In ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)

Translating PDDL2.2. into a Constraint-based Variable/Value Language

Sara Bernardini
University of Trento
Via Sommarive 14, 38055 Trento, Italy
sara.bernardini@unitn.it

David E. Smith
NASA Ames Research Center
Moffet Field, CA 94035–1000
david.smith@nasa.gov

Abstract

We develop a translation from PDDL2.2 into NDDL, the language used by NASA's EUROPA2 planning system. Starting from a PDDL2.2 description of the world based on propositions and operators, we generate an NDDL model that features timelines, activities and compatibilities. The relevance of the translation is twofold. From a theoretical point of view, the translation is interesting since it transforms an action-centered propositional representation into a multi-valued variable representation based on temporal constraints. From a practical point of view, the translation allows testing of application-oriented planning systems on standard benchmarks developed for the Internal Planning Competition (IPC).

Introduction

We present a technique for performing an *automatic translation* from the PDDL2.2 language (Edelkamp & Hoffmann 2004) into a *variable/value constraint-based* language called NDDL (New Domain Definition Language). NDDL is the modelling language of the EUROPA2 planning system (Frank & Jónsson 2003), currently being employed for several NASA mission applications including the Mars Exploration Rover mission (MER) (Bresina *et al.* 2005). Being designed and used for representing complex real-world problems, NDDL is a powerful language that allows one to describe large domains with complex temporal constraints, resources, exogenous events and relative ordering contraints between activities. However, because the language and assumptions are so different, EUROPA2 has not been tested on the many benchmark domains developed for the IPC.

PDDL2.2 and NDDL are expressions of two alternative planning paradigms. EUROPA2, as well as a number of planners used for real-world applications like ASPEN (Chien *et al.* 2000), IxTeT (Ghallab & Laruelle 1994) and OMPT (Fratini, Pecora, & Cesta 2008), implements *constraint-based temporal planning* (Smith, Frank, & Jónsson 2000; Frank & Jónsson 2003), while most planners that use PDDL2.2 are basically *classical planners* extended to treat time and numeric functions. EUROPA2 departs from traditional planning in many respects. For a given domain, the system under consideration is decomposed into a set of primitive *components* that perform mutually exclusive *activities* over time intervals. The transition of a component from an activity to another activity and its synchronization with the behaviour of other components are regulated by a set of temporal constraints called *compatibilities*. Thus, while PDDL represents the world in terms of propositions and operators, NDDL uses multi-valued state variables whose evolution over time and reciprocal relationships are expressed through interval-based and point-based temporal relations.

Most of the translation we develop is not specific to EUROPA2 – it is a general approach to transforming an action-centered propositional representation into a variable/value constraint-based representation. More precisely, the invariant synthesis and the timeline generation are general techniques that can be used to obtain temporal multi-valued state variables from a PDDL instance, while the compatibility generation is a method to extract temporal constraints between state variables from the PDDL specification of durative operators. The translation can be applied, with slight modifications, to a number of constraint-based temporal planners, such as IxTeT (Ghallab & Laruelle 1994) and OMPS (Fratini, Pecora, & Cesta 2008). It will allow us to test these application-oriented planners on standard benchmarks developed for the IPC. Besides, the translation promotes the recognition of the core differences between classical and constraint-based temporal planning and helps to identify how to map successful techniques and features of one paradigm into the other. That is important since the cross-fertilization of ideas between these two planning communities has been limited so far.

Our translation builds on the translation presented in (Helmert 2006b) that transforms a sub-set of the PDDL2.2 language into the *multi-valued planning task* formalism (Helmert 2006a). This translation is limited to non-temporal and non-numeric PDDL domains. In contrast, our translation tackles temporal and numeric domains. Translating such tasks is more complex than handling tasks with instantaneous discrete actions, because interference between concurrent operators complicates the identification of state variables and the synthesis of compatibilities.

In order to explain the translation, it is necessary to have some understanding of the NDDL language. We present the essentials in the next section. We then give an overview of the entire translation process and a detailed description of the main steps. We conclude by discussing current status and future work.

NDDL: EUROPA2's Modelling Language

In NDDL, a *planning instance* I is a pair: $\mathrm{I} = (\mathcal{D}, \mathcal{P})$, where $\mathcal{D}$ is the *planning domain* and $\mathcal{P}$ is the *planning problem*.

A ***planning domain*** $\mathcal{D}$ is represented by:

- A set of ***timelines***: $\mathrm{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$, which are essentially multi-valued state-variables capturing the evolution of a component or quantity over time.
- A set of mutually exclusive ***activities*** associated with each timeline $\mathcal{T}_i$: $\mathcal{A}ct[\mathcal{T}_i] = \{a_1(\vec{x}_1, \delta_1), \ldots, a_n(\vec{x}_n, \delta_n)\}$, where $\vec{x}_j$ is the vector of the parameters of the activity a_j and $\delta_j = [\delta_j^{min}, \delta_j^{max}]$ is a mathematical interval in $\mathbb{N}$ representing the *duration* of a_j. The parameters $start$ and end in $\vec{x}_j$ access the start and end time of a_j.
- An ***evolution rule*** $\mathcal{R}[a]$ for each activity $a \in \mathcal{D}$. The rule $\mathcal{R}[a]$ is a conjunction of *compatibilities*: $\mathcal{R}[a] = \mathcal{K}_1[a] \wedge \ldots \wedge \mathcal{K}_n[a]$. A ***compatibility*** $\mathcal{K}[a]$ describes the relationship between the *master* activity $a(\vec{x}, \delta)$ and a number of *slave* activities $\mathcal{S}l[a, \mathcal{K}] = \{a_1(\vec{y}_1, \delta_1), \ldots, a_k(\vec{y}_k, \delta_k)\}$. The slaves can be associated with the same timeline of a or with different timelines. Formally, a compatibility assumes the following form:

$$\mathcal{K}[a] : \bigwedge_{i=0}^{n} g_i \Rightarrow \bigwedge_{i: a_i \in \mathcal{S}l[a,\mathcal{K}]} (t_i \wedge \bigwedge_{j=0}^{k} p_{ij})$$

  which is composed of:

  – ***Guard constraints***: a set of constraints g_i specifying when the compatibility applies. A *guard constraint* g is an atomic formula $\gamma = t$, where γ is a variable called *guard* and t is a constant. We assume that $g_0 = \top$.

  – ***Temporal constraints***: a set of constraints t_i specifying the temporal relationships between the master activity and its slaves. Given the master a and a slave a_i, a temporal constraint t_i assumes one of two possible forms:

  ∗ a `temporal_relation` a_i,
  where `temporal_relation` is an *interval-based* temporal predicate (`meets`, `met_by`, `contains`, `contained_by`, ...).

  ∗ $tp[a]$ `temporal_relation` $tp[a_i]$,
  where $tp[a]$ is the start time or end time of a and `temporal_relation` is a *point-based* temporal predicate (`precedes` and `concurrent`).

  – ***Codesignation constraints***: a set of constraints p_{ij} specifying restrictions over the possible instantiations of the parameters of the activities participating in a temporal constraint. Given the master $a(\vec{x}, \delta)$ and a slave $a_i(\vec{y}_i, \delta_i)$, a codesignation constraint p_{ij} is of the form x_k `rel` y_{il}, where $x_k \in \vec{x}$, $y_{il} \in \vec{y}_i$ and `rel` $\in \{=, \neq\}$. We assume that $p_{i0} = \top$.

A rich set of temporal relationships is permitted in compatibilities, including: `meets`, `contains`, `before`, `starts`, `equals`, `parallels`, `start_before_end`, `starts_during`, `starts_before`, `starts_after`, `contains_start`, and all their inverse relations. These relations are derived from the thirteen temporal relations defined by Allen (1983)[1].

A ***planning problem*** $\mathcal{P}$ is represented by a pair $\mathcal{P} = \{\mathrm{H}, \mathcal{I}\}$:

- $\mathrm{H} \in \mathbb{N}$ is the end of the ***planning horizon***, meaning that we only care about the behavior of the system with respect to the temporal window $[0, \mathrm{H}]$.

- $\mathcal{I}$ is the ***initial configuration*** represented by a set of activities placed on their corresponding timelines. If we annotate an activity a by a time interval $\tau(a) = [st(a), et(a)]$ (indicating the temporal extent over which a holds), then, for each activity a_i in $\mathcal{I}$, it is possible either to specify the specific position of a_i on the timeline, which means fixing the start and end time of $\tau(a_i)$, or to leave a_i floating on the timeline between the origin and the horizon.

Example: the *Rover* domain As an illustration of a simple domain model, consider a rover equipped with a set of instruments to sample a geological site. We model the subsystems of the rover as timelines: *Battery*, *MobilitySystem*, *Controller*, *IDD*, *Spectrometer*, *Imager* and so on. Each subsystem can only perform certain activities. For example, the *IDD* (Instrument Deployment Device), which is a robotic arm that brings the rover's instruments into contact with rocks, can perform one of the following operations: *Use(inst, rock, 1)*, *Place(rock, 3)*, *Stow(2)*, *Unstow(2)* and *Stowed([1, +inf])*. The first activity consists in using the instrument *inst* against the rock *rock* and lasts 1 time unit. The other specifications are similar. The constraints that regulate the behavior of the *IDD* follow: in order to use an instrument on a rock, the *IDD* must be first unstowed and then properly positioned in the vicinity of the rock. After analyzing a rock, the *IDD* can be placed in another position for performing another experiment or can be stowed. With regard to the interactions between the *IDD* and the other components, the *IDD* can be placed on a rock only if the rover is positioned at that rock and is not changing position. All these constraints are expressed by means of the evolution rules of the activities performed by the *IDD*. We show just a few of them:

$\mathcal{R}$[*Unstow()*] = {
  `meets` *Place(rock)* $\wedge$
  `met_by` *Stowed()*}

$\mathcal{R}$[*Place(rock$_b$)*] = {
  `meets` *Use(inst, rock$_u$)* $\wedge$ *(rock$_u$ = rock$_b$)* $\wedge$
  `met_by` *Unstow()* $\wedge$
  `contained_by` *MobilitySystem.At(rock$_a$)*$\wedge$
  (rock$_a$ = rock$_b$)}

An initial configuration $\mathcal{I}$ for the rover domain can, for example, specify the level of the battery, the position of the mobility system and the status of the instruments at the start of the horizon and, furthermore, can establish that a particular rock should be analyzed within a certain time interval.

[1]It is worth noting that, although the temporal relationships defined by Allen are non-directional and can be inverted, compatibilities cannot be inverted. For example, the compatibility a_i `meets` a_k implies that if a_i exists on a timeline, a_k must also exist, while a_k `met_by` a_i implies that if a_k exists on a timeline, a_i must exist.

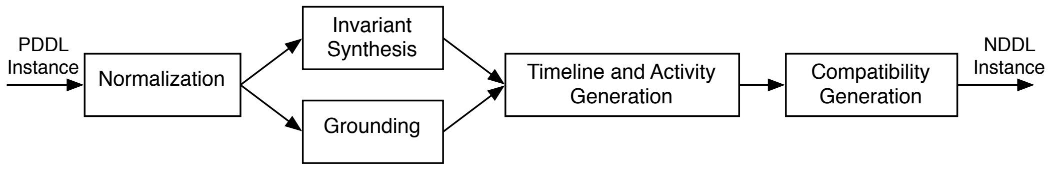

Figure 1: An overview of the translation process

Translation Overview

Given a PDDL2.2 planning instance $\mathrm{I}_p = \{\mathcal{D}_p, \mathcal{P}_p\}$, the translation function Θ applied to I_p produces an equivalent instance $\mathrm{I}_n = \{\mathcal{D}_n, \mathcal{P}_n\}$ in NDDL. An overview of the translation process from PDDL2.2 into NDDL is given in Figure 1. The translation works in five steps. First, the PDDL2.2 instance is *normalized*, i.e. types are removed and conditions and effects are simplified. The next two tasks, which can be performed in any order, are: *invariant synthesis*, which aims at extracting mutual exclusion contraints from an analysis of the domain, and *grounding*, which produces an instance where all the literals are ground. Starting from the invariants provided by the invariant synthesis and the grounded domain, a set of NDDL *timelines* and their associated sets of NDDL *activities* are generated. Finally, the *compatibilities* that describe the behavior of each activity are created on the basis of the PDDL operator specifications and the explanatory frame axioms.

Our translation from PDDL2.2 to NDDL builds on the translation presented in (Helmert 2006b), which transforms a restricted version of PDDL2.2 into the *Multi-valued Planning Task* (MPT) formalism (Helmert 2006a). More specifically, this translation takes as input planning domains specified in PDDL2.2–Level 1, the non-temporal and non-numerical portion of the PDDL language that allows the use of universal and conditional effects, arbitrary first-order formulas in action conditions and goals, and axioms, but does not support durative actions and numeric functions. This translation returns fully instantiated planning tasks written in MPT. The MPT formalism is based on *multi-valued state variables* whose transitions are determined by the application of *instantaneous operators*.

We extend and modify the translation in order to deal with *temporal and numeric* PDDL domains. While the normalization and the grounding steps shown in Figure 1 are simple generalizations of the corresponding steps described in (Helmert 2006b), the remaining steps are different and more complicated. For example, when temporal domains are considered, the discovery of invariants requires proving *mutex conditions* between those durative operators that, when overlapping, could possibly invalidate the invariant conditions. Moreover, the causal evolution of a domain, which in PDDL and MPT is conveyed through the distinction between conditions and effects, must be captured by means of purely temporal constraints in NDDL, since there is no explicit notion of action in NDDL.

In the next sections, we describe the normalization, grounding, invariant synthesis, the timeline and activity generation, and the compatibility creation. For each step, we highlight the similarities and differences with the translation described in (Helmert 2006b).

Due to space limitations, we do not describe the translation of numeric functions. However, that task is straightforward since numeric functions in PDDL are already in a variable/value representation. NDDL provides a notion of *resource* timeline that can be used for translating numeric functions and offers specialized predicates for managing resources (*change, produce, consume* and *use*) that are similar to update assignment operators in PDDL2.2.

Normalization and Grounding

We briefly describe the normalization and the grounding steps, which essentially work as indicated in (Helmert 2006b), with slight modifications for managing time and numeric functions. Normalization performs three tasks: 1) it compiles away types; 2) it transforms goal formula, axiom bodies, operator conditions and conditions of conditional effects into conjunctions of literals; and 3) it transforms each effect into a conjunction of normalized effects, where a normalized effect is a conjunction of literals with an associated set of universal quantifiers and an associated condition. For simplicity, in what follows, we ignore axioms and consider only effects without conditions and quantifiers. However, the techniques presented can be easily extended to manage the general case.

A grounded PDDL domain is a domain where all the literals occurring in the goal formula and operators are ground literals. Grounding a PDDL domain relies on computing an over-approximation of the atoms that can ever be true and then instantiating the goal formula and the operators on them. The over-approximation is calculated by using a relaxed version of the orignal domain where delete effects of operators are ignored and negative literals in conditions are assumed to be true. We refer the reader to (Helmert 2006b) for a detailed description of the procedure for efficiently calculating the reachable atoms of the relaxed domain.

Invariant synthesis

An ***invariant*** is a property of world states such that when it is satisfied by a state s, it is satisfied by all states that are reachable from s. Usually, we are interested in invariants that are satisfied in the initial state. In fact, if an invariant holds in the initial state, it holds in all the reachable states. We look for ***mutual exclusion*** invariants, which state that certain atoms can never be true at the same time, as a preliminary step to synthesizing timelines. To find invariants we identify a set of *invariant candidates* by inspecting the domain. We then check these candidates against a set of properties that assure invariance in order to see whether they are actual invariants or not. If a candidate is not an invariant, in some cases it is possible to refine it so as to make it a real invariant.

Let $\mathrm{I} = (\mathcal{D}, \mathcal{P})$ be a PDDL instance. An ***invariant candidate*** is a tuple $\mathcal{C} = \langle \Phi, \mathrm{F}, \mathrm{V} \rangle$, where Φ is a subset of the atoms in $\mathcal{D}$, and F and V are two disjoint sets of variables, respectively called *fixed* and *counted* variables. Both F and V are subsets of Var$[\Phi]$, which collects the variables in Φ. For example, let us consider a PDDL description of the *Rover* domain and the predicate `at(rover,loc)`. The following is a candidate: $\mathcal{C}_{at} = \langle \{\texttt{at(rover,loc)}\}, \{\texttt{rover}\}, \{\texttt{loc}\} \rangle$, where `rover` is the fixed variable and `loc` the counted variable. An ***instance*** γ of the candidate $\mathcal{C}$ is a function that maps the fixed variables in F to objects of $\mathcal{P}$. Assuming we have an instance with two rovers `rov1` and `rov2`, a possible instance of $\mathcal{C}_{at}$ is $\gamma_{rov1} : \texttt{rover} \rightarrow \texttt{rov1}$. The ***weight*** of γ in a state s is the number of ground instantiations of the variables in V that make some $\phi \in \Phi$ true under γ in s[2]. Thus, considering the *Rover* domain and the instance γ_{rov1}, if we have a state s where the atom `at(rov1,loc1)` holds, then the weight of $\mathcal{C}_{at}$ is 1. Given a ***cardinality set*** $\mathrm{S} = \{x \mid x \in \mathbb{N}\}$, the semantics of a candidate $\mathcal{C}$ is: for all the possible instances γ of $\mathcal{C}$, if the weight of γ is within S in a state s, then it is within S in any successor state s' of s. Thus, if the candidate $\mathcal{C}$ is an actual invariant and holds in the initial state, we have that at most $k = \max(\mathrm{S})$ atoms in Φ are true in any reachable state. Since we are interested in mutually exclusive sets of propositions, we restrict our analysis to the cardinality set $\mathrm{S} = \{0, 1\}$. However, the technique for finding invariants that we will present can be generalized to any cardinality set. Considering the *Rover* domain again, the candidate $\mathcal{C}_{at}$ means that, for all rovers `rover` in the domain, the number of locations `loc` where `at(rover,loc)` is true remains less than or equal to 1 when a transition from one state to another is performed. If we prove that what is stated by the candidate is true and each rover is at maximum 1 location in the initial state, then each rover cannot be at multiple locations at the same time in any reachable state. Hence, for each rover, we can create a timeline that corresponds to the predicate `at` and represents the position of the rover. The activities on this timeline represent the presence of the rover in the various locations that it can occupy.

In order to show that a candidate $\mathcal{C}$ is an actual invariant, we need to guarantee that it holds in the initial state and all the operators in $\mathcal{D}$ keep the weight of any instance γ of $\mathcal{C}$ within the cardinality set $\mathrm{S} = \{0, 1\}$. When an operator satisfies this condition, it does not *threaten* the candidate $\mathcal{C}$. A *sufficient condition* for a candidate $\mathcal{C}$ to be an actual invariant is that it is not *threatened* by any operator.

Given an instance γ of a candidate $\mathcal{C}$, an operator *op* ***does not threaten*** the candidate $\mathcal{C}$ if and only if one of the following conditions holds:

1. The operator *op* does not affect the weight of γ.
2. The operator *op* is *balanced*, i.e. it preserves the weight of γ by checking that the weight is 1 at some timepoint (start/end), decreasing that weight by 1 at that timepoint and increasing the weight by 1 at that same timepoint.
3. The operator *op* increases the weight of γ by 1 at some timepoint (start/end) and its pre-conditions require that the weight of γ is 0 at the same timepoint (start/end).
4. The operator *op* decreases the weight of γ by 1 at some timepoint (start/end).
5. The operator *op* decreases the weight of γ at start or requires that the weight of γ be 0 at start, increases the weight of γ at end, and all operators of type 3 and 5 are *mutex* with op.

In all other cases, the operator *op* threatens the candidate $\mathcal{C}$.

Cases 1 and 2 correspond to the criteria for identifying non-threatening operators used in the translation from PDDL2.2–Level 1 into MPT (Helmert 2006b). Cases 3 and 4 are a generalization of case 2. They can be used not only in the temporal setting, but also in non-temporal planning in order to capture a broader set of invariants. In contrast, Case 5 is specific to temporal planning and accounts for the fact that we accept as non-threatening an operator *op* that is ***temporarily unbalanced*** as long as no other operator disrupts the candidate during the execution of *op*. More specifically, we must guarantee that, when a temporarily unbalanced operator is in execution, no operator can alter the weight of the instances of the candidate. Operators that may disrupt the balance are those of type 3 and type 5. In fact, operators of type 3 only increase the weight of the instances of the candidate and operators of type 5 increase and decrease the weight at two different time points. For example, given the candidate $\mathcal{C}_{at} = \langle \{\texttt{at(rover,loc)}\}, \{\texttt{rover}\}, \{\texttt{loc}\} \rangle$, there is only one operator in the domain *Rover* that affects the weight of the instances of $\mathcal{C}$ and so may threaten it: `navigate`. Suppose that its PDDL specification is:

```
(:durative-action navigate
:parameters (?x - rover ?y - loc ?z - loc)
:duration (= ?duration 5)
:condition (and (over all (can_go ?x ?y ?z))
                (at start (at ?x ?y))
                (over all (visible ?y ?z)))
:effect (and (at start (not (at ?x ?y)))
             (at end (at ?x ?z))))
```

The operator `navigate` is temporarily unbalanced (case 5). In fact, given an instance γ of $\mathcal{C}_{at}$, it decreases the weight of γ at start and increases the weight of γ at end. In order to guarantee that $\mathcal{C}_{at}$ is an invariant, we need to show that all the operators of type 3 and 5 that affect $\mathcal{C}_{at}$ are mutex with `navigate`. This mutex would have been trivially satisfied if `navigate` had the condition that the weight of γ must be 0 over the entire interval at which *op* is applied (*over all* condition). However, this is not our case.

In general, how can we establish whether two durative PDDL operators are mutex or not? Since in PDDL2.2, effects can only happen at the start and end of the operators, and conditions can only be specified at the start, end, and over all, there are nine types of mutex. We refer the reader to (Smith & Jónsson 2002) for a discussion of mutex between actions with general conditions and effects.

Given two durative operators op_1 and op_2, there are nine

[2]The weight of γ is also equal to the cardinality of the set of all ground atoms that unify with some $\phi \in \Phi$ under γ in s.

types of ***mutex operators***:

1. ***Start-Start***: op_1 and op_2 cannot start at the same time if:
   $\exists p \in (\mathrm{Cond}_{start}(op1) \cup \mathrm{Cond}_{all}(op1) \cup \mathrm{Eff}_{start}(op1))$:
   $\neg p \in (\mathrm{Cond}_{start}(op2) \cup \mathrm{Cond}_{all}(op2) \cup \mathrm{Eff}_{start}(op2))$
2. ***End-End***: op_1 and op_2 cannot end at the same time if:
   $\exists p \in (\mathrm{Cond}_{end}(op1) \cup \mathrm{Cond}_{all}(op1) \cup \mathrm{Eff}_{end}(op1))$:
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Cond}_{all}(op2) \cup \mathrm{Eff}_{end}(op2))$
3. ***Start-End***: op_1 cannot start at the time that op_2 ends if:
   $\exists p \in (\mathrm{Cond}_{start}(op1) \cup \mathrm{Cond}_{all}(op1) \cup \mathrm{Eff}_{start}(op1))$:
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Eff}_{end}(op2))$
4. ***Invariant-Start***: op_2 cannot start during op_1 if:
   $\exists p \in \mathrm{Cond}_{all}(op1)$:
   $\neg p \in (\mathrm{Cond}_{start}(op2) \cup \mathrm{Cond}_{all}(op2) \cup \mathrm{Eff}_{start}(op2))$
5. ***Invariant-End***: op_2 cannot end during op_1 if:
   $\exists p \in \mathrm{Cond}_{all}(op1)$:
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Cond}_{all}(op2) \cup \mathrm{Eff}_{end}(op2))$
6. ***Invariant-Invariant***: op_1 and op_2 cannot overlap if:
   $\exists p \in \mathrm{Cond}_{all}(op1) : \neg p \in \mathrm{Cond}_{all}(op2)$

In addition, we have: mutex End-Start (dual to case 3), mutex Start-Invariant (dual to case 4) and mutex End-Invariant (dual to case 5). For brevity, we refer to the above mentioned mutex operators as *mutex-SS*, *mutex-EE*, and so on.

Let us now clarify what checks are needed to establish if a temporarily unbalanced operator *op* of type 5 threatens a candidate. Given an instance γ of a candidate $\mathcal{C}$, *op* is a ***non-threatening unbalanced operator*** for $\mathcal{C}$ if:

- For each operator op' of type 3:
  - if op' increases the weight of γ at start, *op* is mutex-IS or mutex-II with op'.
  - if op' increases the weight of γ at end, *op* is mutex-IE or mutex-II with op'.
- For each operator op' of type 5, *op* is mutex-IE or mutex-II with op'.

The mutex check is redundant and can therefore be avoided if both the following conditions hold:

1. there are no operators of type 3 (which is relatively rare);
2. all the operators of type 5 require that the weight of γ is 1 at start, decreases the weight of γ at start and increases the weight of γ at end (this is a typical resource usage).

In this case, the procedure to identify invariants is much simpler than in the general case. The domain *Rover* and the operator `navigate` respect the conditions 1 and 2. Hence, `navigate` does not threaten the candidate $\mathcal{C}_{at}$.

As with other related techniques (Gerevini & Schubert 2000; Helmert 2006b), the algorithm for finding invariants implements a *guess, check and repair* approach. We start from a simple initial set of candidates that have the following characteristics: the set Φ contains only one atom ϕ and the set V contains only one counted variable. The candidate $\mathcal{C}_{at} = \langle\{$`at(rover,loc)`$\}, \{$`rover`$\}, \{$`loc`$\}\rangle$ is an example. Given a candidate $\mathcal{C} = \langle \Phi, \mathrm{F}, \mathrm{V}\rangle$ that has been rejected because it is threatened by the unbalanced operator *op*, we pick an atom ϕ that unifies with a delete effect of *op* and involves the variables in F and at most one other variable. We then check the new candidate: $\mathcal{C}' = \langle\{\Phi \cup \phi\}, \mathrm{F}', \mathrm{V}'\rangle$. If the new atom ϕ balances the unbalanced add effect of the operator *op* and there are not other operators that threaten $\mathcal{C}'$, then $\mathcal{C}'$ is an invariant. Notice that atoms obtained from constant predicates (Edelkamp & Helmert 1999), i.e. predicates whose atoms have the same truth value in all the states (for example, type predicates), are never added to candidates.

Timeline Generation

Given a PDDL planning instance $\mathrm{I}_p = \{\mathcal{D}_p, \mathcal{P}_p\}$, we use the *invariants* found during the invariant synthesis for creating the timelines of the corresponding NDDL instance $\mathrm{I}_n = \Theta[\mathrm{I}_p]$. More specifically, for each invariant $\mathcal{I} = \langle \Phi, \mathrm{F}, \mathrm{V}\rangle$ and each instance γ of $\mathcal{I}$, we create a timeline $\mathcal{T}[\mathcal{I}, \gamma]$ corresponding to $\mathcal{I}$ and γ. The activities associated with the timeline $\mathcal{T}[\mathcal{I}, \gamma]$ are ground atoms obtained from the atoms in Φ by binding the fixed variables F under γ and by instantiating the counted variables V over the objects in $\mathcal{P}_p$ in all the possible ways. We add an activity $\perp$ to $\mathcal{A}ct[\mathcal{T}]$, which indicates that none of the other activities in the set are executing. The time interval associated with any created activity is $[1, +\infty)$, meaning that the activity can persist for any period of time.

For example, given the domain *Rover* with two rovers and three locations and the invariant $\mathcal{I} = \langle\{$`at(rover,loc)`$\}, \{$`rover`$\}, \{$`loc`$\}\rangle$, we create two timelines $\mathcal{T}_{pos\_r1}$ and $\mathcal{T}_{pos\_r2}$, each representing the position of one rover. The activities associated with the timeline $\mathcal{T}_{pos\_r1}[\mathcal{I},$ `rov1`$]$ are $\mathcal{A}ct[\mathcal{T}_{pos\_r1}] = \{$`at_r1_loc1, at_r1_loc2, at_r1_loc3,`$\perp\}$. The activities associated with $\mathcal{T}_{pos\_r2}[\mathcal{I},$ `rov2`$]$ are similar.

Formally, each reachable ground atom needs to be encoded at least once. However, usually the same atom ends up being in more than one set of activities, each set associated with one timeline. This happens because different invariants can share the same atom. We ignore this redundancy, encode as many timelines as instances of the invariants found and allow a ground atom to appear on multiple timelines[3]. Given a ground atom p that occurs in $\mathcal{D}_p$, we indicate with $\mathrm{T}[p]$ the set of timelines with which the activity p is associated: $\mathrm{T}[p] = \{\mathcal{T} \in \mathrm{T} \mid p \in \mathcal{A}ct[\mathcal{T}]\}$.

Beside the timelines generated from invariants, we also create timelines corresponding to the *operators* in $\mathcal{D}_p$. More specifically, we introduce a timeline $\mathcal{T}_{op}$ for each operator *op* and include two activities into $\mathcal{A}ct[\mathcal{T}_{op}]$: *exe* and *not_exe*, which indicate whether the operator *op* is in execution or not. These activities do not have parameters, but have duration constraints. The duration constraint $[1, +\infty)$ is attributed to the predicate *not_exe*, since it can have any duration. The duration constraints for the PDDL operator *op* are trivially translated into duration constraints for the predicate *exe*. For example, given the domain *Rover* with one rover and two locations, we create a timeline $\mathcal{T}_{navigate\_r1\_loc1\_loc2}$ and a

[3]In practice, we attach the name of the timeline to the ground atom in order to be able to distinguish between the occurrences of it on different timelines

timeline $\mathcal{T}_{navigate\_r1\_loc2\_loc1}$, both characterized by two activities *exe* and *not_exe*.

Compatibility Generation

We generate the compatibilities for the activities that correspond to operators by translating the PDDL specifications of such operators. In particular, given an operator $op = \langle \mathrm{Cond}, \mathrm{Eff} \rangle \in \mathcal{D}_p$ and the corresponding timeline $\mathcal{T}_{op}$, the activity *not_exe* has only two compatibilities: `meets`*(exe)* and `met_by`*(exe)*. The activity *exe* has two similar compatibilities among others. These `meets` and `met_by` compatibilities simply indicate that the timeline $\mathcal{T}_{op}$ alternates *exe* and *not_exe* activities over the entire horizon. However, the activity *exe* has additional compatibilities that describe its interactions with activities on other timelines. These compatibilities are synthesized on the basis of the conditions Cond and the effects Eff of $\mathcal{T}_{op}$. Let us start from conditions and then analyze effects.

Each condition $\mathsf{c} \in \mathrm{Cond}(op)$ belongs to one of the sets Cond_{start}, Cond_{all}, and Cond_{end}. We translate conditions on the basis of the sets to which they belong. We consider positive conditions first.

- **Positive condition at start**: $\mathsf{c} \in \mathrm{Cond}_{start}$

  Since c is a condition at start, any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{c}]$ must ensure the activity *c* at the start of the interval over which the activity *exe* is applied on $\mathcal{T}_{op}$. Given the activities $\mathcal{T}_{op}$*.exe* and $\mathcal{T}$*.c*, there are two possible configurations for them to satisfy the above constraint: either $\mathcal{T}$*.c* starts before and ends after the start of $\mathcal{T}_{op}$*.exe* or $\mathcal{T}$*.c* starts before the start of $\mathcal{T}_{op}$*.exe* and ends at the start of $\mathcal{T}_{op}$*.exe*, i.e.

  $$\mathcal{T}.c.start < exe.start \leq \mathcal{T}.c.end$$

  These constraints are added to the NDDL rule for *exe*:

  $\mathcal{R}[exe()] = \{$`starts_before_end` $\mathcal{T}.c\ \wedge$
  `precedes`$(\mathcal{T}.c.start,\ exe.start)\}$

- **Positive condition over all**: $\mathsf{c} \in \mathrm{Cond}_{all}$

  Since c is a condition over all, any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{c}]$ must ensure the activity *c* over the entire interval over which the activity *exe* is applied on $\mathcal{T}_{op}$. Given the activities $\mathcal{T}_{op}$*.exe* and $\mathcal{T}$*.c*, the start time of $\mathcal{T}_{op}$*.exe* must be greater than or equal to the start time of $\mathcal{T}$*.c* and the end time of $\mathcal{T}_{op}$*.exe* must be less than or equal to the end time of $\mathcal{T}$*.c*,

  $$\mathcal{T}.c.start \leq exe.start \ \wedge\ exe.end \leq \mathcal{T}.c.end$$

  These constraints are added to the NDDL rule for *exe*:

  $\mathcal{R}[exe()] = \{$`contained_by` $\mathcal{T}.c\}$

- **Positive condition at end**: $\mathsf{c} \in \mathrm{Cond}_{end}$.

  Since c is a condition at end, any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{c}]$ must ensure the activity *c* at the end of the interval over which the activity *exe* is applied on $\mathcal{T}_{op}$. Given the activities $\mathcal{T}_{op}$*.exe* and $\mathcal{T}$*.c*, there are two possible configurations for them to satisfy the above constraint: either $\mathcal{T}$*.c* starts before and ends after the end of $\mathcal{T}_{op}$*.exe* or $\mathcal{T}$*.c* starts before the end of $\mathcal{T}_{op}$*.exe* and ends at the end of $\mathcal{T}_{op}$*.exe*, i.e.

  $$\mathcal{T}.c.start < exe.end \leq \mathcal{T}.c.end$$

  These constraints are added to the NDDL rule for *exe*:

  $\mathcal{R}[exe()] = \{$`ends_before` $\mathcal{T}.c\ \wedge$
  `precedes`$(\mathcal{T}.c.start,\ exe.end)\}$

Given a negative condition $\mathsf{c} \in \mathrm{Cond}_{start}$ and any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{e}]$, we have two cases:

- **Protected negative condition at start**

  If the operator op has another condition c' such that $\mathsf{c}' \in \mathcal{A}ct[\mathcal{T}]$ and $\mathsf{c}' \in \mathrm{Cond}_{start}$, then we do not directly translate c. In fact, this negative condition will be automatically translated when the condition c' is translated.

- **Unprotected negative condition at start**

  If a condition c' of such type does not exist, then any activity in $\mathcal{A}ct[\mathcal{T}]$ different from c can possibly hold after the start time of op. We translate this case by creating a disjunction between different compatibilities controlled by a guard variable. More specifically, we create a set of temporal constraints like those illustrated above for positive conditions for each activity in $\mathcal{A}ct[\mathcal{T}]$ different from c. Then, we create a guard constraint for each one of these sets of temporal constraints using the same guard variable. In the end, we obtain a set of mutually exclusive compatibilities involving *exe* and the activities in $\mathcal{A}ct[\mathcal{T}]$ excluding c.

The same holds for **negative condition over all** and **at end**.

Let us now turn our attention to effects. Each effect $\mathsf{e} \in \mathrm{Eff}(op)$ belongs to one of the following sets: Add_{start}, Add_{end}, Del_{start} and Del_{end}. We translate effects on the basis of the sets to which they belong.

- **Add effect at start**: $\mathsf{e} \in \mathrm{Add}_{start}$

  Since e is an add effect at start, any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{e}]$ must ensure the activity *e* at the start of the interval over which the activity *exe* is applied on $\mathcal{T}_{op}$. Given the activities $\mathcal{T}_{op}$*.exe* and $\mathcal{T}$*.e*, there are two possible configurations for them to satisfy the above constraint: either $\mathcal{T}$*.e* starts before and ends after the start of $\mathcal{T}_{op}$*.exe* or $\mathcal{T}$*.e* starts at the start of $\mathcal{T}_{op}$*.exe*, i.e.

  $$\mathcal{T}.e.start \leq \mathcal{T}_{op}.exe.start < \mathcal{T}.e.end.$$

  These constraints are added to the NDDL rule for *exe*:

  $\mathcal{R}[exe()] = \{$`starts_after` $\mathcal{T}.e\ \wedge$
  `precedes`$(exe.start, \mathcal{T}.e.end)\}$

- **Add effect at end**: $\mathsf{e} \in \mathrm{Add}_{end}$

  Since e is an add effect at end, any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{e}]$ must ensure the activity *e* at the end of the interval over which the activity *exe* is applied on $\mathcal{T}_{op}$. Given the activities $\mathcal{T}_{op}$*.exe* and $\mathcal{T}$*.e*, there are two possible configurations for them to satisfy the above constraint: either $\mathcal{T}$*.e* starts before and ends after the end of $\mathcal{T}_{op}$*.exe* or $\mathcal{T}$*.e* starts at the end of $\mathcal{T}_{op}$*.exe*, i.e.

  $$\mathcal{T}.e.start \leq \mathcal{T}_{op}.exe.end < \mathcal{T}.e.end.$$

  These constraints are added to the NDDL rule for *exe*:

  $\mathcal{R}[exe()] = \{$`ends_after_start` $\mathcal{T}.e\ \wedge$
  `precedes`$(exe.end, \mathcal{T}.e.end)\}$

Let us now consider the delete effects Del. Given an atom $\mathsf{e} \in \mathrm{Del}_{start}$ and any timeline $\mathcal{T} \in \mathrm{T}[\mathsf{e}]$, we have two cases:

- **Protected delete effect at start**

  If the operator op has another effect e' such that $\mathsf{e}' \in \mathcal{A}ct[\mathcal{T}]$ and $\mathsf{e}' \in \mathrm{Add}_{start}$, then we do not directly translate e. In fact, this delete effect will be automatically translated when the add effect e' is translated.

- **Unprotected delete effect at start**

If an effect of such type does not exist, then the timeline $\mathcal{T}$ assumes an undefined value during the execution of the operator *exe*. Therefore, we add the following constraint to the NDDL rule for *exe*: $\mathcal{R}[\textit{exe()}] = \{\texttt{contained\_by}\ \mathcal{T}.\bot\}$

The same holds for **delete effects at end**.

We consider again the *Rover* domain with one rover and two locations and in particular the timeline $\mathcal{T}_{navigate\_r1\_loc1\_loc2}$ and the activity *exe*. The PDDL specification of the operator `navigate_loc1_loc2` is translated as follows:

$\mathcal{R}[\textit{exe()}] = \{$
$\quad \texttt{starts\_before\_end}\ \mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc1}\ \wedge$
$\quad \texttt{precedes}(\mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc1.start, exe.start})\ \wedge$
$\quad \texttt{contained\_by}\ \mathcal{T}_{pos\_r1}.\bot\ \wedge$
$\quad \texttt{ends\_after\_start}\ \mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc2}\ \wedge$
$\quad \texttt{precedes}(\textit{exe.end}, \mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc2.end})\}$

Figure 2 illustrates the configuration of the timelines $\mathcal{T}_{pos\_r1}$ and $\mathcal{T}_{navigate\_r1\_loc1\_loc2}$.

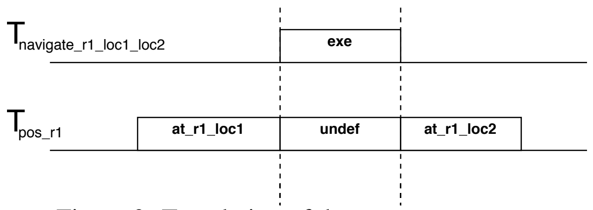

Figure 2: Translation of the operator *navigate*

From Figure 2 it appears that we can use more restrictive constraints for translating the operator `navigate_r1_loc1_loc2`, and in particular:

$\mathcal{R}[\textit{exe()}] = \{$
$\quad \texttt{met\_by}\ \mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc1}\ \wedge$
$\quad \texttt{equals}\ \mathcal{T}_{pos\_r1}.\bot\ \wedge$
$\quad \texttt{meets}\ \mathcal{T}_{pos\_r1}.\textit{at\_r1\_loc2}\}$

We can deduce these more restrictive constraints if we look at all the conditions and effects that affect the timeline $\mathcal{T}_{pos\_r1}$ at once. Writing more restrictive constraints for translating an operator *op* is more costly than translating each condition and effect in isolation, as we did above, since it requires analyzing all the possible combinations between conditions and effects. However, we identify some frequent cases for which more specific constraints can be used and pre-compile them. The case in Figure 2 is one of these common cases: there are a condition and a delete effect at start, an add effect at end and all involving the same timeline.

Translating Frame Axioms

Let *p* be an activity that translates a positive PDDL proposition p. The evolution rule for *p* is composed of two disjunctions: the first disjunction describes which activities cause the start of *p*, whereas the second disjunction accounts for the activities that cause the end of *p*. Basically, if the activity *p* holds on a timeline $\mathcal{T}$, one of the activities in the first disjunction must occur and trigger the start of *p* and one of the activities in the second disjunction must occur and cause the end of *p*. The activities that can start and end *p* correspond to the translation of those PDDL operators that contain the proposition p or its negation in their effects. Thus, when we write the compatibilities for a proposition p, we are basically implementing the ***explanatory frame axioms*** for p and ¬p (Haas 1987), similarly to what is done in SAT-encodings of planning domains (Ernst, Millstein, & Weld 1997).

We start by synthesizing compatibilities that enumerate the operators that can cause the start of *p*. We collect all the operators $op_1, \ldots, op_k$ in $\mathcal{D}_p$ whose *effects at start* contain p. For each one of these operators, we create a disjunct of the form: $\texttt{starts}\ (\mathcal{T}_{op_i}.\textit{exe})$. In this case, *p* starts holding on its timeline $\mathcal{T}$ when *exe* is applied over $\mathcal{T}_{op_i}$. Then, we collect all the operators $op_{k+1}, \ldots, op_n$ whose *effects at end* contain p. For each one of these operators, we create a disjunct of the form: $\texttt{met\_by}\ (\mathcal{T}_{op_i}.\textit{exe})$. In this case, *p* starts holding at the end of the interval over which *exe* is applied on $\mathcal{T}_{op_i}$. Finally, we create one guard constraint for each disjunct. All the disjuncts share the same guard variable. In the end, we obtain a disjunctive rule with n disjuncts controlled by a single guard variable:

$\mathcal{R}[\textit{p()}] = \{$
$\quad \textit{guard} = 0 \Rightarrow \texttt{starts}\ \mathcal{T}_{op_1}.\textit{exe}$
$\quad \ldots$
$\quad \textit{guard} = k \Rightarrow \texttt{starts}\ \mathcal{T}_{op_k}.\textit{exe}$
$\quad \textit{guard} = k+1 \Rightarrow \texttt{met\_by}\ \mathcal{T}_{op_{k+1}}.\textit{exe}$
$\quad \ldots$
$\quad \textit{guard} = n \Rightarrow \texttt{met\_by}\ \mathcal{T}_{op_n}.\textit{exe}\}$

The disjunction says that, if the activity *p* starts holding on its associated timeline $\mathcal{T}$, one of the activities among $\mathcal{T}_{op_1}$.*exe*, …, $\mathcal{T}_{op_n}$.*exe* must have triggered it. Thus, these compatibilities for *p* account for the operators that cause the start of *p*.

We then introduce additional compatibilities that account for the operators that cause the end of *p*. We collect all the operators $op_1, \ldots, op_n$ whose *effects at start* contain ¬p. For each one if these operators, we create a disjunct of the form: $\texttt{meets}(\mathcal{T}_{op_i}.\textit{exe})$. In this case, *p* ceases to hold on its timeline $\mathcal{T}$ when *exe* is applied over $\mathcal{T}_{op_i}$. Then, we collect all the operators $op_{k+1}, \ldots, op_n$ whose *effects at end* contain ¬p. For each one of these operators, we create a disjunct of the form: $\texttt{ends}\ (\mathcal{T}_{op_i}.\textit{exe})$. In this case, *p* ceases to hold at the end of the interval over which *exe* is applied on $\mathcal{T}_{op_i}$. Finally, we create one guard constraint for each disjunct. All the disjuncts share the same guard variable. Thus, we obtain a disjunctive rule with n disjuncts controlled by a single guard variable:

$\mathcal{R}[\textit{p()}] = \{$
$\quad \textit{guard} = 0 \Rightarrow \texttt{meets}\ \mathcal{T}_{op_1}.\textit{exe}$
$\quad \ldots$
$\quad \textit{guard} = k \Rightarrow \texttt{meets}\ \mathcal{T}_{op_k}.\textit{exe}$
$\quad \textit{guard} = k+1 \Rightarrow \texttt{ends}\ \mathcal{T}_{op_{k+1}}.\textit{exe}$
$\quad \ldots$
$\quad \textit{guard} = n \Rightarrow \texttt{ends}\ \mathcal{T}_{op_n}.\textit{exe}\}$

The disjunction says that, if the activity *p* ends, one of the activities among $\mathcal{T}_{op_1}$.*exe*, …, $\mathcal{T}_{op_n}$.*exe* must have ended it. Thus, these compatibilities for *p* account for the operators that cause the end of *p*.

For example, consider the *Rover* domain with one rover and three locations. Below, we show the compatibilities for the activity at_r1_loc1.

$\mathcal{R}[\textit{at\_r1\_loc1()}] = \{$
$\quad \textit{guard}_1 = 0 \Rightarrow \texttt{met\_by}\ \mathcal{T}_{navigate\_r1\_loc2\_loc1}.\textit{exe}$
$\quad \textit{guard}_1 = 1 \Rightarrow \texttt{met\_by}\ \mathcal{T}_{navigate\_r1\_loc3\_loc1}.\textit{exe}$
$\quad \textit{guard}_2 = 0 \Rightarrow \texttt{meets}\ \mathcal{T}_{navigate\_r1\_loc1\_loc2}.\textit{exe}$
$\quad \textit{guard}_2 = 1 \Rightarrow \texttt{meets}\ \mathcal{T}_{navigate\_r1\_loc1\_loc3}.\textit{exe}\}$

Basically, the first disjunction says that, in order for the activity at_r1_loc1 to start holding on $\mathcal{T}_{pos\_r1}$, one of the following activities must hold and end at the start time of at_r1_loc1: $\mathcal{T}_{navigate\_r1\_loc2\_loc1}.\textit{exe}$ or $\mathcal{T}_{navigate\_r1\_loc3\_loc1}.\textit{exe}$. The rover is at location loc1 if it has navigated to that location from loc2 or from loc3. The second disjunction says that, in order for the activity at_r1_loc1 to cease to hold on $\mathcal{T}_{pos\_r1}$, one of the following activities must hold and start at the end time of at_r1_loc1: $\mathcal{T}_{navigate\_r1\_loc1\_loc2}.\textit{exe}$ or $\mathcal{T}_{navigate\_r1\_loc1\_loc3}.\textit{exe}$. The rover leaves location loc1 when it starts navigating toward another location, i.e. loc2 or loc3.

Figure 3 illustrates the configuration of the timelines $\mathcal{T}_{pos\_r1}$ and $\mathcal{T}_{navigate\_r1\_loc1\_loc2}$.

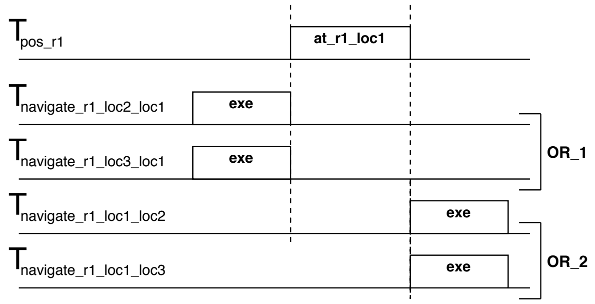

Figure 3: Compatibilities for the activity *at_r1_loc1*

Conclusions and Future Work

We have presented a translation from PDDL2.2 into NDDL. These two languages are significantly different and are used within two alternative planning paradigms: the classical framework and the constraint-based temporal framework. The translation can be easily adapted to be used for several constraint-based temporal planners, which can then be tested on the standard IPC domains and compared with state-of-the-art planners based on different paradigms. Thus, the method presented is a general approach for extracting a temporal multi-valued representation based on temporal constraints from an action-centered propositional representation.

Previous work on translating a propositional representation into a variable/value representation was limited to non-temporal and non-numeric domains (Helmert 2006b). In contrast, our translation tackles temporal and numeric domains. Translating such tasks is more complex than handling tasks with instantaneous actions. Particular care is required to identify invariants since in some cases durative operators can interact so as to invalidate the invariant conditions. We identify mutex conditions on operators that, when satisfied, protect the invariant conditions.

We have implemented two versions of the translation. The first tackles exclusively non-numeric and non-temporal domains and uses the MPT formalism as an intermediate step between PDDL and NDDL. This translation is a direct adaptation of the technique presented in (Helmert 2006b). We have then developed a new version of the translation that tackles metric and temporal domains, which is the one presented here.

In future work, we intend to explore the possibility of developing a translation from NDDL into PDDL. We will identify increasingly complex subsets of the NDDL language and translate such subsets into suitable versions of the PDDL language. It is still unclear whether PDDL can capture the full expressive power of NDDL. The availability of a translation from NDDL into PDDL would be valuable since it will allow us to use and test state-of-the-art planners developed by the academic community on real-world problem specifications, and so promote a full sharing of domains and ideas between the two communities.

Acknowledgements

We thank Malte Helmert for making his code available for translating PDDL instances into MPT tasks. We thank Peter Jarvis, Amedeo Cesta and Gabriele Röger for discussion on translating PDDL2.2 into a constraint-based multi-valued variable representation. This work was supported in part by the Automation for Operations (A4O) project of the NASA Exploration Technology Development Program.

References

Allen, J. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Bresina, J.; Jónsson, A.; Morris, P.; and Rajan, K. 2005. Activity planning for the Mars Exploration Rovers. In *Proc. of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*, 40–49.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; B.Smith; Fisher, F.; Barret, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space missions operations. In *6th International Conference on Space Operations (SpaceOps 2000)*.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. of the Fifth European Conference on Planning (ECP'99)*, 135–147.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg.

Ernst, M.; Millstein, T. D.; and Weld, D. S. 1997. Automatic SAT-compilation of planning problems. In *Proc. of the Fifteen International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1169–1177.

Frank, J., and Jónsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints* 8(4):339–364. Special Issue on Planning.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):5–45.

Gerevini, A., and Schubert, L. 2000. Discovering state constraints in discoplan: Some new results. In *In Proc. of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 761–767.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 61–67. AAAI Press.

Haas, A. 1987. The case for domain specific frame axioms. In *The Frame Problem in Artificial Intelligence: Proc. of the 1987 Workshop*, 343–348. Morgan Kaufmann Publishers, Inc.

Helmert, M. 2006a. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2006b. *Solving Planning Tasks in Theory and Practice*. Ph.D. Dissertation, University of Freiburg.

Smith, D., and Jónsson, A. 2002. The logic of reachability. In *Proc. of the Sixth International Conference on AI Planning and Scheduling (AIPS-02)*, 379–387.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review* 15(1):61–94.