NDL – A Modeling Language for Planning

Jussi Rintanen
Aalto University
Department of Computer Science

February 20, 2017

Abstract

This document describes the NDL modeling language for deterministic full information planning problems.

Contents

Foreword

The NDL language has been in development since 2014, with latest iterations in the design completed in early 2017. The design objective was to provide a more compact, expressive and clean modeling language than what was available before, especially targeting research and education uses.

An important language in AI Planning has been the PDDL language, which was first developed for the 1998 planning competition [GHK$^+$98], and which has been in its various incarnations in wide use by the research community since then. Major problems with PDDL have been its clumsy and non-uniform syntax, lack of clean semantics, and lack of features supporting powerful modeling of realistic problems, especially for temporal planning and most notably resources [BLP96, Rin15]. This has been only a limited problem to the research community, which has focused on not modeling and solving broad classes of planning problems, but with experimenting with a relatively narrow suite of standard benchmark problems.

Expressivity, features and ease of modeling is a far bigger issue for large realistic applications of AI technologies, and also in educational use. For both of these purposes a major problem has also been that de facto PDDL has only been a very limited subset of the original and broader PDDL definitions: since research prototypes have not implemented e.g. many critical data types, including integers, reals and enums, also most of the standard planning benchmarks are very clumsily expressed in terms of Boolean state variables only. This has rendered many of the more advanced features essentially non-standard. For educational and real-world modeling these deficiencies pose an unnecessary burden, and in many cases lead to inefficient and clumsy models.

1 Introduction

In classical planning, each action describes an atomic change in the system state, from the current state to the next. An action has no duration, and multiple actions cannot be involved in the transition from one state to the next. As durations do not play a role, the execution of a sequence of actions is simply a sequence of states, each of which is an assignment of values to all state variables.

A plan that is a *solution* to a problem instance is a finite sequence of actions that induces a sequence of states, the last of which satisfies the goal condition.

2 Lexical Items

Alphanumeric symbols start with a lower or upper case letter in the range "a" to "z", followed by zero or more letters, numbers 0 to 9, or the underscore symbol. The language is case-sensitive, so "x1" and "X1" are different symbols.

Integer constants are sequences of one or more number between 0 to 9, possibly prefixed with the negation sign "-".

Real constants are sequences of one or more numbers followed by a period and further 1 or more numbers, and possibly prefixed with the negation sign "-".

3 Data Types

3.1 Booleans

Boolean constant *true* and *false* are represented by symbols 1 and 0, respectively, or, alternatively, the symbols "true" and "false".

Boolean expressions evaluate to true or false, and have all the standard connectives from the propositional logic, expressed by symbols

```
&   |   not   ->   <->
```

3.2 Integers

The language supports arbitrary (unbounded) signed integers. Particular implementations might limit to some finite subset, e.g. those representable in 32 or 64 bits.

Integer expressions consists of integer constants, integer-valued state variables, integer-valued action parameters, and the following arithmetic operations.

```
+   -   *
```

Parentheses are used in the standard way to indicate how operators in a multi-operator expression are associated. Operator precedence is standard: multiplication binds stronger than addition and subtraction.

3.3 Reals

The language supports arbitrary real numbers. Particular implementations typically limit to some finite-precision subset, or the rationals.

Real expressions are syntactically like integer expressions, except that they allow also real constants in addition to integer constants. Real constants consists of two non-empty sequences of numbers with a decimal period . in between.

Additionally, numeric inequalities formed with numeric relational operators

```
<   >   <=   >=   =   !=
```

evaluate to Boolean values.

3.4 Enumerated Types

An enumerated type consists of a finite set of alphanumeric symbols.
The only operations on enumerated values are equality and inequality.

```
=   !=
```

4 Language Constructs

The only obligatory parts of an NDL specification are the state variable, initial state and goal declarations. All sensible specifications also have action declarations.

4.1 Type Declarations

Type declarations associate a name with a data type that is used as a parameter in a state variable declaration or as the value of state variables.

`enum`	set of enumerated values
`int`	integer
`[n..m]`	integer range $\{i, i+1, \ldots, m-1, m\}$
`real`	reals/rationals
`bool`	Booleans

Sets of enumerated values may be expressed by listing a set of alphanumeric constants, for example

```
type objectstate = { ready, active, inactive };
```

or by set-theoretic expressions involving the operations of intersection, union and difference, respectively expressed with the following symbols.

```
^   U   \
```

For example, we might define the following.

```
type objectstate = { ready, active, inactive };
type activestates = objectstate \ { inactive };
```

4.2 State Variable Declarations

4.3 Formulas and Numeric Expressions

Goals and preconditions of actions are Boolean formulas, formed with the Boolean connectives. The atomic formulas are Boolean state variables as well as Boolean-valued numeric expressions (equality, inequality), or comparisons of enumerated values (equality, inequality.)
Boolean state variables can be used as Boolean-valued expressions as such. Hence *workday(monday)* is the same as *workday(monday) = 1*.

4.4 Action Declarations

Action declarations represent one or more action instances (ground actions). The action declarations consists of

1. name of the action,
2. parameters,
3. precondition, and
4. effects.

All components are obligatory, but the parameter list may be empty, and the precondition can be the constant *true* to indicate an action that can always be taken.

The instances of an action are obtained by generating all possible combinations of the allowed values of the parameters, and instantiating the precondition-effects part accordingly.

An example of an action is the following.

```
action fueltank( v : vehicle, fuel : [1..10] )
fuellevel[v] + fuel < capacity[v]
=>
fuellevel[v] := fuellevel[v] + fuel;
```

If there are 5 objects in the type *vehicle*, then this action has a total of $5 \times 10 = 50$ instances.

4.4.1 Parameters

The parameters of an action can be of the following two finite types.

- enums
- integer intervals

These parameters are primarily used in naming state variables. For example, if two parameters to an action are $x : [0, 5]$ and $y : [0, 5]$, we can refer to state variables `whatshere[x,y]` which have been declared by

```
type content = { a, b, c, empty };
type coord = [0,5];
decl whatshere[coord,coord] : content;
```

Since the number of combinations of parameters values determines the number of ground instances an action has, it is important that the number of possible parameter values is not too high.

4.4.2 State Variable References

State variables an action refers to in its precondition and effects may be non-schematic state variables as well as schematic variables of the form $p(t_1, \ldots, t_n)$, where t_i are *terms*. The terms may be enums and integer expressions with values from the range indicated in the state variable declaration. For example, if an action has parameters x and y, we could refer to a state variable by

```
whatshere[x+1,y-1]
```

So terms may be arithmetic expressions with references to integer parameters and numeric constants, but **no integer-valued state variables**. This restriction is critical, as otherwise the parameters of the action could not be handled by grounding.

4.4.3 Preconditions

Preconditions are arbitrary Boolean-valued expressions (formulas).

4.4.4 Effects

Atomic effects are *assignments* of values to state variables. They are written similarly to conventional programming languages, with the name of the state variable, the assignment symbol ":=", followed by an expression of the same type as the state variable.

Assignments *b := 1* and *b := 0* to Boolean values *b* can be written simply as *b* and *not b*, respectively.

If *be* is a Boolean-valued expression (formula), e_1 and e_2 are effects, v is a parameter variable possibly with occurrences in e_1, and ee is an enum-valued expression, then also the following are effects.

1. `if` *be* `then` e_1;
2. `if` *be* `then` e_1 `else` e_2;
3. `forall` v : *ee* e_1;

4.5 Initial States

An initial state is described by a sequence of assignments of constant values (integer, real, Boolean and enum constant values) to state variables of corresponding types. Numeric state variables have value 0 by default, if not indicated otherwise, and Boolean variables are *false* (0) by default. All enum variables must be given a value explicitly.

```
initial
  whatshere[0,0] := nothing;
  whatshere[0,1] := a;
  whatshere[1,0] := b;
  whatshere[1,1] := c;
```

4.6 Goals

The goal is Boolean formula with references to any state variables.

```
goal (whatshere[0,1] = nothing) & (whatshere[1,1] = a);
```

Index

References

[BLP96] Philippe Baptiste and Claude Le Pape. Disjunctive constraints for manufacturing scheduling: principles and extensions. *International Journal of Computer Integrated Manufacturing*, 9(4):306–310, 1996.

[GHK+98] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University, October 1998.

[Rin15] Jussi Rintanen. Models of action concurrency in temporal planning. In *IJCAI 2015, Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pages 1659–1665. AAAI Press, 2015.