

# The ANML Language

David E. Smith  
*Intelligent Systems Division  
NASA Ames Research Center  
Moffet Field, CA 94035–1000  
David.Smith@nasa.gov*

William Cushing  
*Dept. of Comp. Sci. and Eng.  
Arizona State University  
Tempe, AZ 85281  
William.Cushing@asu.edu*

## Abstract

*The Action Notation Modeling Language (ANML) is being developed to provide a high-level, convenient, and succinct alternative to existing planning languages such as PDDL, the EUROPA modeling language (NDDL), and the ASPEN modeling language (AML). ANML is based on strong notions of action and state (like PDDL and AML), uses a variable/value model (like NDDL and AML), supports rich temporal constraints (like NDDL and AML), and provides simple, convenient idioms for expressing the most common forms of action conditions, effects, and resource usage. The language supports both generative and HTN planning models in a uniform framework and has a clear, well-defined semantics. In this paper, we highlight the central and unique features of the ANML language.*

## 1. Introduction

There are a number of different languages that have been developed and used at NASA for modeling planning domains and problems. Chief among these are the NDDL language [1] developed for the EUROPA2 planner at Ames Research Center, and the ASPEN Modeling Language (AML) [2] developed at JPL. While both of these languages have their strong points, they are quite different in their basic concepts and assumptions, and both have significant weaknesses.

The NDDL language facilitates the description of constraints among intervals. It characterizes every state or activity as an interval and specifies the possible temporal relationships between those intervals. However, NDDL has no notion of action, state, fact, or goal. A planning problem in NDDL simply involves filling out the set of timelines with intervals so that there are no gaps and all the constraints are obeyed. NDDL models are often quite verbose, unintuitive, and contain redundant constraints. It is also easy to make modeling mistakes in the language, and debugging models is difficult. In contrast, AML is based on fairly intuitive notions of action and state, and contains many convenient constructs for describing resource usage and temporal constraints. However, the language is fundamentally geared towards HTN planning, and does not fully sup-

port generative planning.

In contrast to these NASA languages, the academic community continues to develop and use the PDDL family of languages. PDDL1.2 [3] is based on a STRIPS model of action, so there is no ability to model time, duration, concurrency, numeric resources, or temporal constraints. The successors, PDDL2.1 [4] and PDDL2.2 [5], introduced *durative* actions, and continuous numeric variables. These languages therefore permit simple modeling of time, concurrency, and resources. However, the languages are still primarily proposition based, and do not directly support more complex temporal conditions, effects, or constraints. It is also cumbersome to express common patterns of change and resource usage, and it is easy to make errors in doing so [6]. PDDL3.0 [7] has introduced the ability to model more complex temporal constraints and preferences on goals, but the capabilities do not extend to actions or conditions. Despite these limitations, PDDL has become the de facto standard in the academic planning community and is used in the biennial International Planning Competitions [8, 9, 10]. As a result, many example domains are available in PDDL, and many effective planners and techniques have been developed for different versions of the language.

We have been developing the Action Notation Modeling Language (ANML) in an effort to 1) provide a high level, convenient, and succinct alternative to existing languages, 2) support both generative and HTN planning models in a uniform framework, 3) provide a language with clear, well-defined semantics, and 4) allow greater compatibility with the evolving PDDL family of languages. ANML is based on strong notions of action and state (like AML and PDDL), uses a variable/value representation (like NDDL and AML), supports rich temporal constraints (like NDDL and AML), and provides simple, convenient idioms for expressing the most common forms of action conditions and effects. As an example action description in ANML, we could express a simple high-level navigate action for a rover as shown in Figure 1. Navigate has two location parameters from and to, and a fixed duration of 5. The *temporal qualifier* over all states that the three following statements apply to the entire duration of the action. The first is a condition stating that the arm must

```

action Navigate (location from, to) {
  duration := 5 ;
  over all { arm == stowed ;
            position == from ::= to ;
            batterycharge :consumes 2.0 } }

```

Figure 1: A simple ANML action.

```

(:durative-action navigate
 :parameters (?from - location ?to - location)
 :duration (= ?duration 5)
 :condition (and (at start (position ?from))
                (at start (stowed))
                (over all (stowed))
                (at start (>= (batterycharge) 2.0))
                )
 :effect (and (at start (decrease (batterycharge) 2.0))
             (at start (not (position ?from)))
             (at end (position ?to))
             ))

```

Figure 2: The equivalent PDDL2.1 action.

remain stowed over the entire action. The second is a combination of a condition and two effects stating that the location must initially be the location from, is undefined in the interim, and will be the location to at the end of the action. The third is an effect stating that the action consumes two units of energy. Among other things, more complex temporal qualifiers are possible and more complex functional expressions are possible for duration and energy consumption.

For comparison purposes, Figure 2 shows an equivalent model in PDDL2.1. There are a number of syntactic differences between the two descriptions, but these are relatively unimportant. The more significant differences in both size and simplicity are due partly to ANML's variable/value representation, but also to ANML's more powerful constructs for describing change. In the ANML description, we have not partitioned the statements into conditions and effects. Instead, we have described what happens to each relevant variable in turn.

In the sections that follow, we explain these capabilities in greater detail, and highlight the novel features of the ANML language. We focus on the powerful and concise constructs in ANML for temporal qualification, for describing change over the course of an action, for describing resource usage, and for integrating task decomposition with traditional action models. To do this, we must first introduce the basic entities of the ANML language, time varying *variables* and *functions*.

## 2. Basic Declarations

Constants, variables, and functions in ANML are typed. There are a number of built-in types, including: int, float, bool, string, symbol, object, vector, variable, function, and action. There is no need to declare numeric or string constants in ANML, but objects and

symbols must be declared. For example:

```
object spirit, opportunity, rock1, sample2 ;
```

declares spirit, opportunity, rock1, and sample2 to be constants of type object.<sup>1</sup> It is also possible to declare new types in ANML. For example:

```
type positiveInt float [0.0, inf] ;
type rover object {spirit, opportunity, pathfinder} ;
type color symbol {blue, green, red, yellow, purple} ;
```

defines the types positiveInt, rover, and color as specializations of the types Int, object, and symbol respectively. It is also possible to define specialized vector types, e.g:

```
type location vector( positiveInt x, y ) ;
type path vector( location from, to ) ;
```

which defines locations to be vectors of two integer elements, x and y, and paths to be vectors of two locations, from and to. ANML also allows the definition of more complex structured types, but we will not discuss this further here.

### 2.1 Variables

When declaring a variable in ANML, one must specify the domain of the variable. The declaration therefore consists of the keyword variable followed by a type, followed by the variable name. Any predefined or user-defined type can be used for this purpose. For example:

```
variable bool havePicture ;
variable string sampleName, pictureName ;
variable positiveInt [0, 5] sample# ;
variable float [0.0, inf] batterycharge, wheelCurrent ;
variable color {blue, red, green} filterColor ;
variable rover r ;
variable location position ;
```

are all legitimate variable declarations.

For convenience, variables can be initialized in a declaration. For example:

```
variable float roverSpeed := 30.0 ;
```

is equivalent to stating:

```
variable float roverSpeed ;
at start { roverSpeed := 30 } ;
```

Vectors can also be initialized in this way. For example:

```
variable vector( int x, y ) position := (5, 20) ;
```

is equivalent to stating:

```
variable vector( int x, y ) position ;
at start { position.x := 5 ;
          position.y := 20
          }
```

<sup>1</sup>As far as ANML is concerned, object and symbol are synonymous, but the former is typically used for tangible entities like spacecraft and rovers, while symbol is typically used for intangibles like colors or shapes.

## 2.2 Functions

Functions in ANML are essentially parameterized variables. So when declaring a function symbol, in addition to specifying the range of the function, one must also specify the domains of the parameters or arguments. The declaration for a function symbol therefore consists of the keyword `function` followed by a type, followed by a function symbol and its typed argument list. For example:

```
function float [0, inf] batterycharge(rover r) ;
```

indicates that the function `batterycharge` of a rover is a positive float. As with variables, functions in ANML are implicit functions of time. In fact, variables in ANML can be considered as functions having zero arguments. Unlike variables, the value of a function cannot be assigned in the declaration, because doing so might require different assignments for different arguments.

Predicates in ANML are simply functions onto the boolean values `true` and `false`. Thus, the statements:

```
predicate hasesample ;
predicate stowed(instrument i) ;
```

are equivalent to stating:

```
variable bool hasesample ;
function bool stowed(instrument i) ;
```

## 3. Actions

Actions are the means by which one changes the world. In general, an ANML action description consists of:

- the action name
- a typed parameter list (optional)
- a duration assignment (optional)
- local variable or function declarations (optional)
- one or more temporally qualified conditions, effects, or change statements

In Figure 1 we showed an example of a simple navigate action in ANML:

```
action Navigate (location from, to) {
  duration := 5
  over all { arm == stowed ;
            position == from ::= to ;
            batterycharge :consumes 2.0 } }
```

The action name and typed parameter list are shown in the first line, and the duration assignment is shown in the second line. The action does not contain any local variable definitions, but we could have done something like `define` a local variable for energy use in terms of duration, and then express the consumption using this local variable:

```
variable float energy := duration * use-rate ;
over all { batterycharge :consumes energy } ;
```

The main body of our example action consists of the three lines:

```
over all { arm == stowed ;
          position == from ::= to ;
          batterycharge :consumes 2.0 } ;
```

In general, the body can express simple conditions (like the first line above), simple effects, or more complex combinations of conditions and effects (lines 2 and 3). To be more precise, these statements take the form:

*Temporal Qualifier*  $\{\phi_1; \dots; \phi_n\}$

where *Temporal Qualifier* is something like `at start` or `over all`, and each of the  $\phi_i$  is a condition, effect, or change expression. A simple condition expression is of the form:

*variable relation expression*

where *relation* is typically `==`, `in`, or a numeric comparator (`<`, `≤`, `>`, `≥`). The expression is frequently a constant, or another variable, but can be a set, interval or an algebraic expression of numeric variables and constants.

A simple effect expression is of the form:

*variable assignment expression*

where *assignment* is one of `:=` or `:in`. The expression is the same as for conditions, but can also be undefined (it is not allowed to condition on a variable being undefined). Conditions and effects can be distinguished because the relations in conditions are distinct from the assignment operators used in effects. Because of this, there is no need to separate them or delineate them with a keyword as in PDDL.

Combinations of conditions and effects only make sense over intervals. In general, they are of the form:

```
variable relation expression1
assignment expression2
:assignment expression3 ;
```

The relation and expression in the first line expresses a condition that must hold at the beginning of the interval. For example, in the expression:

```
over all { position == from
          := undefined
          ::= to } ;
```

the condition is `position == from`, which must hold at the beginning of the interval. This condition expression may be omitted, which would indicate that there is no requirement on the initial value of the variable. The assignment and expression in the second line is an effect indicating the value of the variable over the interior of the interval. This assignment can also be undefined, in which case, the variable is assumed to be undefined during the interval. The last assignment and expression is also an effect indicating the final value of the variable at the end of the interval. In the example above, the final effect `::= to` specifies that the position will be the destination `to` at the end of the interval. Note that the assignment operator for a final effect is preceded by an additional colon to distinguish it from an interim effect. Thus the above expression could be expressed as a simple condition and two simple effects:

```

at start { position == from } ;
over interim { position := undefined } ;
at end { position := to } ;

```

### 3.1 Temporal Qualifiers

A temporal qualification indicates the time or time period over which a variable has a particular value or changes its value. In the example above, we saw the use of the temporal qualifiers: *over*, *at start* and *at end*. These qualifiers are special cases of a more general temporal qualifier *in*, and can also refer to time points or intervals other than just the start, end, or entire duration of an action. The general form for the temporal qualifier *in* is:

```
in  $i$  dur  $d$   $\Phi$  ;
```

where  $i$  is an interval of time,  $d$  is a duration and  $\Phi$  is a collection of conditions. It means that  $\Phi$  must hold for at least the duration  $d$  within the interval  $i$ . If the duration is omitted,  $\Phi$  need only hold for some instant within the specified interval. As an example, the condition:

```
in [start+5,end-2) dur 3 { heater == on } ;
```

specifies the condition that the heater must be on for at least 3 time units between the start of the action plus 5 time units and the end of the action minus 2 time units. The time interval can be closed or open at either end, and time points are specified relative to start and end, meaning the start and end of the action respectively. The keyword *all* is short for the closed interval [start,end] and the keyword *interim* is short for the open interval (start,end). The full power of the qualifier *in* is allowed for specifying conditions. However, if used for effects or change it could allow the expression of uncertainty. We therefore limit its use to pure conditions. For effects and change, we are limited to the two special cases *at* and *over*. The temporal qualifier *at* is defined as:

```
at  $t$   $\Phi \equiv$  in [ $t,t$ ]  $\Phi$  ;
```

and means that  $\Phi$  must hold or take place at the time instant  $t$ . As with *in*,  $t$  can be any time point relative to the start or end of the action.

Similarly, the temporal qualifier *over* is defined as:

```
over  $i$   $\Phi \equiv$  in  $i$  dur  $\|i\|$   $\Phi$  ;
```

where  $\|i\|$  refers to the length of the interval  $i$ . As with *in*, the interval  $i$  can be closed or open at either end, and the time points are specified relative to the start or end of the action.

### 3.2 Relative Change

For numeric variables, it is often useful to specify effects relative to the existing value, rather than in absolute terms. For example we might want to specify that a particular drilling operation advances the drill a certain amount beyond its current depth. We could state this as:

```
at end { depth := depth + increment } ;
```

Some languages make this a bit easier by allowing additional operators like  $+=$  and  $-=$ . In ANML we do this by specifying change on the “delta” of the depth variable rather than on the depth variable itself. For the above example we would say:

```
at end {  $\Delta$ depth := increment } ;
```

The meaning of this is that the depth variable changes by the amount *increment*.<sup>2</sup> The reason we take this approach is that it allows us to conveniently say other more difficult things like:

```
over all {  $\Delta$ depth ::= increment } ;
over all {  $\Delta$ depth :in [0, increment] ::= increment } ;
```

The first of these implies that the change in depth is undefined over the course of the interval, before taking on the final value *increment*. The second implies that the change in depth is bounded by the interval [0, *increment*] over the course of the interval, before taking on the final value *increment*. These statements are considerably more cumbersome using only the basic primitives, because they require defining a temporary variable to hold the initial value of the depth variable, e.g:

```
variable float start-depth := depth ;
over interim { depth := undefined } ;
at end { depth := start-depth + increment } ;
```

As we will see in the next section, composite incremental change statements will prove extremely convenient for describing resource consumption and production.

## 4. Resources

*Resources* are common and convenient abstractions in many planning and scheduling applications. There are many different kinds of resources – they can be discrete or continuous, and consumable or reusable.<sup>3</sup> Discrete resources can also be *unit-capacity*, or *multi-capacity*. The different types of resources are illustrated in Table 1.

	<b>Discrete</b>	<b>Continuous</b>
<b>Reusable</b>	instruments, tools	bandwidth, power
<b>Consumable</b>	solid rockets	energy, fuel, cryogen

Table 1: Resource types.

As far as ANML is concerned, resources are just numeric variables. They are therefore declared in the same way:

<sup>2</sup>As a practical matter, the delta symbol ( $\Delta$ ) is a bit hard to type on most keyboards, so we use the caret symbol (^) as a substitute, e.g. ^depth instead of  $\Delta$ depth.

<sup>3</sup>The first of these dimensions, *discrete or continuous*, is a property of the resource variable itself. The second, *consumable or reusable* is a property of how the resource is used. In fact, it is entirely possible for a quantity to be a reusable resource for one action and a consumable resource for another.

```
variable float [10.0, 100.0] batterycharge := 50.0 ;
variable integer [0, 12] sample-bags := 12 ;
```

It is the usage of these resources where ANML provides additional facilities beyond the notation described so far. We start with *reusable* resources.

#### 4.1 Usage

A *reusable* resource is one that is *consumed* at the beginning of an action, but given back (or *produced*) at the end. Using the mechanisms for relative change that we introduced above, we could express this as:

```
at start {  $\Delta$ resource := -quantity } ;
at end {  $\Delta$ resource := quantity } ;
```

However, because resource use is so common, we have introduced a more convenient way of expressing this pair of effects:

```
over all { resource :uses quantity } ;
```

It is important to note that for a resource effect like this, the condition:

```
at start { resource >= quantity } ;
```

is implicit because of the definition of the resource variable. For example, a declaration of *batterycharge* as:

```
variable float [10.0, 100.0] batterycharge ;
```

implicitly requires that the quantity remain within the interval [10.0, 100.0]. Any action that would violate these bounds would not be legal.

The counterpart to *using* a resource for a period of time is to make a resource available for a period of time. For example, by running a generator, we could make additional power available. We could simply model this as negative resource usage, but for convenience and clarity we refer to this as *lending* a resource and express it as:

```
over all { resource :lends quantity } ;
```

#### 4.2 Consumption and Production

As we hinted above, resource usage can be thought of as a pair of consumption and production effects. Using the mechanisms for modeling relative change, we can model instantaneous resource consumption as:

```
at t {  $\Delta$ resource := -quantity } ;
```

For convenience and clarity we allow this to also be stated as:

```
at t { resource :consumes quantity } ;
```

Similarly, resource production can be modeled as:

```
at t {  $\Delta$ resource := quantity } ;
```

or for convenience as:

```
at t { resource :produces quantity } ;
```

When dealing with consumption and production of resources, it is common to make a conservative modeling assumption that consumption occurs at the beginning of an action and production occurs at the end. In

reality, consumption and production usually occur gradually over the course of actions, although the actual function may be complex or even unknown. The conservative discretization ensures that there is enough of a resource at the beginning of a consumption action, and that we do not rely on any production actions until their end. In effect, this approach tracks a lower bound for the resource, and guarantees that plans will never violate the lower bound limit for the resource variable. While this works well in many situations, it can run into trouble if the resource variable also has an upper bound, or *capacity*, such as for a battery, fuel tank, or storage container [6]. Figure 3 illustrates the problem. A navigate action is performed while the battery is be-

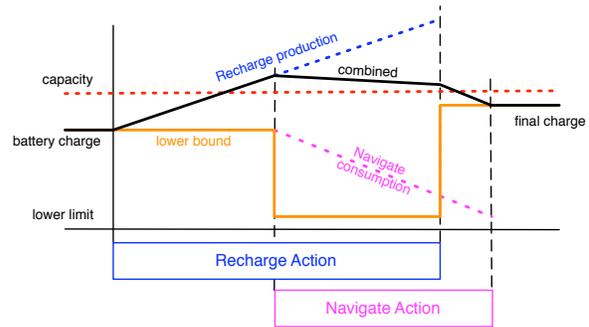


Figure 3: Illustration of simultaneous consumption and production activities. Although the lower bound remains within the allowed range for the resource, the actual value exceeds resource capacity unless the consumption activity is started earlier.

ing recharged (by solar panels). The conservative lower bound envelope for the two actions remains within the allowed range for the battery. However, since production actually occurs before the end of the recharge action, the actual charge envelope will exceed battery capacity. This kind of problem can occur any time there is the possibility of simultaneous consumption and production, and the resource has a capacity limit. Cushing and Smith [6] discuss this problem in detail, and discuss some alternative approaches to dealing with it. The most obvious way of dealing with this problem (without resorting to detailed modeling and reasoning about continuous consumption and production) is to keep track of both a lower bound and an upper bound for a resource. For example, we could model the behavior of a consumption action as:

```
at start {  $\Delta$ resource_lb := -quantity } ;
at end {  $\Delta$ resource_ub := -quantity } ;
```

and a production action as:

```
at start {  $\Delta$ resource_ub := quantity } ;
at end {  $\Delta$ resource_lb := quantity } ;
```

While sound, this requires that we define and keep track of two explicit variables for each resource, and get the timing and polarity of the conditions and effects right.

We regard this as both complex and cumbersome, and likely to result in many modeling errors. Instead, we think that the user should be able to simply state that consumption or production occurs over the course of an action (or interval). The planner should manage the details of keeping track of lower and upper bounds for variables in order to assure soundness. Using the notation for relative change, and for compound change statements, we would express a typical resource consumption as:

```
over all {  $\Delta$ resource :in [-quantity,0] ::= -quantity } ;
```

This states that over the course of the action, the resource change is guaranteed to be between zero (upper bound) and  $-quantity$  (lower bound) and will be at the lower bound at the end of the interval. We define the shorthand:

```
over all { resource :consumes quantity } ;
```

to mean precisely this for an interval; the quantity change is bounded over the course of the interval, and ends with the specified change. Similarly, resource production over an interval:

```
over all { resource :produces quantity } ;
```

is defined as:

```
over all {  $\Delta$ resource :in [0,quantity] ::= quantity } ;
```

The `:consumes` and `:produces` statements are extraordinarily convenient and powerful, because they allow the user to specify complex production and consumption activities without requiring details of the actual consumption and production functions, and without requiring that the user explicitly provide upper and lower bound discrete approximations. Finally, it is worth noting that for some lengthy activities, like solar recharging, and driving, it may be necessary to break production and consumption into phases so that actions can be performed concurrently in order to make effective use of resources, and keep them within bounds. For example we could model solar production over the course of an extended period as a series of production activities:

```
over [start, start+5] { energy :produces 2.0 } ;
over [start+5, start+10] { energy :produces 3.5 } ;
...
over [start+95, end] { energy :produces 1.3 } ;
```

This allows consumption activities like navigation or shunting to be performed throughout the production without violating either lower bound or capacity limits.

## 5. HTN Decomposition

Hierarchical Task Networks (HTNs) and task decomposition planning techniques are used for many practical planning systems and applications [11, 12, 13, 14]. For HTN systems, goals are stated in terms of high level *tasks* to be performed, and *methods* allow for the decomposition of these tasks into lower level tasks and primitive actions. Many within the planning community have argued against the HTN representation and

task decomposition on the grounds that they describe what actions *should* be used for (their purpose), rather than what actions *do* (their conditions and effects). As such, this representation can lead to systems that are highly tailored to a certain class of problems, but can be brittle if they are asked to solve problems that require using actions in unanticipated ways. All of this is true. Nevertheless, there are some good arguments for allowing task decomposition in a modeling language:

1. Some planning domains seem to be more naturally expressed in terms of task decomposition.
2. There are situations where the modeler may be able to characterize ways of achieving tasks without having a clear understanding of the effects and conditions of the underlying actions.

In developing ANML, we did not wish to unduly constrain the modeler, or force the modeler to describe actions at an unnecessarily fine level of detail. As a result, ANML allows the specification and use of action decomposition by allowing decomposition expressions within action descriptions. We have done this in a novel way that allows action decomposition to be fully integrated with ordinary action descriptions containing conditions and effects. There are three things necessary to make this work:

1. Each action (instance)  $A$  is regarded as having an implicit effect proposition of the same name ( $A$ ) over its execution. In other words:
 

```
over all { A == false := true ::= false } ;
```
2. We allow these “action” propositions to be used as ordinary conditions within action descriptions.
3. We allow relative ordering constraints on conditions.

Allowing relative ordering constraints among conditions requires some additional machinery which we now describe.

### 5.1 Relative Ordering Constraints

Suppose that we have two action conditions  $p$  and  $q$ , that must be true at some point during the action, e.g:

```
in all { p ; q } ;
```

Now imagine that we also require that  $p$  become true before  $q$ . Using the machinery presented so far, we could specify explicit intervals for  $p$  and  $q$  that guaranteed this, but we could not simply specify a relative ordering constraint between the two conditions. To do this, we need to be able to name conditions, and refer to their start and end times. We name conditions by following them with the symbol  $!$  and a name, e.g:

```
in all { p ! c1 ; q ! c2 } ;
```

To refer to the start and end times of conditions we generalize the keywords `start` and `end` to allow them to refer to the start and end of specific actions, e.g: `start(c1)`. We can then specify the desired relative ordering constraint as:

$\text{start}(c_1) < \text{start}(c_2)$  ;

The entire action condition would therefore be:

in all {  $p ! c_1$  ;  $q ! c_2$  } ;  
 $\text{start}(c_1) < \text{start}(c_2)$  ;

Alternatively, we could specify the two conditions separately as:

in all {  $p ! c_1$  } ;  
 in (  $\text{start}(c_1)$ , end ] {  $q$  } ;

This avoids the need to actually name the second condition  $q$ . This ordering capability is quite general and allows us to specify complex temporal constraints among arbitrary sets of conditions. For convenience, we introduce the shorthand  $\text{ordered}(p_1, \dots, p_k)$  to refer to an ordered set of conditions. In other words:

in all {  $\text{ordered}(p_1, \dots, p_k)$  } ;

is equivalent to naming each of the conditions, and specifying temporal constraints between each successive pair:

in all {  $p_1 ! c_1$  ; ... ;  $p_k ! c_k$  } ;  
 $\text{end}(c_1) \leq \text{start}(c_2)$  ;  
 ...  
 $\text{end}(c_{k-1}) \leq \text{start}(c_k)$  ;

A similar shorthand  $\text{unordered}(p_1, \dots, p_k)$  can be used to refer to an unordered set of conditions. Together,  $\text{ordered}$  and  $\text{unordered}$  can be used to describe a partial ordering of conditions. For example:

$\text{ordered}(p, \text{unordered}(q, \text{ordered}(r, s)), t)$  ;

corresponds to the partial order shown in Figure 4.

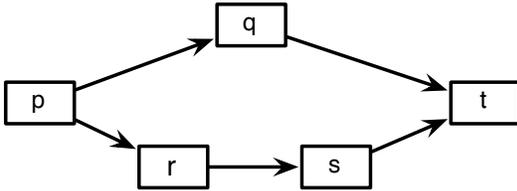


Figure 4: A partially ordered set of conditions.

## 5.2 Decompositions

With the ability to express relative ordering constraints on conditions, we now have the tools necessary to express decomposition in an action description. As an example, suppose we want to specify a method for the high level task `CollectSample`, which can be decomposed into four primitive actions of unstowing the instrument, placing the instrument, taking the sample, and stowing the instrument. We can express this in ANML as:

```

action CollectSample(location l) {
  over all { position == l } ;
  in all { ordered( Unstow, Place(l),
                  TakeSample(l), Stow ) } ;
  ... } ;
  
```

In effect, this says that in order to successfully perform the `CollectSample` action, we must successfully perform the `Unstow`, `Place()`, `TakeSample()`, and `Stow` actions, in order.

## 6. Related Work

The ANML language has taken inspiration from several existing languages. Among these, the NDDL language has the most powerful capability for expressing temporal constraints. However, these capabilities are not necessarily easy to understand or use, and we find many aspects of the language to be both cumbersome and difficult to use effectively. The AML language has very convenient and natural syntax for expressing resource usage, but the HTN nature of the language is limiting. The PDDL family of languages has been carefully developed, and is widely adopted in the research community. However, the propositional nature of the language, the limited constructs for describing change and resource usage, and the limited ability to model time and temporal constraints make it difficult or impossible to use for serious applications. The change notation in ANML most closely resembles constructs developed in the SAS family of languages [15]. However, in overall capability and style, the IxTeT language [16] is perhaps the closest.

There have been a few previous attempts to merge HTN decomposition into more traditional action languages like PDDL [3, 14]. However, these attempts have not been widely used or adopted because the two paradigms have had separate semantics and have not really been integrated. The semantics we ascribe to decompositions in ANML is quite different. Essentially, we regard a decomposition as simply being another set of conditions necessary for performing the action. In other words, if the subtasks can be performed in the order indicated, then the high-level task can be performed. This makes sense for several reasons:

1. If the decompositions for an action prove to be impossible (cannot be performed), then the action itself is not possible (or we do not know its outcome).
2. Multiple decompositions correspond to a disjunction of sets of conditions, and any one of these sets would be sufficient to accomplish the action.
3. Logical conditions can be mixed with decompositions.
4. It is consistent with, and can be seen as a generalization of allowing general temporal constraints among conditions.

## 7. Conclusions and Future Work

In this short paper, we have only sketched some of the key features of the ANML language. In particular, we have described the powerful and concise constructs in ANML for temporal qualification, for describ-

ing change over the course of an action, for describing resource usage, and for integrating task decomposition with traditional action conditions and effects. There are additional features and details of the language that we have not mentioned, or have only glossed over. Among other things, there is the ability to:

- define structured objects
- express quantification
- express disjunctive conditions
- express conditional effects
- express complex goals

A draft manual describing the language is available by request, and we are in the process of building translators from PDDL into ANML and from ANML into NDDL. Because of the expressiveness of ANML, only a subset of the language can be translated into AML, but this is also being considered.

There is an additional effort underway to extend the syntax of ANML to allow description of continuous change, processes and exogenous events. We would like to be able to express resource usage as a function of time where known and convenient. However, our view here is not that a planner must necessarily be able to reason about continuous change. Instead, our view is that we should allow the user to naturally express the domain, at whatever level of detail is appropriate. It is then perfectly reasonable for a planner to make sound but incomplete approximations to effectively reason about the domain. As a case in point, reasoning about lower and upper bound envelopes is a useful approximation that is both computationally tractable, and perfectly adequate for many domains. However, we want the planner to be able to choose the approximation, rather than forcing the user to encode it explicitly. Thus in the case of continuous change, we would like to see the planner choose an appropriate discretization or linearization in order to enable sound, but effective, reasoning about the domain.

## 8. Acknowledgements

Thanks to Tony Barrett, Matthew Boyce, Maria Fox, Jeremy Frank, Ari Jonsson, and Conor McGann for considerable discussion and feedback on the language and its design. This work was supported by the Automation for Operations project of the NASA ETDP program.

## 9. References

- [1] T. Bedrax-Weiss, C. McGann, A. Bachmann, W. Edgington, and M. Iatauro, "EUROPA2: User and contributor guide," NASA Ames Research Center, Tech. Rep., 2005.
- [2] R. Sherwood, B. Engelhardt, G. Rabideau, S. Chien, and R. Knight, "ASPEN user's guide," JPL, Tech. Rep. D-15482, 2005. Available: <http://ai.jpl.nasa.gov/public/projects/aspn>
- [3] D. McDermott, "PDDL – the Planning Domain Definition Language: Version 1.2," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998. Available: <http://www.cs.yale.edu/homes/dvm>
- [4] M. Fox and D. Long, "PDDL2.1: An extension of PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, 2003, pp. 61–124.
- [5] S. Edelkamp and J. Hoffmann, "PDDL2.2: The language for the classical part of the 4th International Planning Competition," Fachbereich Informatik, University of Dortmund, Tech. Rep. 195, 2004. Available: <http://ls5-web.cs.uni-dortmund.de/edelkamp/ipc-4/DOCS/pddl-ipc-2.ps.gz>
- [6] W. Cushing and D. Smith, "The perils of discrete resource models," in *ICAPS-07 Workshop on International Planning Competition: Past, Present and Future*, 2007.
- [7] A. Gerevini and D. Long, "Plan constraints and preferences in PDDL3: The language of the Fifth International Planning Competition," Department of Electronics for Automation, University of Brescia, Tech. Rep., 2005. Available: <http://zeus.ing.unibs.it/ipc-5/pddl-ipc5.pdf>
- [8] D. Long, H. Kautz, B. Selman, B. Bonet, H. Geffner, J. Koehler, M. Brenner, J. Hoffmann, F. Rittinger, C. R. Anderson, D. S. Weld, D. E. Smith, and M. Fox, "The AIPS-98 planning competition," *AI Magazine*, 2000.
- [9] D. Long and M. Fox, "The 3rd International Planning Competition: Results and analysis," *Journal of Artificial Intelligence Research*, vol. 20, 2003, pp. 1–59.
- [10] J. Hoffmann and S. Edelkamp, "The deterministic part of IPC-4: An overview," *Journal of Artificial Intelligence Research*, vol. 24, 2005, pp. 519–579.
- [11] D. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.
- [12] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barret, G. Stebbins, and D. Tran, "ASPEN - Automated planning and scheduling for space missions operations," in *International Conference on Space Operations (SpaceOps 2000)*, Toulouse, France, 2000.
- [13] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, 2003, pp. 379–404.
- [14] L. Castillo, J. Fdez-Olivares, O. Garcia-Pérez, and F. Palao, "Efficiently handling temporal knowledge in an HTN planner," in *Proc. of the Sixteenth Intl. Conf. on Automated Planning and Scheduling (ICAPS-06)*, 2006.
- [15] P. Jonsson and C. Bäckström, "Tractable planning with state variables by exploiting structural restrictions," in *Proc. of the Twelfth National Conf. on Artificial Intelligence (AAAI-94)*, 1994.
- [16] M. Ghallab and H. Laruelle, "Representation and control in IxTeT, a temporal planner," in *Proc. of the Second Intl. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*. AAAI Press, 1994, pp. 61–67.