments. In *Proc. of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, C. Belta and F. Ivancic, Eds. ACM, 353–362.

Marek, V. W. and Truszczynski, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. Springer Verlag, Berlin, 375–398.

Marriott, K., Stuckey, P. J. and Wallace, M. 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, Chapter 12. Constraint Logic Programming, 409–452.

McDermott, D. V. 2003. Reasoning about autonomous processes in an estimated-regression planner. In *Proc. of 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*, E. Giunchiglia, N. Muscettola and D. S. Nau, Eds. AAAI Press, 143–152.

Mellarkod, V. S., Gelfond, M. and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1–4, 251–287.

Miller, R. and Shanahan, M. 1996. Reasoning about discontinuities in the event calculus. In *Proc. of 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 1996)*, L. C. Aiello, J. Doyle and S. C. Shapiro, Eds. Morgan Kaufmann, 63–74.

Nau, D., Ghallab, M. and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Morgan Kaufmann.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3–4, 241–27.

Ostrowski, M. and Schaub, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming 12,* 4–5, 485–503.

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

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press

Piacentini, C., Magazzeni, D., Long, D., Fox, M. and Dent, C. 2016. Solving realistic unit commitment problems using temporal planning: Challenges and solutions. In *Proc. of 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, A. J. Coles, A. Coles, S. Edelkamp, D. Magazzeni and S. Sanner, Eds. AAAI Press.

Piotrowski, W., Fox, M., Long, D., Magazzeni, D. and Mercorio, F. 2016. Heuristic planning for PDDL+ domains. In *Proc. of 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, S. Kambhampati, Ed. IJCAI/AAAI Press, 3213–3219.

Plaku, E., Kavraki, L. E. and Vardi, M. Y. 2013. Falsification of LTL safety properties in hybrid systems. *Software and Tools for Technology Transfer 15,* 4, 305–320.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Shanahan, M. 1990. Representing continuous change in the event calculus. In *Proc. of 9th European Conference on Artificial Intelligence (ECAI 1990)*. 598–603.

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

Smith, B. M. 2006. Modelling. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek and T. Walsh, Eds. Foundations of Artificial Intelligence, Chapter 11. Elsevier, 377–406.

Tabuada, P., Pappas, G. J. and Lima, P. U. 2002. Composing abstractions of hybrid systems. In *Proc. of 5th International Conference on Hybrid Systems: Computation and Control (HSCC 2002)*, C. Tomlin and M. R. Greenstreet, Eds., Lecture Notes in Computer Science, Vol. 2289. Springer, 436–450.

Vallati, M., Magazzeni, D., Schutter, B. D., Chrpa, L. and McCluskey, T. L. 2016. Efficient macroscopic urban traffic models for reducing congestion: A PDDL+ planning approach. In *Proc. of 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 3188–3194.

https://doi.org/10.1017/S1471068417000187 Published online by Cambridge University Press