Extending PDDL to nondeterminism, limited sensing and iterative conditional plans

Piergiorgio Bertoli[1], **Alessandro Cimatti**[1], **Ugo Dal Lago**[2], **Marco Pistore**[1,3]
[1] IRST - Istituto Trentino di Cultura, 38050 Povo, Trento, Italy
[2] Università di Bologna, 40127 Bologna, Italy
[3] Università di Trento, 38050 Povo, Trento, Italy
{bertoli,cimatti}@irst.itc.it
dallago@cs.unibo.it
pistore@dit.unitn.it

Abstract

The last decade has witnessed a dramatic progress in the variety and performance of techniques and tools for classical planning. The existence of a de-facto standard modeling language for classical planning, PDDL, has played a relevant role in this process. PDDL has fostered information sharing and data exchange in the planning community, and has made international classical planning competitions possible.

At the same time, in the last few years, non-classical planning has gained considerable attention, due to its capability to capture relevant features of real-life domains which the classical framework fails to express. However, no significant effort has been made to achieve a standard mean for expressing non-classical problems, making it difficult for the planning community to compare non-classical approaches and systems.

This paper provides a contribution in this direction. We extend PDDL in order to express three very relevant features outside classical planning: uncertainty in the initial state, nondeterministic actions, and partial observability. NPDDL's extensions are designed to retain backward compatibility with PDDL, and with an emphasis on compactness of the representation. Moreover, we define a powerful, user-friendly plan language to go together with NPDDL. The language allows expressing program-like plans with branching and iterations structures, as it is necessary to solve planning problems in the presence of initial state uncertainty, nondeterminism and partial observability. We are testing NPDDL's ability to cope with a variety of problems, as they are handled by a state-of-the-art planner, MBP.

Introduction

Planning is an extremely active field of research. Because of its potential in terms of real-life applications, a wide variety of approaches have been developed, and several powerful automated planning systems have been designed to cope with complex problems. The existence of PDDL, a de-facto standard language for planning has been crucial for fostering the reuse of models, establishing a common repository of problems, and comparing and integrating systems, as it is evident from the results of the international competitions (McDermott 2000; Bacchus 2000; Fox & Long 2002).

PDDL is however limited to "classical" planning problems, and it is unable is to capture many relevant features that are important for modeling real world domains. In particular:

- the initial situation may be only partially specified;
- it is often unrealistic to assume that actions have a fully predictable outcome;
- the status of the domain may be only partially observable by the plan executor, and sensing might convey unreliable results. In general, in most cases it is unrealistic to assume the omniscience of the plan executor;
- problems of interest may often go beyond planning to reach a condition; in general, it is highly desirable to express properties about the whole execution of a plan, to state e.g. safety or maintenance requirements;
- sequences of actions are not sufficient to express solutions for problems and domains with the aforementioned features. More complex structures, e.g. loops and conditions, are required.

The growth of scientific interest for expressive planning, taking into account nondeterminism, partial observability and complex temporal goals, is evident. A number of publications and events (e.g.(C. Boutilier & Koenig 2002; Cimatti *et al.* 2001a)) have taken place, and increasingly many powerful planning systems are designed to deal with (combinations of) the features above, using a variety of approaches (Bonet & Geffner 2000a; Weld, Anderson, & Smith 1998; Smith & Weld 1998; Bertoli *et al.* 2001; Castellini, Giunchiglia, & Tacchella 2001; Kabanza 1999; Doherty & Kvarnström 2001; Rintanen 1999). However, no significant effort has been made to provide a standard mean for expressing nondeterministic, partially observable planning domains.

This paper presents the NPDDL language, a first contribution in the direction of a general PDDL-like language for planning with incomplete information. We first describe an extension to the current standard PDDL2.1 (Fox & Long 2001; Ghallab *et al.* 1998) to allow for description of nondeterministic, partially observable planning domains. Furthermore, we add a rich, user-friendly plan language that captures the iterative, branching plan structures needed to plan for domains involving the aforementioned features. The language we describe is the input to MBP (Bertoli *et al.* 2001), a state-of-the art planner integrating plan synthesis, valida-

tion and simulation within the planning via symbolic model checking framework. The input language to MBP also provides a means to express a rich class of temporal requirements; this shows the potential for further extension of the standard.

The paper is structured as follows. We first present a conceptual reference model for planning with incomplete information. We then introduce the NPDDL syntax, and show how it allows for the description of the features of interest. We present some results, compare NPDDL with the related work, and discuss some open issues. A BNF characterization of NPDDL is available at `http://sra.itc.it/tools/mbp/npddl.ps`.

The Framework

As a reference example, we consider a simple domain featuring uncertainty in the initial condition, nondeterminism and partial observability, and model a planning problem for it. The domain consists of a line of rooms that can be traversed by a robot. A printer is situated at one end of the line, and it may print a paper everytime its exit tray is empty. Each printed paper has a banner, where the destination room is reported. The banner can be read by the robot. The robot can pick up a paper at the printer, and can leave it at an office. The robot can only check the printer tray's status when at the printer's place. To identify its position, the robot is equipped with a sensor that detects whether the printer is in the same room. For this domain, we consider the problem of having the robot correctly deliver all the papers queued at the printer - the length and content of the queue being unknown, and the robot being initially positioned anywhere. A solution plan must consider the available sensing, and requires an iterative conditional structure whose execution is possibly infinite.

We rely on a simple, general framework to provide a semantic foundation to our work; this is depicted in fig.1. In our view, a domain is a (possibly nondeterministic) finite state machine, whose state evolves according to the actions received as input, and to the previous state. The domain conveys information to the plan by means of observations. We think of a plan as a deterministic finite state machine, which determines the actions to be performed according to the observations from the domain, and to its state. The execution of a plan in the domain can be thought of as an iteration where (i) an observation is given as input to the plan, (ii) the status of the plan evolves and the next action is determined, (iii) the action affects the domain status and the possible observations. A formal characterization of the framework is provided in (Bertoli *et al.* 2002b). In the following it is sufficient to limit the discussion to the underlying intuitions.

With this approach, it is possible to encompass incompletely specified initial conditions, and nondeterministic action effects. Incompletely specified initial conditions are represented by specifying a set of possible initial states of the domain. In our example, the initial states cover every possible position of the robot, and every possible content of the printer queue.

Action effects are characterized by associating actions with transitions from state to state. Nondeterministic ac-

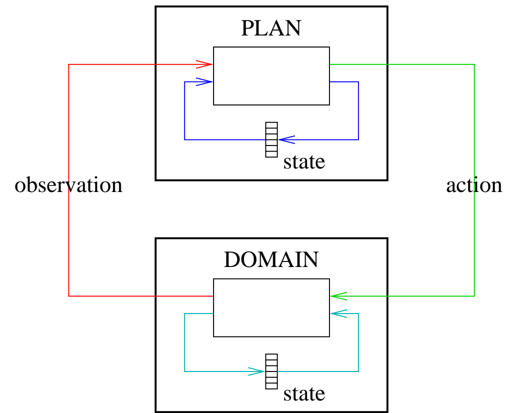

Figure 1: Planning Framework

tion effects are obtained by associating an action with several transitions from the same initial state to different result states. The domain is therefore characterized as a relation, rather than as a partial function as in the deterministic case. In our example, the effect of picking a paper may result in several states, where the printer queue may be either empty, or may have on top papers with different banners.

Our modeling of observations maps states of the domain, which may not be directly observable, into observation variables, the value of which can be directly inspected at run time by the executor. Our approach, that is somewhat similar to (Goldman & Boddy 1996; Bonet & Geffner 2000b; Nourbakhsh & Genesereth 1996), also allows us to capture noisy/unreliable and partial sensing, where information is available only under specific conditions. This is achieved by relating values to observation variables not as a function of the domain state, but as relations. In this way, when an observation variable conveys no information, it can assume any value in its range, nondeterministically. As a special case, an observation O may be undefined over a domain state, by making every value of O possible on S. Within this framework, different forms of sensing can be modeled. With "automatic sensing" (Tovey & Koenig 2000), information can always be acquired, as usual in embedded controllers, where a signal from the environment is sampled and acquired at a fixed rate, latched and internally available. Observations resulting from the execution of "sensing actions" (Cassandra, Kaelbling, & Littman 1994; Weld, Anderson, & Smith 1998; Pryor & Collins 1996; Bonet & Geffner 2000a) can be modeled by representing in the domain state the last executed action, and exploiting the possibility for describing undefinedness. In our example, the domain features three observations: one, in case the robot is at the printer's place, indicates whether the tray is full or not, and provides no information otherwise; another signals whether the robot is at the printer's place or not; the last signals to the robot the current banner of the carried paper. These are easily modeled exploiting the possibility of unreliable/noisy sensing.

The idea of plans as finite state machines makes it possible to express complex courses of actions. In particular, we take into account the possibility of branching (which is able to exploit the information acquired at run-time in order to tackle the nondeterminism of the domain) iteration (which enables to express possibly cyclic, trial-and-error courses of actions). Such more expressive plans are required to solve problems under partial observability and with nondeterminism.

The NPDDL Language

NPDDL is a language that extends PDDL to encompass the intuitions outlined in previous section. NPDDL provides a set of independent extensions to describe incompletely specified initial conditions, nondeterministic action effects, and partial observability.

NPDDL starts from the standard PDDL 2.1. In particular, it starts from the level 2 of PDDL 2.1, thus inheriting a compact and clearly defined set of constructs to handle numeric and conditional effects, and first-order quantification, as well as STRIPS effects on predicates. The aspects related to the higher levels (level 3 to level 5) of PDDL2.1, such as durative actions, are not taken into account with this version of NPDDL.

In the following, we assume that we are referring to a specific ground planning domain, where $\mathcal{F}$ is the finite set of fluents. The instantiation of operators and predicates to the objects in the domain in NPDDL follows the very same schema as in PDDL. Our discussion is based on possibly non-boolean fluents; each $f_i \in \mathcal{F}$ is associated with a finite range of values. An assignment, denoted in the following by $a, a_1, a_2, \ldots$, is an expression that maps a fluent f_i on a value v_j of the associated range. (In the following, without loss of generality, we do not explicitly treat predicates, that can be seen as fluents with values over the binary range `boolean`.) NPDDL allows for n-ary constructs whenever PDDL does. However, we restrict our discussion to their binary version, leaving the trivial generalization to the reader.

Incompletely Specified Initial Conditions

PDDL allows for the specification of completely characterized initial condition, i.e. of a single initial state. This is described with the `:init` statement, containing a set of conjunctive assignments $i \doteq \{i_1, \ldots, i_n\}$. Each conjunctive assignment is a (possibly nested) conjunction of fluent assignments. The initial state specification i is in fact an implicit conjunction of fluent assignments; the top-level conjunction is left implicit. Fluents in $\mathcal{F}$ that are not explicitly assigned are treated according to the Closed-World Assumption (CWA) and given a value (by convention, we assume the first value in the range is given). More formally, let $\textsc{Assigned}(i)$ be the set of fluents assigned by i. Let $F \subseteq \mathcal{F}$ be a set of fluents, and $\textsc{Cwa}(F)$ the CWA-implied assignment to the set of fluents F, then the initial state is identified by

$$\textsc{Initial}(i) \doteq i \wedge \textsc{Cwa}(\mathcal{F} \setminus \textsc{Assigned}(i))$$

The set of assigned fluents is computed as follows:

1. $\textsc{Assigned}(\text{“}\texttt{(assign } f\ v\texttt{)}\text{”}) = \{f\}$
2. $\textsc{Assigned}(\text{“}\texttt{(and } i_1\ i_2\texttt{)}\text{”}) = \textsc{Assigned}(i_1) \cup \textsc{Assigned}(i_2)$

In NPDDL an incompletely specified initial condition is characterized by describing the set of possible initial states. To allow for multiple initial states, NPDDL introduces a `oneof` construct ($\texttt{oneof } i_1 \ldots i_n$), meaning that exactly one of the specifications described by i_j is active. Thus a statement of the form ($\texttt{oneof } i_1 \ldots i_n$) is associated to a set of partial assignments. The corresponding set of initial states is simply obtained by applying CWA to each of those partial assignments. The `oneof` construct can be combined arbitrarily with PDDL constructs, in order to allow for an independent description of uncertainty over distinct sets of domain fluents. As such, a generic PDDL initial state specification i is associated to a set of partial assignments $\textsc{AssSet}(i)$. This is defined as follows, considering the semantics of `oneof`, and that of `and` (which distributes upon the set of possible assignments):

- $\textsc{AssSet}(\text{“}\texttt{(assign } f\ v\texttt{)}\text{”}) = \{\texttt{(assign } f\ v\texttt{)}\}$
- $\textsc{AssSet}(\text{“}\texttt{(and } i_1\ i_2\texttt{)}\text{”}) = \bigcup \{\texttt{(and } b_1\ b_2\texttt{)}\}$ where $b_j \in \textsc{AssSet}(i_j)$, for $j \in \{1, 2\}$
- $\textsc{AssSet}(\text{“}\texttt{(oneof } i_1\ \ldots\ i_n\texttt{)}\text{”}) = \bigcup_{1 \leq j \leq n} \textsc{AssSet}(i_j)$

The initial states are then built by complementing each partial assignment with the CWA, as follows:

$$\textsc{Initials}(i) \doteq \{\textsc{Initial}(i_j) | i_j \in \textsc{AssSet}(i)\}$$

Notice that, if the initial condition is completely specified, NPDDL maps back to PDDL: $\textsc{AssSet}(i) = \{i\}$ and $\textsc{Initials}(i) = \{\textsc{Initial}(i)\}$.

NPDDL also contains the `unknown` construct, to express the set of all possible assignments to a generic fluent f. This construct enables us to avoid explicitly listing every possible value of f. “(unknown f)” is equivalent to specifying a `oneof` statement on any possible value of the type of f:

$$\texttt{(unknown } f\texttt{)} \doteq \texttt{(oneof (assign } f\ v_1\texttt{)...(assign } f\ v_n\texttt{))}$$

where $\{v_1, \ldots, v_n\}$ is the finite range of f.

Example Initially, the status of the queue is unspecified, and the robot may be in any room. This can be easily expressed in NPDDL by a conjunction of `unknown` statements.

```
(:init
   (unknown (robot_room))
   (unknown (paper_at_printer))
   (not (papers_around))
   (not (arm_busy)))
```

In the appendix, we provide a full NPDDL modeling for the reference domain and problem.

Nondeterministic Action Effects

Action effects in PDDL

In PDDL, actions are deterministic: the execution of an action in a domain state S result in a single outcome S'. The

way S' is determined depends on the interaction of PDDL's effect features: conditional effects, quantifiers and inertia handling. In order to describe actions in NPDDL, we first reduce the general structure of PDDL to a simple normal form. First, universal quantifications can be eliminated by replacement with n-ary conjunctions. Then, we observe that nested conditional effects can be eliminated by rewriting. In Appendix, we show that it is enough to consider top-level conditional effects whose branches are mutually exclusive. When executing an action A featuring such a set of top-level conditional effects, exactly one of the conditions holds, triggering the associated condition-free effect E. E's outcome consists in the (possibly partial) assignment described by E, complemented by assigning "inertially" those fluents not assigned by E:

$$\text{OUTCOME}(E) \doteq E \wedge \text{INERTIA}(\mathcal{F} \setminus \text{ASSIGNED}(E))$$

where $\text{INERTIA}(E)$ assigns each fluent in E its current value. Notice the similarity with the way the initial state is computed.

Action Effects in NPDDL

NPDDL allows for nondeterministic actions, whose execution on a domain state S may have several possible outcomes. NPDDL uses the `oneof` construct in action effects for this purpose; intuitively, in an action effect, (`oneof` $e_1 \ \dots \ e_n$) indicates that exactly one of the e_i effects will take place, and, as such, it is associated to a set of partial assignments (those resulting from e_i). Since NPDDL allows for a general combination of `oneof` statements with PDDL's constructs, a generic condition-free NPDDL effect e is associated to a set of partial assignments. This is computed by $\text{ASSSET}(E)$.

The set of possible outcomes of a nondeterministic effect E simply results from the set of (possibly partial) assignments $\text{ASSSET}(E)$:

$$\text{OUTCOMES}(E) \doteq \{\text{OUTCOME}(e_i) | e_i \in \text{ASSSET}(E)\}$$

Notice that, if the effect is fully deterministic, $\text{ASSSET}(E) = \{E\}$, implying $\text{OUTCOMES}(E) = \{\text{OUTCOME}(E)\}$, i.e. NPDDL maps back to PDDL.

Example Uncertain action effects are in that, when picking a paper, it may be the last or not, and on the information reported by the banner. This is modeled in the `pick_paper` operator

```
(:action pick_paper
 :precondition (and (paper_at_printer)
                    (not (arm_busy))
                    (= (robot_room) 0))
 :effect  (and
            (arm_busy)
            (unknown (paper_banner))
            (unknown (paper_at_printer))))
```

It is possible to use the `unknown` construct in action effects in order to express the assignment of a fluent to any value. As for the initial condition, this avoids the explicit listing of assignment for each possible fluent value. The `unknown` construct in action effects is handled similarly to the case of initial states: "(`unknown` f)" is equivalent to

(`oneof` (`assign` f v_1)...(`assign` f v_n))

where $\{v_1, \dots, v_n\}$ is the finite range of f.

Compactness of the representation NPDDL is designed to compactly model domains where actions could possibly have high branching rates, and problems with a possibly large number of initial states. To this end, it is very important that `oneof` constructs can be arbitrarily nested and combined with other operators, in order to compactly specify problems with high degrees of uncertainty. A solution where `oneof` constructs are allowed only at top level would result in very clumsy specifications, since it would force considering every combination of the effects of nondeterminism/initial uncertainty over each fluent. This is also clear in the reference example, where independent facets of initial uncertainty (robot position and print queue) would otherwise result in a lengthy state enumeration.

The characterization provided in this section is by no means intended to suggest a practical way to deal with NPDDL, e.g. to build a parser and a domain constructor. One of the challenges in planning with nondeterministic domains is to be able to internally represent domains without having to enumerate their initial states and the possible action outcomes. Symbolic techniques (Rintanen 2002; Bertoli *et al.* 2001; Castellini, Giunchiglia, & Tacchella 2001) appear to have significant leverage in this respect.

Partial Observability in NPDDL

In order to allow for partial observability, NPDDL introduces a notion of "observation". Similar to (Bonet & Geffner 2000b; Nourbakhsh & Genesereth 1996), and in accord to the framework, in order to model unreliable sensing, observations are defined as arbitrary relations from the domain state to finitely valued observation variables, to be intended as the "sensors" available to the plan executor.

In the concrete syntax of NPDDL, this is achieved by a parametric `:observation` construct. An observation variable V is characterized by a boolean formula over V and the domain fluents. The intuition is that the formula defines the relation between the value of the observation variable, and the values of the (possibly unobservable) domain fluents. The formula is arbitrary, with the only (semantic) constraint that every domain state must correspond to at least one value of V. For the sake of simplicity, a domain fluent f can be declared `:observable`. This amounts to introducing a new observation, the value of which faithfully reports the values of f.

Example The sensing described in the example can be modeled as follows. Notice the way NPDDL allows describing a `paper_in_printer` sensor which does not provide information (is undefined) unless the robot is appropriately placed at the printer.

```
(:observable (paper_banner) - room_number)

(:observation (robot_at_printer) - :boolean
```

```
   (iff (robot_at_printer) (= (robot_room) 0)))

(:observation (paper_in_printer) - :boolean
   (and
     (imply (paper_in_printer)
          (or (> (robot_room) 0)
              (paper_at_printer)))
     (imply (not (paper_in_printer))
          (or (> (robot_room) 0)
              (not (paper_at_printer))))))
```

Problems in NPDDL

When dealing with nondeterminism and partial observability, a plan may admit a set of different executions, possibly resulting in different final states. This makes it necessary to specify whether every execution is required to be successful or not, and whether the possibility of having infinite executions is accepted. These natural requirements are captured by specifying that plans have to be "weak", "strong", "strong cyclic" (`:weakgoal`,`:stronggoal`, `:strongcyclicgoal` resp). Intuitively, a plan is "weak" if it admits at least one successful finite execution; it is "strong" if every admissible execution is finite and successful; it is "strong cyclic" if every (possibly infinite) execution always admits a possibility of succeeding finitely. See (Cimatti *et al.* 2001b) for formal definitions.

Moreover, several problems, e.g. classical planning problems or conformant planning problems, define implicit assumptions about the observability of the domain. To support users in naturally specifying these, NPDDL introduces an optional `:observability` keyword. This allows users to specify that a plan must be synthesized under e.g. full observability assumptions, regardless of the observations specified in the domain. Default observability is assumed to be `:full`, to retain backward compatibility with PDDL.

Example A possible expression of desired goal consists in the following:

```
(:observability :partial)
(:strongcyclicgoal
   (and
        (not (arm_busy))
        (not (papers_around))
        (not (paper_at_printer))))
```

Plans in NPDDL

In classical planning, a plan is simply a set of partially ordered actions. Nondeterminism entails the necessity for iterative structures; partial observability requires the introduction of branching in the plan language. NPDDL supports a high-level plan language. A plan may have local, internal variables, different from the ones of the domain, containing information to encode, for instance, the progress of the plan. We call such variables "plan variables"; plan variables are in a finite number, and feature finite ranges. The basic steps of the language, differently from what happens in a simple programming language, must take into account the issue of execution, with the delivery of actions to the domain actuators. The basic construct is `evolve`, with syntax

`(evolve <assignment>+ <action-call>)`

that specifies a set of assignments to plan variables, followed by an action. Unless assigned, plan variables retain their previous value. The `action` construct, with syntax

`(action <action-call>)`

is a variation on `evolve` that does not alter the plan state. The `done` construct indicates that the plan has to be intended as terminated; no specification is given upon which actions are produced by a plan after `(done)` is executed.

The plan language also provides a number of imperative-style constructs:

- *Sequencing Commands*: `(sequence <command>+)` corresponds to sequentially executing all the commands in the specified order.
- *Branching Commands*: `(if <condition> <plan_then> <plan_else>)`, with its usual semantic.
- *Iterative Commands*: `(while <condition> <plan>)` and `(repeat <plan>)`. The `while` semantics is standard; `repeat` causes an endless looping execution.
- *Labeling and Jumping Commands*. Labeling may refer to a command or to a given point inside a command; this is reflected by two alternative syntaxes, namely `(label <name> <command>)` and `(label <name>)`. The `(goto <name>)` construct has the usual semantics.

Example Consider the following course of actions, solving the example problem. "Loop forever, acting as follows: first the robot re-positions at the printer's room, then, if some paper is there, it picks it up, and delivers it to the proper room; if no paper is there, the plan terminates." This is represented by the following plan. Once picked a paper, to deliver it to the proper room, the robot traverses x rooms, where x is the paper banner content. Unless the printing queue is initially empty, the reference problem may require an infinitely executing plan, e.g. one like the following.

```
(define (plan deliver_paper)
(:domain paper_delivery)
(:problem continuous_delivery)
(:planvars known_room - room_number)
(:init (= (known_room) 1))
(:body
 (repeat
   (sequence
     (while (not (robot_at_printer))
       (action (move_left)))
     (if (not (paper_in_printer))
        (done)
        (sequence
          (evolve
            (assign (known_room) 0)
            (action (pick_paper)))
          (while (< (known_room) (paper_banner))
            (evolve
               (assign (known_room)
                       (+ (known_room) 1))
               (action (go_right))))
          (action (leave_paper)))))))
```

A plan can be converted into a Finite-State Machine. The conversion procedure, that results in a definition of the tran-

sition relation of the FSM, is the following:

- Sequencing, branching and iterative constructs are removed by introducing label/jump pairs, and splitting the plan into a set of labeled plans whose bodies only includes `goto` and basic commands.
- `goto` occurrences are eliminated by inlining the bodies of the labeled plan they refer to.
- The set of labeled plans is translated into an equivalent finite state machine, whose state space is augmented with a variable ranging over the set of all used labels; this additional variable serves to model a "plan program counter" to appropriately sequentialize the executions of labeled plans.

Related Work and Discussion

Several works present PDDL-like languages to deal with some aspects related to nondeterminism (Bonet & Geffner 2000a; Kabanza 1999; Smith & Weld 1998; Weld, Anderson, & Smith 1998). The language of GPT (Bonet & Geffner 2000a) adopts a PDDL-like syntax, extended with a quantitative model with probabilities and rewards. Such issues were purposefully avoided in NPDDL, where a qualitative model is assumed. If we compare the language of GPT and NPDDL disregarding quantitative issues, however, NPDDL exhibits an improved flexibility in expressing nondeterminism. In particular, the GPT language does not allow for a general description of nondeterminism in the form of nested/conjoined nondeterministic assignments. The result raises an issue of compactness in the domain/problem descriptions, as an explicit enumeration of all of the possible initial states (or action outcomes) is in order. Our clean treatment of nondeterminism, on the other hand, opens up the possibility of extending NPDDL with quantitative aspects. In fact, the characterization in terms of sets of assignments presented in this paper seems to be amenable to a straightforward extension, that results in a quantitative model without suffering from the problems related to the explicit enumeration of initial states and transitions. The extension is the object of an ongoing investigation. As far as observability is concerned, the GPT language is based on "knowledge-gathering actions", making it difficult to express automatic sensing, where observations can be automatically gathered at each "cycle". Although NPDDL focuses on automatic sensing, action-dependent sensing can be easily encoded by suitably defining observed values as undetermined unless a given action is formerly executed.

The PDDL-like languages used in CGP (Smith & Weld 1998) and SGP (Weld, Anderson, & Smith 1998) allow the description of incompletely specified initial conditions, but are limited to deterministic actions. Sensing in SGP is restricted to knowledge-acquisition actions, and the underlying model is very limited.

SimPlan (Kabanza 1999) is able to deal with initial uncertainty and nondeterminism, but does not admit partial observability. The ADL/STRIPS interface of SimPlan allows for a very limited forms of nondeterminism, resulting from the combinations of controllable actions with environment actions. A more accurate handling of nondeterminism is only possible using SimPlan's custom interface language to specify a domain transition relation, but the specification of complex domains turns out to be very cumbersome. NADL (Jensen & Veloso 2000) is a language that allows for expressing qualitative models of nondeterministic domains in a multiagent setting. Exogenous actions are supported, as well as a limited form of concurrency for actions that involves a notion of resource constraint.

Both SimPlan and NADL try to address the fact that often a nondeterministic domain is naturally described as the composition of a (possibly deterministic) plant with some form of nondeterminism due, for instance, to uncontrollable agents in the domain. For instance, the reference example could be extended with the notion of doors between rooms, that are being opened/closed e.g. possibly by non controllable agents (this example is remarkably similar to the kid doors in (Kabanza, Barbeau, & St-Denis 1997)). The problem with this kind of dynamics is in the fact that it is not naturally described in the style of PDDL, where operators describe the possible tasks of the agent the activity of which is being planned for. Every action should have this environment dynamics in its effects, e.g. doors being nondeterministically open or closed. In fact, even if a `do-nothing` action is performed, the effect of doors possibly being open or closed by other agents should be taken into account. Similar examples arises when a system has a certain dynamics (e.g. a timer being set in a certain situation and expiring after N time units). In a PDDL-style characterization, each action should have the effect of decrementing the timer value, and possibly making the "expired" predicate become true. A design choice underlying NPDDL was to retain the operator-based description of actions. Although we believe that the current expressive power is enough to characterize many interesting domains, whether the modeling is *natural* this is an open issue. A principled analysis is in order for the extension of NPDDL to deal with this important issue.

Somewhat less related are high level action languages such as AR (Giunchiglia, Kartha, & Lifschitz 1997) and C (Giunchiglia & Lifschitz 1998), that deal with the problem of providing expressive languages for domain description in presence of nondeterminism. AR deals with ramification constraints, can represent different forms of nondeterminism, and its semantics is defined in terms of a minimization procedure to solve the frame and the ramification problem. C is an action language based on causal explanation, allowing for nondeterminism and concurrency. In both cases, the underlying semantics and the representation style are very far from PDDL's, and observability issues are not taken into account.

Finally, (Petrick & Bacchus 2002) discusses a different approach to nondeterminism, based on encoding actions at the knowledge level, and exploiting "knowledge databases" to trigger knowledge-level derivations. Knowledge-level reasoning is not explicitly addressed by NPDDL, although for several domains/problems suitable knowledge-level abstractions are possible within standard PDDL.

Results

NPDDL is the input language of the MBP planner. MBP integrates plan synthesis, plan validation and plan simulation; MBP handles plans in the format described in this paper. MBP applies some restrictions to the language; in particular, observations are currently to boolean variables and have a fixed structure; the `number` type is not allowed, and union types are not implemented. Several NPDDL examples have been designed, e.g. the "maze" benchmark for partial observability (Bertoli, Cimatti, & Roveri 2001), and domains taken from the Power Supply Restoration (Bertoli *et al.* 2002a). Moreover, MBP supports two extended goal languages for expressing maintenance goals, safety goals, liveness goals. Most of these classes identify constraints over the plan execution, e.g. a certain property must hold throughout the whole execution, or it must never hold throughout the execution, and so on. As such, they can be captured by a temporal logic that deals with nondeterminism, such as CTL (Emerson 1990). More recently, the necessity of expressing intentionality in the goals to achieve high-quality plans has been highlighted ((Pistore, Bettin, & Traverso 2001)). MBP allows for CTL and for EaGLe ((Dal Lago, Pistore, & Traverso 2002)), an extended temporal language to express intentionality into goals.

Example We consider a variation of the original goal, adding the requirement that, everytime a robot has a paper and is in the right room to deliver it, it should leave it with no delay. This can be modeled as a CTL goal:

```
(:ctlgoal
     (and
      (ag
       (imply (= (robot_paper_x) (robot_x))
              (not (next (arm_busy)))))
      (aw
         (and
            (not (papers_around))
            (ef (not (paper_at_printer))))
         (not (paper_at_printer)))))
```

Conclusions

In this paper we have presented NPDDL, an extension to PDDL for planning in nondeterministic domains. NPDDL retains all the features of PDDL, and allows for a compact characterization of incompletely specified initial conditions and nondeterministic action effects. NPDDL provides a clear solution to the issues related to the interaction between closed world assumption and initial condition, and nondeterministic action effects and law of inertia. The model for observations underlying NPDDL allows to characterize and combine action-dependent and automatic sensing. In addition, NPDDL allows to model complex plan structures and temporally extended goals. NPDDL is implemented in MBP, has been used to model complex real-world problems, and will hopefully prove to be an adequate starting point for a standard extension to PDDL. In this sense, we are studying ways to improve its flexibility, e.g. by suitably extending the PDDL `:requirements` to allow certain features (e.g. support for temporal goals) without forcibly requiring them.

Acknowledgements

The starting point of this work is a number of discussions that were in August 2001 at IJCAI, including Enrico Giunchiglia, David Smith, Blai Bonet, Keith Golden, Jussi Rintanen. We thank Marco Roveri and Paolo Traverso for many fruitful discussions on the topic.

References

Bacchus, F. 2000. AIPS-2000 planning systems competition. On-line report at http://www.cs.toronto.edu/aips2000/.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001. MBP: a Model Based Planner. In *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*.

Bertoli, P.; Cimatti, A.; J.Slaney; and S.Thiebaux. 2002a. Solving power supply restoration problems with planning via model checking. In *Proceedings of ECAI'02*.

Bertoli, P.; Cimatti, A.; M.Pistore; and Traverso, P. 2002b. Plan validation for extended goals under partial observability (preliminary report). In *Proc. of AIPS-02 Workshop on Planning via Model Checking*.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proceedings of IJ-CAI'01*.

Bonet, B., and Geffner, H. 2000a. Planning with Incomplete Information as Heuristic Search in Belief Space. In Chien, S.; Kambhampati, S.; and Knoblock, C., eds., 5th *International Conference on Artificial Intelligence Planning and Scheduling*, 52–61. AAAI-Press.

Bonet, B., and Geffner, H. 2000b. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS 2000*, 52–61.

C. Boutilier, T. D., and Koenig, S., eds. 2002. *"Artificial Intelligence Journal, Special Issue on Planning with Uncertainty and Incomplete Information"*. Elsevier.

Cassandra, A.; Kaelbling, L.; and Littman, M. 1994. Acting optimally in partially observable stochastic domains. In *Proc. of AAAI-94*. AAAI-Press.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2001. Improvements to sat-based conformant planning. In *Proc. of 6th European Conference on Planning (ECP-01)*.

Cimatti, A.; Giunchiglia, E.; Geffner, H.; and Rintanen, J., eds. 2001a. *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001b. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. Technical report, IRST, Trento, Italy.

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a Language for Extended Goals. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI02)*.

Doherty, P., and Kvarnström, J. 2001. TALplanner: A temporal logic-based planner. *The AI Magazine* 22(1):95–102.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier. chapter 16, 995–1072.

Fox, M., and Long, D. 2001. PDDL2.1: an extension to pddl for modelling time and metric resources.

Fox, M., and Long, D. 2002. The third international planning competition: Temporal and metric planning. In *Proc. of Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS02)*.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl—the planning domain definition language.

Giunchiglia, E., and Lifschitz, V. 1998. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, 623–630.

Giunchiglia, E.; Kartha, G. N.; and Lifschitz, V. 1997. Representing action: Indeterminacy and ramifications. *Artificial Intelligence* 95(2):409–438.

Goldman, R., and Boddy, M. 1996. Expressive Planning and Explicit Knowledge. In *Proc. of AIPS-96*.

Jensen, R., and Veloso, M. 2000. OBDD-based Universal Planning for Synchronized Agents in Non-Deterministic Domains. *Journal of Artificial Intelligence Research* 13:189–226.

Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–113.

Kabanza, F. 1999. Simplan - theoretical background.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nourbakhsh, I., and Genesereth, M. 1996. Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots Journal* 3(1):49–67.

Petrick, R., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS'02, Toulouse, France*, 212–221.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic techniques for planning with extended goals in non-deterministic domains.

Pryor, L., and Collins, G. 1996. Planning for Contingency: a Decision Based Approach. *J. of Artificial Intelligence Research* 4:81–120.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research* 10:323–352.

Rintanen, J. 2002. Backward plan construction for planning with partial observability. In M. Ghallab, J. H., and Traverso, P., eds., *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS02)*.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 889–896. Menlo Park: AAAI Press.

Tovey, C., and Koenig, S. 2000. Gridworlds as testbeds for planning with incomplete information. In *Proceedings of the National Conference on Artificial Intelligence*.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 897–904. Menlo Park: AAAI Press.

PDDL Conditional effects rewriting

In the following we outline a procedure for rewriting a generic effect (with nested conditional effects) into an effect featuring only non-nested conditional effects. The procedure is based on a set of rewrite rules, and introduces, in the intermediate steps, a `SWITCH` construct that generalizes the `when` dealing with a set of mutually exclusive `CASE`s. The result of the procedure is a PDDL term, where the intermediate `SWITCH` and `CASE` have been eliminated. Rule 1 is used as the first rewriting step to translate `when`s into binary `SWITCH`es. Rules 2, 3 are used to bring switches to top-level, eliminating nesting within other switches or into PDDL constructs. Finally, rule 4 is used to transform the top-level switch into a set of `when`s, making use of the fact that the switch features mutually exclusive conditions covering all CASEs.

Rule 1: `when` **elimination**

```
(when <condition> <effect>)
```

becomes

```
(SWITCH
   (CASE <condition> <effect>)
   (CASE (not <condition>) true))
```

Rule 2

```
(and
    (SWITCH
       (CASE <condition^1_1> <effect^1_1>)
       ...
       (CASE <condition^1_n1> <effect^1_n1>))
    ...
    (SWITCH
       (CASE <condition^m_1> <effect^m_1>)
       ...
       (CASE <condition^m_nm> <effect^m_nm>)))
```

becomes

```
(SWITCH
   (CASE
      (and <condition^1_1> ...  <condition^m_1>)
      (and <effect^1_1> ...  <effect^m_1>))
   ...
   (CASE
      (and <condition^1_n1> ...  <condition^m_nm>)
      (and <effect^1_n1> ...  <effect^m_nm>)))
```

Rule 3

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^(i-1)> <effect^(i-1)>)
   (CASE <condition^i>
      (SWITCH
         (CASE <innercondition^1> <innereffect^1>)
         ...
         (CASE <innercondition^m> <innereffect^m>)))
   (CASE <condition^(i+1)> <effect^(i+1)>)
   ...
   (CASE <condition^n> <effect^n>))
```

becomes

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^(i-1)> <effect^(i-1)>)
   (CASE (and <condition^i> <innercondition^1>)
         <innereffect^1>)
   ...
   (CASE (and <condition^i> <innercondition^m>)
         <innereffect^m>)
   (CASE <condition^(i+1)> <effect^(i+1)>)
   ...
   (CASE <condition^n> <effect^n>))
```

Rule 4: `when` **introduction**

```
(SWITCH
   (CASE <condition^1> <effect^1>)
   ...
   (CASE <condition^m> <effect^m>))
```

becomes

```
(and
   (when <condition^1> <effect^1>)
   ...
   (when <condition^m> <effect^m>))
```

NPDDL Conditional Effects Rewriting

The following rule is used, in conjunction to rules 2 and 3, to bring switches to top-level, in the case of nondeterminism.

```
(oneof
      (SWITCH
         (CASE <condition^1_1> <effect^1_1>)
         ...
         (CASE <condition^1_n1> <effect^1_n1>))
      ...
      (SWITCH
         (CASE <condition^m_1> <effect^m_1>)
         ...
         (CASE <condition^m_nm> <effect^m_nm>)))
```

becomes

```
(SWITCH
   (CASE
      (and <condition^1_1> ...  <condition^m_1>)
      (oneof <effect^1_1> ...  <effect^m_1>))
   ...
   (CASE
      (and <condition^1_n1> ...  <condition^m_nm>)
      (oneof <effect^1_n1> ...  <effect^m_nm>)))
```

The Complete NPDDL Model

```
(define (domain paper_delivery)
  (:types room_number)
  (:predicates
    (arm_busy)
    (papers_around)
    (paper_at_printer))
  (:functions
       (robot_room)   - room_number
       (paper_banner) - room_number)

  (:action move_left
   :precondition (not (= (robot_room) 0))
   :effect       (assign (robot_room) (- (robot_room) 1)))

  (:action move_right
   :precondition (not (= (robot_room) (sup room_number)))
   :effect       (assign (robot_room) (+ (robot_room) 1)))

  (:action pick_paper
   :precondition (and (paper_at_printer)
                      (not (arm_busy))
                      (= (robot_room) 0))
   :effect  (and
              (arm_busy)
              (unknown (paper_banner))
              (unknown (paper_at_printer))))

  (:action leave_paper
   :precondition (arm_busy)
   :effect  (and
              (not (arm_busy))
              (when (not (= (robot_room) (paper_banner)))
                    (papers_around)))))

(:observable (paper_banner) - room_number)

(:observation (robot_at_printer) - :boolean
   (iff (robot_at_printer) (= (robot_room) 0)))

(:observation (paper_in_printer) - :boolean
   (and
     (imply (paper_in_printer)
          (or (> (robot_room) 0)
              (paper_at_printer)))
     (imply (not (paper_in_printer))
          (or (> (robot_room) 0)
              (not (paper_at_printer))))))


(define (problem continuous_delivery)
        (:domain paper_delivery)
        (:typedef room_number - (range 0 50))
        (:init
           (unknown (robot_room))
           (unknown (paper_at_printer))
           (not (papers_around))
           (not (arm_busy)))
        (:observability :partial)
        (:strongcyclicgoal
           (and
                (not (arm_busy))
                (not (papers_around))
                (not (paper_at_printer)))))
```