NASA/TM-2006-214518

An Abstract Plan Preparation Language

Ricky W. Butler
Langley Research Center, Hampton, Virginia

César A. Muñoz
National Institute of Aerospace, Hampton, Virginia

November 2006

NASA/TM-2006-214518

An Abstract Plan Preparation Language

Ricky W. Butler
Langley Research Center, Hampton, Virginia

César A. Muñoz
National Institute of Aerospace, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

November 2006

An Abstract Plan Preparation Language

Rick W. Butler[*] César A. Muñoz[†]

November 21, 2006

Abstract

This paper presents a new planning language that is more abstract than most existing planning languages such as the Planning Domain Definition Language (PDDL) or the New Domain Description Language (NDDL). The goal of this language is to simplify the formal analysis and specification of planning problems that are intended for safety-critical applications such as power management or automated rendezvous in future manned spacecraft. The new language has been named the *Abstract Plan Preparation Language* (APPL). A translator from APPL to NDDL has been developed in support of the Spacecraft Autonomy for Vehicles and Habitats Project (SAVH) sponsored by the Explorations Technology Development Program, which is seeking to mature autonomy technology for application to the new Crew Exploration Vehicle (CEV) that will replace the Space Shuttle.

Acronyms	
AI	Artificial Intelligence
APPL	Abstract Plan Preparation Language
CEV	Crew Exploration Vehicle
NDDL	New Domain Description Language
PDDL	Planning Domain Definition Language
SAL	Symbolic Analysis Laboratory
SAVH	Spacecraft Autonomy for Vehicles and Habitats

[*]`r.w.butler@larc.nasa.gov`. NASA Langley Research Center.

[†]`munoz@nianet.org`. National Institute of Aerospace.

Contents

1 Introduction

The New Domain Description Language (NDDL) is a powerful planning language developed at NASA Ames [2] which has evolved from PDDL[3] It is the planning language of the Extensible Universal Remote Operations Architecture (EUROPA 2). EUROPA 2 is a component-based software library for constructing highly-tailored, domain-specific planners. The authors write "Our goal in developing EUROPA 2 is to provide a fast, flexible, extensible, reusable technology platform for building planning and scheduling applications for space exploration [1]." Its predecessor, EUROPA, was used for a variety of NASA missions including MAPGEN (Mars rovers) and HSTS (Deep Space 1). The EUROPA planner seeks to solve both the planning (i.e., sequencing of actions) and scheduling (i.e., allocation of time and resources) problems at the same time. Complex real-world problems such as controlling autonomous spacecraft do not lend themselves to a decoupled solution approach. Here, one has to deal with issues such as

- operations which take time,
- operations which may be non-deterministic,
- subsystems that can experience failure,
- issues of resource consumption such as fuel or battery power,
- need to limit cost of operations,
- situations where previous choices and operations impact the choice of the next actions, and
- the initial use of instruments requires calibration.

The Abstract Plan Preparation Language (APPL) is a planning language strongly inspired by NDDL that is centered around the idea that not all constraints are alike in the specification of an AI planning problem. In particular, the temporal interval constraints that specify the actions that may occur on parallel timelines have special attributes. Most importantly, these actions must be mutually exclusive on a particular timeline. These features enable a more compact specification of actions and states that is in a manner analogous to the specification of state transition systems with special notations for temporal interval operations. All the syntactic features of APPL are fully exploited in a translator from APPL to NDDL that automatically generates the voluminous set of temporal constraints required in a NDDL specification.

Because the syntax and semantics of APPL are simpler than those of NDDL, APPL is more suitable for formal verification, static analysis, and automated test generation. Indeed, we are in the process of developing a translator from APPL to SRI's Symbolic Analysis Laboratory (SAL) that will enable the application of safety analysis techniques to plans written in APPL.[1] That work will be reported in a subsequent paper.

[1]This is being done in collaboration with SRI international, the developers of the SAL model checker. Many different types of verifications and analyzes are possible here. John Rushby presented a number of these in a recent SAVH V&V workshop, May 2006.

The rest of this paper is organized as follows. Section 2 presents the APPL language by way of example using the Simple Sample Application (The Planetary Rover) tutorial problem that comes with the EUROPA 2 distribution. A formal description of the the language is given in Section 3. Section 4 describes the translator from APPL to NDDL. Finally, Section 5 proposes extensions and enhancements to the APPL language. Examples in APPL and their corresponding NDDL generated code are included in the appendices.

2 A Short Description of the Simple Rover Problem

The Simple Rover problem is defined in a tutorial entitled the *Simple Sample Application (The Planetary Rover)* provided with the EUROPA 2 planning system developed at NASA Ames. This problem describes a planetary rover that has four main components: a navigator, an instrument controller, a command interface, and a battery. The navigator controls the rover's movement; the instrument controller manages a scientific instrument; the command interface manages instructions from the scientists controlling the rover; and the battery provides the power needed to operate the rover.

Associated with each of these components is a timeline of actions (or a function that maps times into actions) that must be scheduled by an AI planning system such as EUROPA.[2] Since only one of these actions can be active on a single timeline at any given time, they are mutually exclusive. Although there are several other constraints that govern the scheduling of these actions, the mutual exclusive constraint is fundamental to a planning system. Furthermore, these actions are adjacent to each other. Therefore the allowed sequence of actions can be specified like a state machine. This is illustrated graphically in Figure 1. In APPL, the valid sequence of actions are described using a notation that essentially elaborates the transition matrix of the state machine. For example the allowed transitions of the navigator timeline component is described as follows:

```
At(x) -> Going(x,y) -> At(y)
```

All the components of the Simple Rover example are declared in a module enclosed within the keywords `PLAN SimpleRover` and `END SimpleRover`.

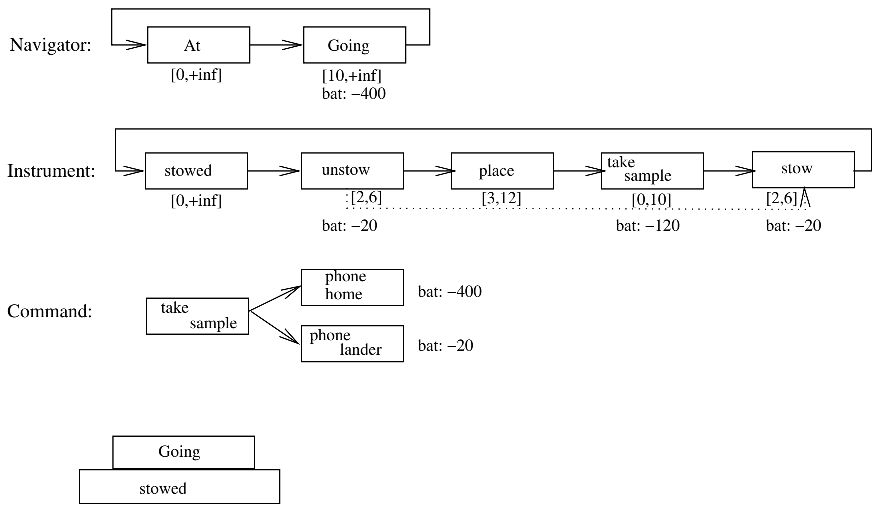
2.1 Simple Rover Types

The Simple Rover problem requires the definition of some types such as `Location` and `Paths`. These are defined in APPL as follows:

```
TYPE Location(name:string; x,y:int)

TYPE Path(name:string; from,to:Location; cost:float)
```

The keyword `TYPE` appears first, followed by the name of the type, e.g., `Location` and `Path`, and a list of typed attributes, e.g., `name`, `x`, and `x` for `Location`, and `name`, `from`, `to`, and `cost` for `Path`.

[2]In NDDL, these *actions* are called *predicates*.



Initial State: At lander with instrument stowed, battery = 1000

Goal: Take sample at rock4

Figure 1: Graphical Specification of Simple Rover Problem

2.2 Simple Rover Timelines

The four timeline components of the rover, e.g., `Navigator`, `Instrument`, `Command`, and `Battery`, are defined using a timeline section. The section begins with the keyword `TIMELINE` followed by the name of the timeline and a list of typed attributes. The keyword `ACTIONS` initiates the delineation of the set of mutually exclusive actions that can be scheduled on a timeline. The keyword `TRANSITIONS` initiates an optional section where the valid sequence of actions are defined. The keyword `END` followed by the name of the timeline closes this section.

In the case of the timeline `Battery`, the list of attributes consists of `initial_charge`, `load_level_min`, and `load_level_max`, all of them of type `float`. There is only one action on this timeline that records the changes on the battery load.

```
TIMELINE Battery(initial_charge, load_level_min, load_level_max: float)
ACTIONS

  Change(quantity:float)

END Battery
```

The timeline `Navigator` has two actions: `At` and `Going`, which correspond, respectively, to the rover stopped at a location and going from one location to another.

```
TIMELINE Navigator
```

```
ACTIONS

  At(location:Location)

  Going(from,to: Location)
    WITH from != to;
         let path: Path(_,from,to,cost) in
         starts Battery.Change(cost)

TRANSITIONS

     At(x) -> Going(x,y) -> At(y)

END Navigator
```

The `At` action has one parameter `location` which is defined using the `Location` component defined previously. The `Going` action has two parameters: `from` and `to`, both of type `Location` as well. The action `Going` has associated a constraint, which follows the keyword `WITH`. The first part of the constraint states that the parameters `from` and `to` are different. The second part of the constraint is more involved. It specifies a path between the locations `from` and `to` and that the cost of taking this path has to be recorded in the timeline of some `Battery`. Since in the Simple Rover example there is only one battery, there is no need to keep track of which battery is used. The operator `starts` is the basic interval operator with the same name in NDDL. The complete list of Allen's interval operators is: `contains`, `contained_by`, `before`, `meets`, `met_by`, `overlaps`, `starts`, `equals`, and `ends`.

The constraint in the specification of the action `Going` illustrates an important syntactic feature of APPL called *parameter matching*. Parameter matching allows for implicit declaration of variables when they first occur as arguments of actions or compound types. For instance, the variable `cost` is implicitly declared to be of type `float` as required by `Path`. Subsequent occurrences of the same name refers to the same variable. In NDDL, this feature will generate an explicit equality between the fourth argument of `Path` and the only argument of the action `Change` in the timeline `Battery`. The symbol "`_`" stands for an unspecified value and never imposes any constraints.

The final part of the specification of the timeline `Navigator` (i.e. the `TRANSITIONS` section), delineates the allowed set of action transitions. The `->` syntax is used to enumerate all possible actions that can follow an action. If an action can be followed by several different actions, a special syntax is provided to simplify the specification. This is illustrated subsequently. Parameter matching is also used in the declaration of transitions: parameters with the same name must be equal. For instance, the first parameter of `Going` must be equal to the parameter of the preceding `At`. Also the `Going` action must be followed by an `At` action where its parameter is the same as the second parameter of `Going.` These are used by the APPL to NDDL translator to construct the explicit constraints.

The timeline `Instrument` is defined as follows:

```
TIMELINE Instrument
```

```
ACTIONS

  TakeSample(rock:Location) : [0,10]
    WITH starts Battery.Change(-120);
         contained_by Navigator.At(rock)

  Place(rock:Location) : [3,12]
    WITH starts Battery.Change(-20);
         contained_by Navigator.At(rock)

  Stow : [2,6]
    WITH contained_by Navigator.At

  Unstow : [2,6]
    WITH contained_by Navigator.At

  Stowed

TRANSITIONS

    Stowed -> Unstow -> Place(r) -> TakeSample(r) -> Stow -> Stowed

END Instrument
```

Five actions are defined: `TakeSample`, `Place`, `Stowed`, `Unstow`, and `Stowed`. These correspond to the actions that the rover's instrument can sequence through. The interval that follows the declaration of an action specifies a time constraint on the duration of the action. For example, the action `Unstow` must have a duration between 2 and 6 time units. In the specification of a time interval, the symbol "`_`" is used to denote minus infinity, if it appears in the lower bound, or plus infinity, if it appears as the upper bound.

The timeline `Command` is defined as follows:

```
TIMELINE Commands
ACTIONS

  Idle

  TakeSample(rock:Location): [20,25]
    WITH contains Instrument.TakeSample(rock)

  PhoneHome
    WITH starts Battery.Change(-600)

  PhoneLander
    WITH starts Battery.Change(-20)
```

```
TRANSITIONS

  Idle -> TakeSample -> (PhoneHome | PhoneLander)

END Commands
```

Four different actions are defined: `Idle`, `TakeSample`, `PhoneHome`, and `PhoneLander`. The action `Idle` has no constraints associated with it. The action `TakeSample` must completely contain the action `TakeSample` defined in the instrument timeline. The actions `PhoneHome` and `PhoneLander` record their power consumption in the battery timeline. The construct "`|`" in the transitions section indicates that either of the actions `PhoneHome` and `PhoneLander` may follow the action `TakeSample`.

2.3 The Rover

The rover itself is defined as a type whose attributes are the battery, the navigator, the instrument, and the commands. We also define specific instances of locations, paths, battery, navigator, instrument, commands, and rover.

```
TYPE Rover(battery:Battery;
           navigator:Navigator;
           instrument:Instrument;
           command:Commands)

VARIABLES
  lander: Location("Lander", 0, 0)
  rock1 : Location("ROCK1", 9, 9)
  rock2 : Location("ROCK2", 1, 6)
  rock3 : Location("ROCK3", 4, 8)
  rock4 : Location("ROCK4", 3, 9)

  p1: Path("Very Long Way", lander, rock4, -2000.0)
  p2: Path("Moderately Long Way", lander, rock4, -1500.0)
  p3: Path("Short Cut", lander, rock4, -400.0)

  bat : Battery(10000.0, 0.0, 1000.0)
  nav : Navigator(bat)
  ins : Instrument(bat,nav)
  com : Commands(bat,nav,ins)

  spirit : Rover(bat,com,nav,ins)
```

2.4 Initial State and Goals

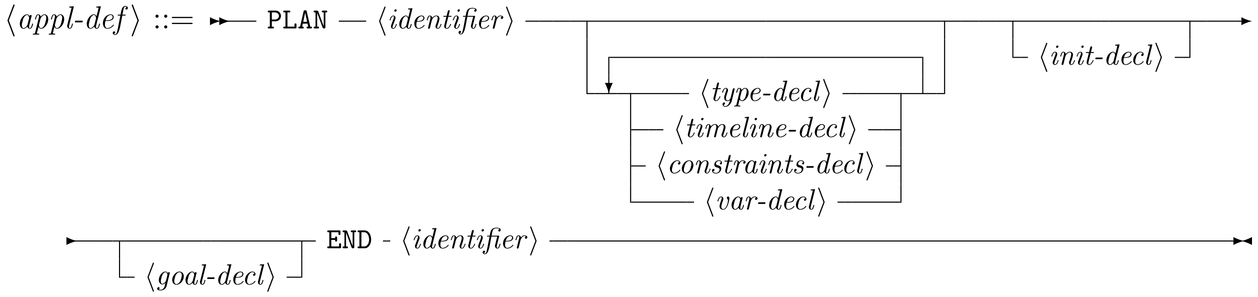
The subsection `INITIAL-STATE` specifies the initial actions of the timelines. Timelines that are not initialized may start in an arbitrary state. Finally, the desired end goal is specified in the subsection `GOALS`. In this case, we specify that the rover must reach `rock4`.

```
INITIAL-STATE

  |-> nav.At(lander)
  |-> ins.Stowed
  |-> com.Idle

GOALS
  com.TakeSample(rock4)
```

3 The Abstract Plan Preparation Language

The Abstract Planning Language (APPL) is strongly inspired on the New Domain Description Language (NDDL) and the Constraint-based Temporal Planning paradigm. Therefore, there are several conceptual similarities between NDDL and APPL. However, in contrast to NDDL, APPL does not follow the object oriented paradigm. Instead, APPL offers a more declarative approach that supports static type-checking, pattern matching, and convenient notations for temporal operators. Furthermore, in ADDL, we distinguish between compound types, which can be seen as record types, and timelines, which are special kinds of compound types that perform actions and whose temporal behavior is constrained by a set of valid transitions.

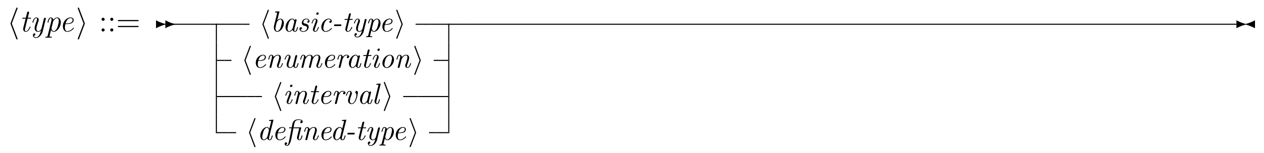
The specification of a planning problem in APPL consists of a sequence of types, timelines, constraints, variables, initial states, and goal declarations. These declarations form a module, which is enclosed between the keywords `PLAN <identifier>` and `END <identifier>`.

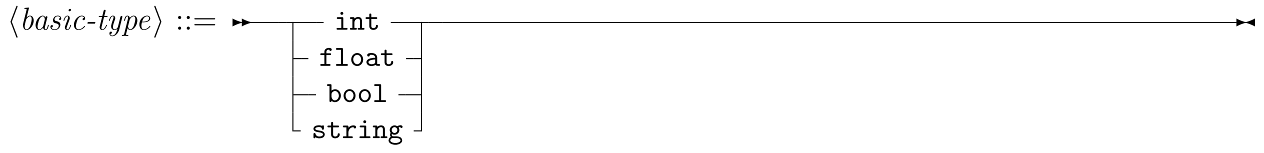


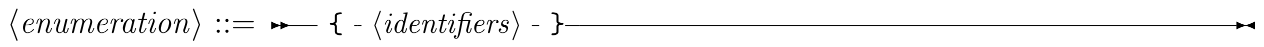
3.1 Types

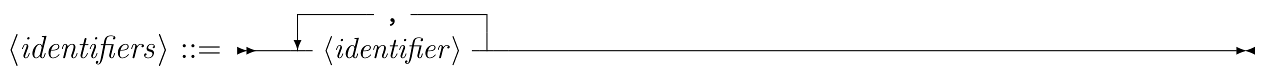
APPL is a strongly typed language, i.e., all elements of an APPL program are declared of a given type. From an operational point of view, types in APPL can be understood as containers or sets.

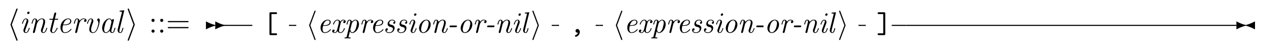
The basic types of APPL are `int`, `float`, `bool`, and `string`. Furthermore, APPL supports enumerations, intervals, and user-defined types.

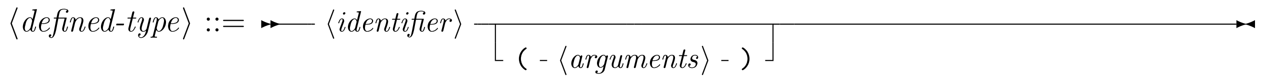


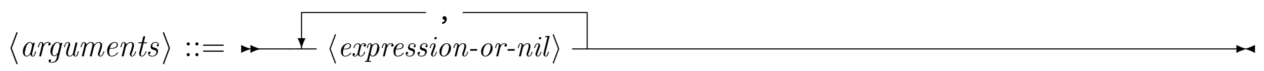




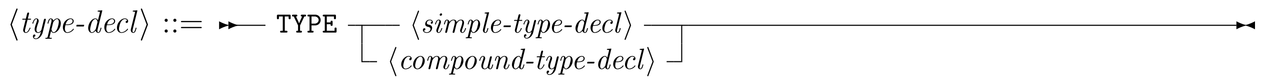


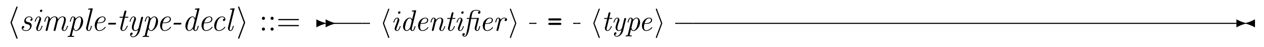


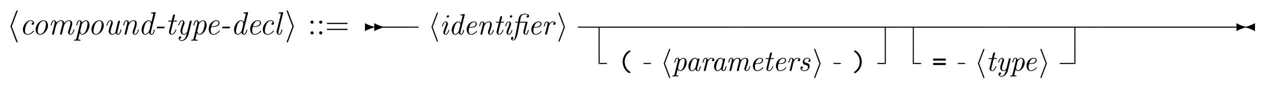


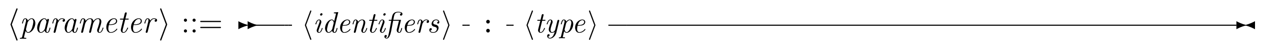


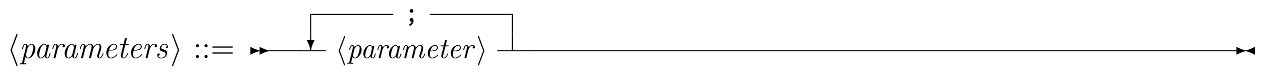
APPL enables the declaration of simple and compound types. Simple types are aliases, enumerations, and intervals. Compound types are records declared as parameterized types. The parameters of a compound type are called *attributes* and they correspond to the fields of the record. A compound type can be defined as a subtype of a previously defined (compound) type.









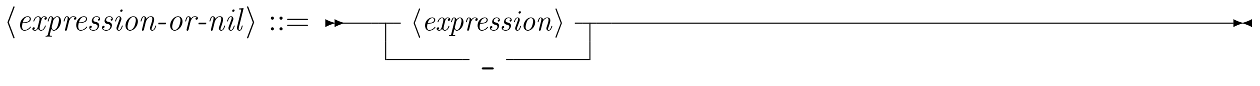


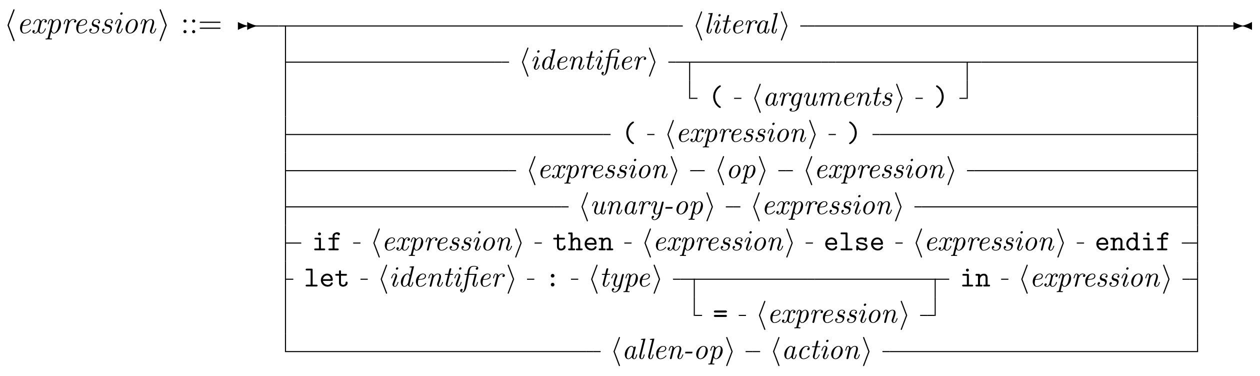
The following are valid type declarations in APPL. Note that the variables that occur in the type "`Location("Home",x,y)`" are declared as attributes of the type "`MyLocation`". Should this not be the case, an error would be reported by the ADDL compiler.

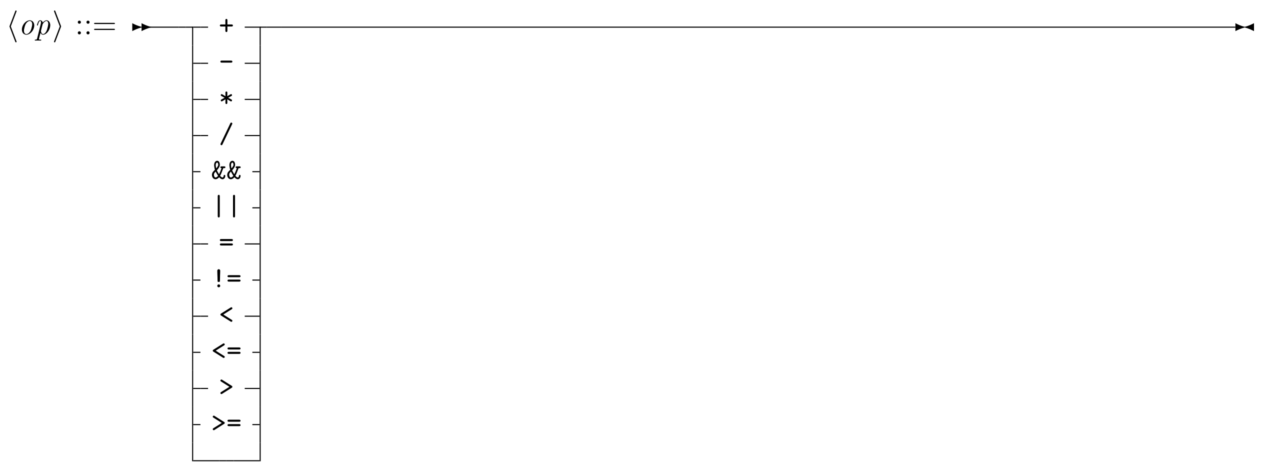
```
// An alias type
TYPE myint : int

// An enumeration type
TYPE OnOff = { ON, OFF }

// An interval type
TYPE Int_ge_1 = [1,_]

// A compound type
TYPE Location(name:string; x,y:int)

// A subtype of a compound type
TYPE MyLocation(x,y:int) = Location("Home",x,y)
```

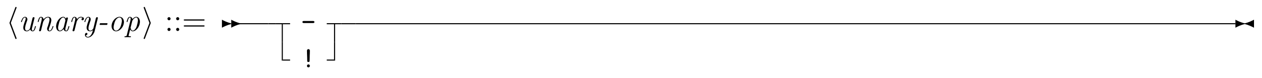
3.2 Expressions

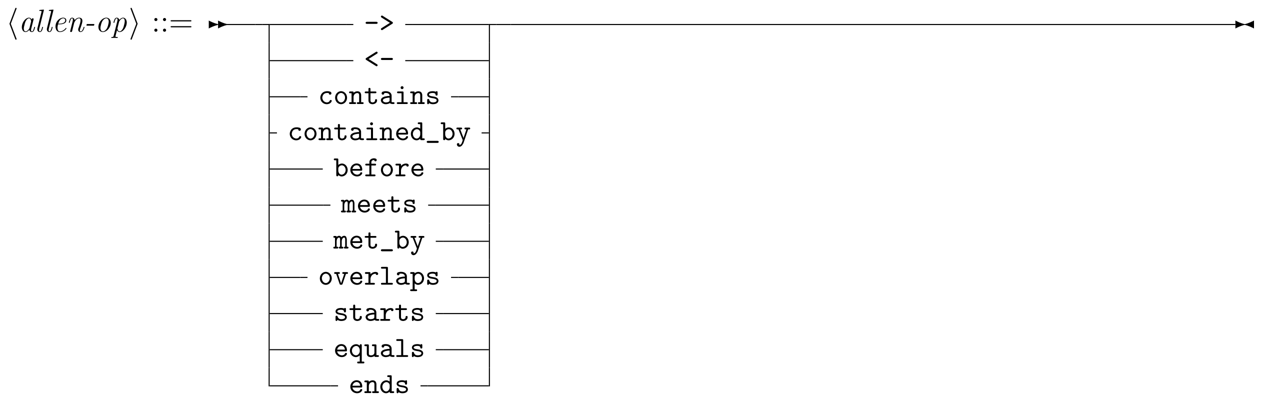
The set of expressions in APPL are drawn from NDDL. The operator "`let`" declares a variable of a certain type that satisfies a given constraint. The symbol "`_`" denotes an unspecified argument when it appears as an argument of an action or compound type. Within an interval, it stands for $\pm\infty$ depending on its position as lower bound or upper bound. The symbols "`->`" and "`<-`" are shorthands for the Allen's operators "`meets`" and "`met-by`", respectively.




⟨expression⟩ ::=
⟨literal⟩
⟨identifier⟩
(- ⟨arguments⟩ -)
(- ⟨expression⟩ -)
⟨expression⟩ – ⟨op⟩ – ⟨expression⟩
⟨unary-op⟩ – ⟨expression⟩
if - ⟨expression⟩ - then - ⟨expression⟩ - else - ⟨expression⟩ - endif
let - ⟨identifier⟩ - : - ⟨type⟩
in - ⟨expression⟩
= - ⟨expression⟩
⟨allen-op⟩ – ⟨action⟩

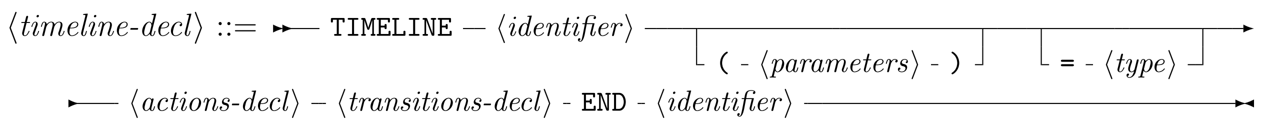


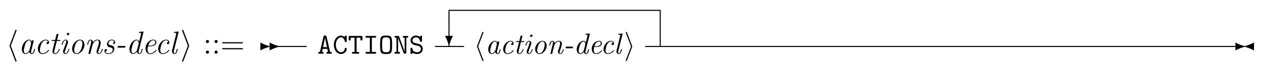


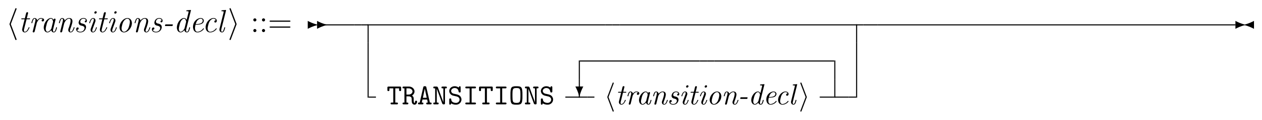


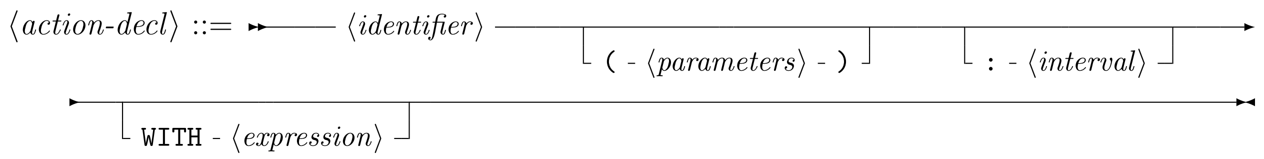
3.3 Timelines

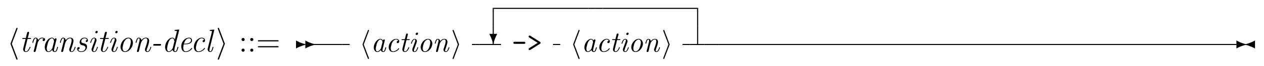
Timelines are sophisticated compound types. In addition to attributes, a timeline also specifies *actions* and *transitions*. If an action is not qualified, then it refers to the timeline in which it is being defined in.

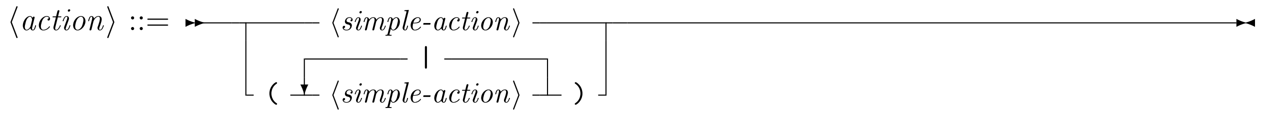


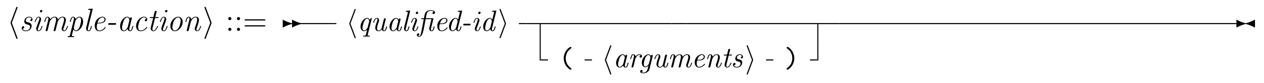


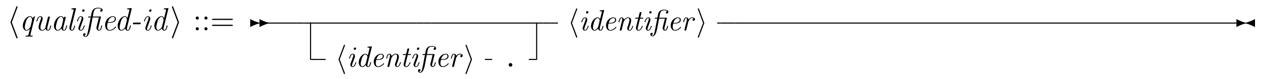












The following is an alternative declaration of the navigator timeline of the Simple Rover example. In this case, the battery that provides the power to the navigator is explicitly declared. This may be important if there is more than one battery in the system.

```
TIMELINE Navigator2(battery:Battery)

ACTIONS

  At(location:Location)

  Going(from,to: Location)
    WITH from != to;
         let path: Path(_,from,to,cost) in
         starts battery.Change(cost)

TRANSITIONS

     At(x) -> Going(x,y) -> At(y)

END Navigator2
```

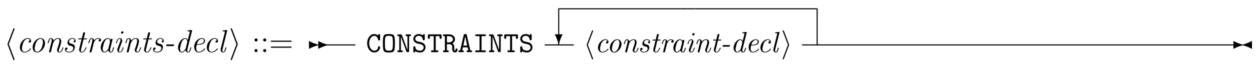
Alternative, but more detailed, specifications of the instrument timeline are shown below. The declaration of the attributes of `Instrument2` explicitly states that the battery used by the navigator and the instrument are the same. In `Instrument3`, the battery and the navigator are also explicitly declared. However, in contrast to `Instrument2`, nothing is said about the relation between the batteries used by the instrument and the navigator.

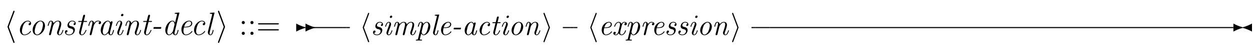
```
TIMELINE Instrument2(battery:Battery; navigator:Navigator(battery))
ACTIONS

  TakeSample(rock:Location) : [0,10]
    WITH starts battery.Change(-120);
         contained_by navigator.At(rock)

  Place(rock:Location) : [3,12]
    WITH starts battery.Change(-20);
         contained_by navigator.At(rock)

  Stow : [2,6]
    WITH contained_by navigator.At

  Unstow : [2,6]
    WITH contained_by navigator.At

  Stowed

TRANSITIONS

    Stowed -> Unstow -> Place(r) -> TakeSample(r) -> Stow -> Stowed

END Instrument2

TIMELINE Instrument3(battery:Battery; navigator:Navigator)
ACTIONS

  TakeSample(rock:Location) : [0,10]
    WITH starts battery.Change(-120);
         contained_by navigator.At(rock)

  Place(rock:Location) : [3,12]
    WITH starts battery.Change(-20);
 contained_by navigator.At(rock)

  Stow : [2,6]
    WITH contained_by navigator.At

  Unstow : [2,6]
    WITH contained_by navigator.At

  Stowed
```

```
TRANSITIONS

    Stowed -> Unstow -> Place(r) -> TakeSample(r) -> Stow -> Stowed

END Instrument3
```

3.4 Constraints Section

In the specification of a planning problem, it is often necessary to express constraints that involve more than one timeline. Although it is expected that all cross-timeline constraints could be expressed in the `ACTIONS` section, it may be convenient to gather some of these constraints in a section together. Therefore a `CONSTRAINTS` section has been added to the language:





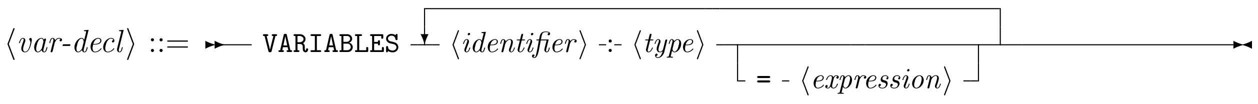
Parameter matching is used to specify parameter relationships within a constraint:

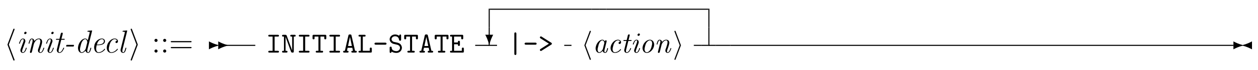
```
  CONSTRAINTS

    Commands.TakeSample(rock) contains Instrument.Take_Sample(rock)

    Instrument.Take_Sample(rock) contained_by Navigator.At(rock)

    Instrument.Place(rock) contained_by Navigator.At(rock)
```

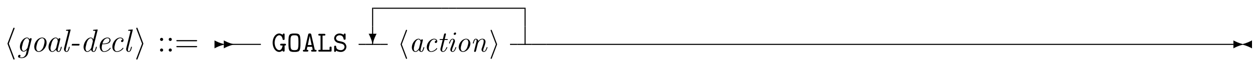
In the `CONSTRAINTS` section full name qualification of actions is required.

3.5 Initial States and Goals

Initial states and goals are declared according to the following syntax.







4 The APPL to NDDL Translator

A prototype translator has been developed that generates NDDL code from an APPL input file.[3] The following command

```
java APPL -nddl Rover.appl
```

is used to compile the file `rover.appl` into NDDL. The translator halts at the first error. Otherwise, it generates the files `Rover-model.nddl` and `Rover-initial-state.nddl`.

To illustrate the power of the APPL language, we have included the output from the Simple Rover and the classical Banana-Monkey example in the appendices. The level of abstraction provided by the APPL language is seen in the twofold increase in size as one goes from APPL to NDDL.

4.1 Features of the Translator

Much of the verbosity of the NDDL language derives from the need to specify sequences of actions. The following code defined in a timeline named `Location` in `Monkey.appl`:

```
At(x) -> Going(x,y) -> At(y)
```

results in the following NDDL:

```
// Transitions for Location

Location::At {
  meets(Going v0_);
  eq(v0_.from,loc);
}

Location::Going {
  meets(At v1_);
  eq(v1_.loc,to);
}
```

where a simple matching algorithm is used to generate the constraints on the parameters of the actions `Going` and `At`. Even when there are no matching parameters as in

```
Low -> Climbing -> High -> Climbing_Down -> Low
```

the expansion can be significant:

```
// Transitions for Altitude

Altitude::Low {
  meets(Climbing);
```

[3]The translator is electronically available from `http://research.nianet.org/~munoz/APPL`.

```
}

Altitude::Climbing {
  meets(High);
}

Altitude::High {
  meets(Climbing_Down);
}

Altitude::Climbing_Down {
  meets(Low);
}
```

The following language construct in `Rover.appl`

```
Idle -> TakeSample -> (PhoneHome | PhoneLander)
```

provides a efficient mechanism for specifying choice. In NDDL this must be expressed using an explicit variable. The translator generates the following NDDL code

```
// Transitions for Commands

Commands::Idle {
  meets(TakeSample);
}

Commands::TakeSample {
  int v3_ = [1 2];
  if (v3_ == 1) {
    meets(PhoneHome);
  }
  if (v3_ == 2) {
    meets(PhoneLander);
  }
}
```

The translation of the `WITH` statements associated with actions such as

```
Going(from,to: Location)
   WITH from != to;
        let path:Path(_,from,to,cost) in
        starts Battery.Change(cost)
```

is quite efficient. This is translated into

```
  Navigator::Going {
    Path path;
    eq(path.from,from);
    eq(path.to,to);
    starts(Battery.Change v0_);
    eq(v0_.quantity,path.cost);
  }
```

This illustrates how APPL handles parameter matching and `let-in` expressions. A variable `path` is defined such that the first two parameters correspond to the parameters of the predicate `Going`. The variable `cost` implicitly declared in APPL will be translated into an equality that relates the parameter of the predicate `Change` in `Battery` and the third parameter of `path`.

5 Conclusion and Future Work

A major goal of the APPL language is to provide as much abstraction as possible. We believe that this will greatly facilitate the translation into SAL and provide a rich environment for the formal analysis of the planning domain under consideration. Therefore, in the first version of the language, we have deliberately eschewed enhancements that would increase its complexity even where the power or flexibility of the language would be significantly increased. As needs arise in real applications, we will expand the language to meet these needs. It is our hope that such modifications can be kept at a minimum and that the abstract nature of the language can be preserved. In this section, we will briefly discuss some of potential enhancements that we have considered but not included.

5.1 Intervals as Timelines

One limitation of the current version of APPL is that it does not provide a mechanism to explicitly access the starting and ending times of an action. For instance, in the Simple Rover example, the goal of taking a sample is specified as:

```
goal(Commands.TakeSample sample);
sample.start.specify(63);
precedes(sample.end, world.m_horizonEnd);
```

The intended meaning is that `TakeSample` must start at time 63 and end before a constant `world.m_horizonEnd`, which has the value 100.

We have considered adding the ability to directly specify the starting and ending times of actions in expressions. For example

```
    Commands.TakeSample().start = 63;
    Commands.TakeSample().end < 100
```

Alternatively, we have explored the idea of allowing intervals to be used as special anonymous actions in Allen's expressions. In this case, the lower bound of the interval specifies the starting time of the anonymous actions and the upper bound the ending time. For instance, the previous goal could be expressed:

```
GOALS
  Commands.TakeSample() starts [63,_]
```

This concise notation fully exploits Allen's operators and may simplify specifications where absolute time intervals are needed. For instance,

```
GOALS
  Commands.TakeSample() equals [63,100]
```

specifies that the action `TakeSample` exactly starts at time 63 and ends at time 100, while

```
GOALS
  Commands.TakeSample() contained_by [63,100]
```

specifies that the action `TakeSample` starts after time 63 and ends before time 100.

However, the semantic implications of either one of these alternatives have not yet been analyzed.

5.2 General Initial State Specifications

The APPL planning language has been defined with a specific viewpoint about what constitutes a planning problem. In particular, that a planning problem consists of

1. a well-defined initial state,
2. a well-defined goal,
3. actions to achieve that goal, and
4. constraints on those actions.

It appears that NDDL allows a fuzzier notion of initial state, where some timelines do not become active until a time later than 0 or have relaxed start and end points. In APPL this could be handled by allowing general initial state specifications. For instance,

```
INITIAL-STATE
  |->  Navigator.At(lander) contained_by [10,_]
```

would specify that the initial location for `Navigator` is `lander` after time 10. Nothing is said about the state of the navigator before that time.

This relaxation of initial state specifications would prevent detection of errors where one inadvertently omits a timeline initialization.

References

[1] Tania Bedrax-Weiss, Conor McGann, Andrew Bachmann, Will Edington, and Michael Iatauro. Europa-2: User and contributor guide. Technical report, NASA AmesResearch Center, Moffett Field, CA, Feb 2005.

[2] Jeremy Frank and Ari Jonsson. Constraint-based attribute and interval planning. Technical report, NASA AmesResearch Center, Moffett Field, CA, 2002. to appear in the Journal of Constraints, Special Issue on Constraints and Planning.

[3] Drew McDermott and AIPS'98 IPC Committee. PDDL–the planning domain definition language. Technical report, Yale University, 1998. Technical report, Available at: www.cs.yale.edu/homes/dvm, 1998.

A Simple Rover Problem

A.1 APPL Code

```
PLAN Rover

  TYPE Paths = {Very_Long_Way, Moderately_Long_Way, Short_Cut}

  TYPE Locations = {Lander, ROCK1, ROCK2, ROCK3, ROCK4}

  TYPE Location(name:Locations; x, y:int)

  TYPE Path(name:Paths; from,to: Location; cost:float)

  TYPE Battery(ic,ll_min,ll_max:float) =
       Resource(ic,ll_min,ll_max,0.0,0.0,-inff,-inff)

  TIMELINE Navigator

  ACTIONS

    At(location:Location)

    Going(from,to: Location)
      WITH from != to;
           let path:Path(_,from,to,cost) in
           starts Battery.change(cost)

  TRANSITIONS

    At(x) -> Going(x,y) -> At(y)

  END Navigator


  TIMELINE Commands

  ACTIONS

    Idle

    TakeSample(rock:Location): [20,25]

    PhoneHome
      WITH starts Battery.change(-600)

    PhoneLander
      WITH starts Battery.change(-20)

  TRANSITIONS

    Idle -> TakeSample -> (PhoneHome | PhoneLander)

  END Commands

  TIMELINE Instrument

  ACTIONS

    TakeSample(rock:Location) : [0,10]
      WITH starts Battery.change(-120);
           contained_by Navigator.At(rock);
           contained_by Commands.TakeSample(rock)

    Place(rock:Location) : [3,12]
      WITH starts Battery.change(-20);
           contained_by Navigator.At(rock)

    Stow : [2,6]
      WITH contained_by Navigator.At

    Unstow : [2,6]
      WITH contained_by Navigator.At

    Stowed

  TRANSITIONS

    Stowed -> Unstow -> Place(r) -> TakeSample(r) -> Stow -> Stowed

  END Instrument


  TYPE Rover(battery:Battery;command:Commands; navigator:Navigator;
            instrument:Instrument)

  VARIABLES
```

```
    lander: Location(Lander, 0, 0)
    rock1 : Location(ROCK1, 9, 9)
    rock2 : Location(ROCK2, 1, 6)
    rock3 : Location(ROCK3, 4, 8)
    rock4 : Location(ROCK4, 3, 9)

    p1: Path(Very_Long_Way, lander, rock4, -2000.0)
    p2: Path(Moderately_Long_Way, lander, rock4, -1500.0)
    p3: Path(Short_Cut, lander, rock4, -400.0)

    bat : Battery(1000.0, 0.0, 1000.0)
    com : Commands
    nav : Navigator
    ins : Instrument
    spirit : Rover(bat,com,nav,ins)

  INITIAL-STATE

    |-> Navigator.At(lander)
    |-> Instrument.Stowed
    |-> Commands.Idle

  GOALS

    Commands.TakeSample(rock4) equals [63,t]

END Rover
```

A.2 Translator-Generated NDDL Code

```
// File Rover-model.nddl
// Generated from Rover.appl
// On Tue Nov 21 14:52:57 EST 2006
// By APPL-c.4 (11/17/06)

// Import Standard Europa NDDL Libraries

  #include "Plasma.nddl"
  #include "PlannerConfig.nddl"

  PlannerConfig World = new PlannerConfig(0,100,500,+inf);

// Forward declarations

  class Battery;
  class Path;
  class Rover;
  class Location;
  class Instrument;
  class Commands;
  class Navigator;

// Class declarations

  enum Paths {Very_Long_Way,Moderately_Long_Way,Short_Cut};

  enum Locations {Lander,ROCK1,ROCK2,ROCK3,ROCK4};

  class Location {
    Locations name;
    int x;
    int y;

    Location(Locations _name,
             int _x,
             int _y) {
      name = _name;
      x = _x;
      y = _y;
    }// Constructor Location

  }// Class Location

  class Path {
    Paths name;
    Location from;
    Location to;
    float cost;

    Path(Paths _name,
         Location _from,
         Location _to,
         float _cost) {
      name = _name;
      from = _from;
      to = _to;
      cost = _cost;
    }// Constructor Path
  }// Class Path

  class Battery extends Resource {
    Battery(float ic,
            float ll_min,
            float ll_max) {
      super(ic,ll_min,ll_max,0.0,0.0,-inff,-inff);
    }// Constructor Battery

  }// Class Battery

  class Navigator extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate At {
      Location location;
      precedes(0,start);
      precedes(end,100);
    }// Predicate At

    predicate Going {
      Location from;
      Location to;
      precedes(0,start);
      precedes(end,100);
      neq(from,to);
    }// Predicate Going

  }// Timeline Navigator

  Navigator::Going {
    Path path;
    eq(path.from,from);
    eq(path.to,to);
    starts(Battery.change v0_);
    eq(v0_.quantity,path.cost);
  }

// Transitions for Navigator

  Navigator::At {
    meets(Going v1_);
    eq(v1_.from,location);
  }

  Navigator::Going {
    meets(At v2_);
    eq(v2_.location,to);
  }

  class Commands extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate Idle {
      precedes(0,start);
      precedes(end,100);
    }// Predicate Idle

    predicate TakeSample {
      Location rock;
      eq(duration,[20 25]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate TakeSample

    predicate PhoneHome {
      precedes(0,start);
      precedes(end,100);
    }// Predicate PhoneHome

    predicate PhoneLander {
      precedes(0,start);
      precedes(end,100);
    }// Predicate PhoneLander

  }// Timeline Commands

  Commands::PhoneHome {
    starts(Battery.change v3_);
    eq(v3_.quantity,-600);
  }

  Commands::PhoneLander {
    starts(Battery.change v4_);
```

```
    eq(v4_.quantity,-20);
  }

// Transitions for Commands

  Commands::Idle {
    meets(TakeSample);
  }

  Commands::TakeSample {
    int v5_;
    eq(v5_,[1 2]);
    if (v5_ == 1) {
      meets(PhoneHome);
    }
    if (v5_ == 2) {
      meets(PhoneLander);
    }
  }

  class Instrument extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate TakeSample {
      Location rock;
      eq(duration,[0 10]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate TakeSample

    predicate Place {
      Location rock;
      eq(duration,[3 12]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Place

    predicate Stow {
      eq(duration,[2 6]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Stow
    predicate Unstow {
      eq(duration,[2 6]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Unstow

    predicate Stowed {
      precedes(0,start);
      precedes(end,100);
    }// Predicate Stowed

  }// Timeline Instrument

  Instrument::TakeSample {
    starts(Battery.change v6_);
    eq(v6_.quantity,-120);
    contained_by(Navigator.At v7_);
    eq(v7_.location,rock);
    contained_by(Commands.TakeSample v8_);
    eq(v8_.rock,rock);
  }

  Instrument::Place {
    starts(Battery.change v9_);
    eq(v9_.quantity,-20);
    contained_by(Navigator.At v10_);
    eq(v10_.location,rock);
  }

  Instrument::Stow {
    contained_by(Navigator.At);
  }

  Instrument::Unstow {
    contained_by(Navigator.At);
  }

// Transitions for Instrument

  Instrument::Stowed {
    meets(Unstow);
  }
  Instrument::Unstow {
    meets(Place v11_);
  }

  Instrument::Place {
    meets(TakeSample v12_);
    eq(v12_.rock,rock);
  }

  Instrument::TakeSample {
    meets(Stow);
  }

  Instrument::Stow {
    meets(Stowed);
  }

  class Rover {
    Battery battery;
    Commands command;
    Navigator navigator;
    Instrument instrument;

    Rover(Battery _battery,
          Commands _command,
          Navigator _navigator,
          Instrument _instrument) {
      battery = _battery;
      command = _command;
      navigator = _navigator;
      instrument = _instrument;
    }// Constructor Rover

  }// Class Rover

// File Rover-initial-state.nddl
// Generated from Rover.appl
// On Tue Nov 21 14:52:57 EST 2006
// By APPL-c.4 (11/17/06)

// Import Rover NDDL Model

  #include "Rover-model.nddl"
```

```
// Variables

  Location lander = new Location(Lander,0,0);
  Location rock1 = new Location(ROCK1,9,9);
  Location rock2 = new Location(ROCK2,1,6);
  Location rock3 = new Location(ROCK3,4,8);
  Location rock4 = new Location(ROCK4,3,9);
  Path p1 = new Path(Very_Long_Way,lander,rock4,-2000.0);
  Path p2 = new Path(Moderately_Long_Way,lander,rock4,-1500.0);
  Path p3 = new Path(Short_Cut,lander,rock4,-400.0);
  Battery bat = new Battery(1000.0,0.0,1000.0);
  Commands com = new Commands();
  Navigator nav = new Navigator();
  Instrument ins = new Instrument();
  Rover spirit = new Rover(bat,com,nav,ins);
  close();

// Initial states

  goal(Navigator.At v13_);
  eq(v13_.start,0);
  v13_.location.specify(lander);
  goal(Instrument.Stowed v14_);
  eq(v14_.start,0);
  goal(Commands.Idle v15_);
  eq(v15_.start,0);

// Goals

  goal(Commands.TakeSample v16_);
  v16_.rock.specify(rock4);
  v16_.start.specify(63);
```

B Banana-Monkey Problem

The classical Banana-Monkey planning problem consists of three basic timelines: `Location`, `Altitude`, and `Monkey`. The location timeline has two predicates: `At` and `Going`. These must alternate on the timeline.

$$\texttt{At} \rightarrow \texttt{Going} \rightarrow \texttt{At} \rightarrow \texttt{Going} \rightarrow \dots$$

The altitude timeline has four predicates which must sequence as follows

$$\texttt{Low} \rightarrow \texttt{Climbing} \rightarrow \texttt{High} \rightarrow \texttt{Climbing\_Down} \rightarrow \texttt{Low} \rightarrow \dots$$

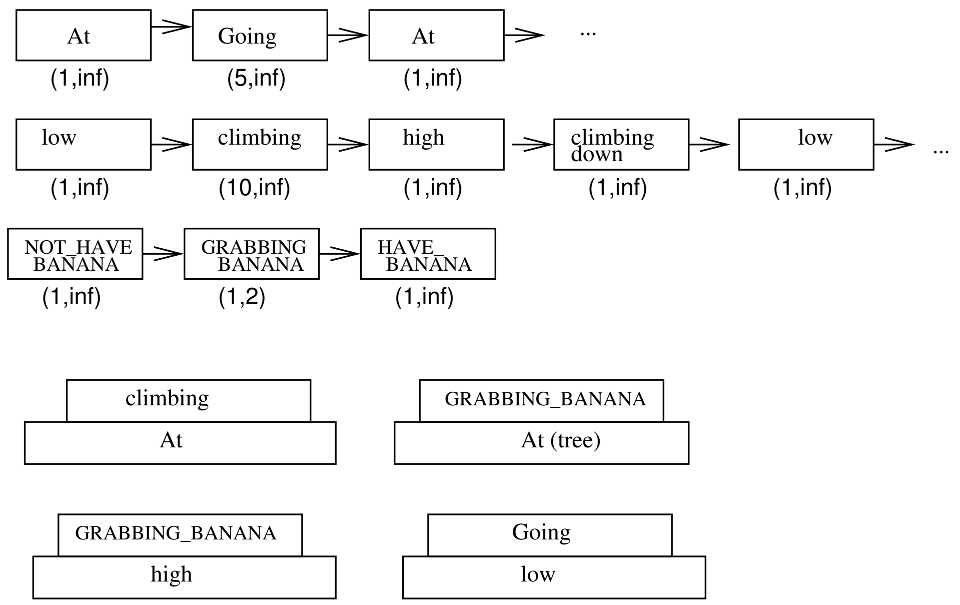


Figure 2: Monkey1 Problem

The banana timeline has three predicates that must sequence as follows:

`Not_Have_Banana` $\rightarrow$ `Grabbing_Banana` $\rightarrow$ `Have_Banana`

The NDDL specification constrains the time intervals as follows

- The `Climbing` interval must be contained by an `At` interval. A parameter of `Climbing` determines whether the climbing is a tree climbing action or a rock climbing action.
- The `Grabbing_Banana` interval must be contained by an `At` interval and it must be contained by a `High` interval.
- The `Grabbing_Banana` predicate must only occur when the monkey is `high` in the tree.
- The `Going` predicate must only occur when the altitude is `low`.

Graphically this problem can be presented as shown in figure 2 The intervals below the blocks in the figure specify the duration of each predicate, i.e., the minimum and maximum times in this state.

B.1 APPL Code

```
PLAN Monkey

TYPE Label = { Rock, Tree }

TIMELINE Location
```

```
  ACTIONS

    At(loc:Label): [1,_]
    Going(from,to:Label): [5,_]

  TRANSITIONS

    At(x) -> Going(x,y) -> At(y)

END Location

TIMELINE Altitude

  ACTIONS

    Low: [1,_]

    High: [1,_]

    Climbing(flag:[1,2]): [10,_]
      WITH
        if flag = 1 then
          contained_by Location.At(Rock)
        else
          contained_by Location.At(Tree)
        endif

    Climbing_Down: [1,_]

  TRANSITIONS
    Low -> Climbing -> High -> Climbing_Down -> Low

END Altitude

TIMELINE Monkey

  ACTIONS

    Not_Have_Banana: [1,_]

    Have_Banana: [1,_]

    Grabbing_Banana: [1,2]
  TRANSITIONS

    Not_Have_Banana -> Grabbing_Banana -> Have_Banana

END Monkey

CONSTRAINTS

  Location.Going :: contained_by Altitude.Low
  Monkey.Grabbing_Banana :: contained_by Location.At(Tree)
  Monkey.Grabbing_Banana :: contained_by Altitude.High

VARIABLES

  loc : Location
  alt : Altitude
  mon : Monkey

INITIAL-STATE

  |-> loc.At(Rock)
  |-> alt.Low
  |-> mon.Not_Have_Banana

GOALS

  mon.Have_Banana

END  Monkey
```

B.2 Translator-Generated NDDL Code

```
// File Monkey-model.nddl
// Generated from Monkey.appl
// On Tue Nov 21 14:53:00 EST 2006
// By APPL-c.4 (11/17/06)

// Import Standard Europa NDDL Libraries

  #include "Plasma.nddl"
  #include "PlannerConfig.nddl"

  PlannerConfig World = new PlannerConfig(0,100,500,+inf);
```

```
// Forward declarations

  class Monkey;
  class Altitude;
  class Location;

// Class declarations

  enum Label {Rock,Tree};

  class Location extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate At {
      Label loc;
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate At

    predicate Going {
      Label from;
      Label to;
      eq(duration,[5 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Going

  }// Timeline Location

// Transitions for Location

  Location::At {
    meets(Going v0_);
    eq(v0_.from,loc);
  }

  Location::Going {
    meets(At v1_);
    eq(v1_.loc,to);
  }
  class Altitude extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate Low {
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Low

    predicate High {
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate High

    predicate Climbing {
      int flag;
      eq(flag,[1 2]);
      eq(duration,[10 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Climbing

    predicate Climbing_Down {
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Climbing_Down

  }// Timeline Altitude

  Altitude::Climbing {
    if (flag==1) {
      contained_by(Location.At v2_);
      eq(v2_.loc,Rock);
    } else {
      contained_by(Location.At v3_);
      eq(v3_.loc,Tree);
    }
  }

// Transitions for Altitude
```

```
  Altitude::Low {
    meets(Climbing);
  }

  Altitude::Climbing {
    meets(High);
  }

  Altitude::High {
    meets(Climbing_Down);
  }

  Altitude::Climbing_Down {
    meets(Low);
  }

  class Monkey extends Timeline {
    predicate Null {
      precedes(0,start);
      eq(end,100);
    }
    predicate Not_Have_Banana {
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Not_Have_Banana

    predicate Have_Banana {
      eq(duration,[1 +inf]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Have_Banana

    predicate Grabbing_Banana {
      eq(duration,[1 2]);
      precedes(0,start);
      precedes(end,100);
    }// Predicate Grabbing_Banana

  }// Timeline Monkey

// Transitions for Monkey
```

```
  Monkey::Not_Have_Banana {
    meets(Grabbing_Banana);
  }

  Monkey::Grabbing_Banana {
    meets(Have_Banana);
  }

// Constraints

  Location::Going {
    contained_by(Altitude.Low);
  }

  Monkey::Grabbing_Banana {
    contained_by(Location.At v4_);
    eq(v4_.loc,Tree);
  }

  Monkey::Grabbing_Banana {
    contained_by(Altitude.High);
  }

// File Monkey-initial-state.nddl
// Generated from Monkey.appl
// On Tue Nov 21 14:53:00 EST 2006
// By APPL-c.4 (11/17/06)

// Import Monkey NDDL Model

  #include "Monkey-model.nddl"

// Variables

  Location loc = new Location();
  Altitude alt = new Altitude();
  Monkey mon = new Monkey();
  close();

// Initial states

  goal(loc.At v5_);
  eq(v5_.start,0);
  v5_.loc.specify(Rock);
  goal(alt.Low v6_);
  eq(v6_.start,0);
  goal(mon.Not_Have_Banana v7_);
  eq(v7_.start,0);

// Goals

  goal(mon.Have_Banana v8_);
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01- 11 - 2006	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE An Abstract Plan Preparation Language	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Butler, Ricky W.; and Muñoz, César A.	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 70680.04.13.01.02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-19280
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM-2006-214518

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

This paper presents a new planning language that is more abstract than most existing planning languages such as the Planning Domain Definition Language (PDDL) or the New Domain Description Language (NDDL). The goal of this language is to simplify the formal analysis and specification of planning problems that are intended for safety-critical applications such as power management or automated rendezvous in future manned spacecraft. The new language has been named the Abstract Plan Preparation Language (APPL). A translator from APPL to NDDL has been developed in support of the Spacecraft Autonomy for Vehicles and Habitats Project (SAVH) sponsored by the Explorations Technology Development Program, which is seeking to mature autonomy technology for application to the new Crew Exploration Vehicle (CEV) that will replace the Space Shuttle.

15. SUBJECT TERMS
Artificial Intelligence; Autonomy; Formal Methods; Planning; Verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	43	(301) 621-0390