NASA/TM-2007-215088

The ANMLite Language and Logic for Specifying Planning Problems

Ricky W. Butler
Langley Research Center, Hampton, Virginia

Radu I. Siminiceanu and César A. Muñoz
National Institute of Aerospace, Hampton, Virginia

November 2007

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2007-215088

The ANMLite Language and Logic for Specifying Planning Problems

Ricky W. Butler
Langley Research Center, Hampton, Virginia

Radu I. Siminiceanu and César A. Muñoz
National Institute of Aerospace, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

November 2007

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Contents

1 Overview of Planning Problems

In this report we seek to define a simple language that can be used to describe planning problems. Hopefully, by drastically restricting the constructs in the language, two benefits will accrue: (1) the language will be easy to understand and write, and (2) the language will lend itself to formal verification.

We have named the language ANMLite because it was developed to support the analysis of planning domains described in the Action Notation Modeling Language (ANML) [1] under development at NASA Ames. In ANMLite, a planning problem consists of a finite set of disjoint timelines, a set of valid actions for each timeline, and a set of temporal constraints that govern the correct scheduling of the actions. The constraints can be broadly categorized into two groups. The first group is specified by a *transition relation* and only involves actions on the same timeline. These constraints express the valid succession of actions along the timeline. The transition relation disallows overlapping actions and gaps on a timeline. The second group consists of general constraints, expressed in some logic of choice, which specify cross-timeline relationships between actions. The temporal logic must be chosen with care. It has to be rich enough to cover all the significant relations that can occur (such as Allen temporal operators [2], a popular logic in planning), but simple enough to avoid inconsistencies and ambiguities. Furthermore, since we seek to develop a framework for formal verification, it must be translatable into a form suitable for model checking or theorem proving. We are currently targeting the SAL model checker [3].

Some planning problems require data structure mechanisms to properly specify them. We have begun to look at some approaches to specifying such problems. This has led to the introduction of simple constraint and effect statements that operate on global variables and data structures. We suspect that it is wise to limit the power of these constructs and avoid them where possible, but we are not prepared at this time to make any firm recommendations about these new constructs. At this time they are significantly weaker than the corresponding ANML constructs.

2 Timelines

Discovering a suitable sequence of actions on a timeline is fundamental to the planning/scheduling problem. The first step is to identify all the actions that can be scheduled on a timeline. We have followed the ANML convention and will specify timelines using the keyword `OBJTYPE`. All of the allowed actions are then enumerated as in the following example:

```
OBJTYPE A ACTIONS
    A0
    A1: [10,_]
    A2: [2,_]
    A3
```

This specification defines the timeline `A` and its four actions: `A0`, `A1`, `A2`, and `A3`. Actions `A1` and `A2` have time duration constraints: `A1` takes at least 10 time units and `A2` takes at

least 2 time units. Usually, in a planning problem, there are also constraints on the sequence of actions, so an intuitive, unambiguous specification of these constraints is highly desirable. There are two different approaches to the specification of these constraints:

- Assume that all action sequences are possible unless specifically forbidden and then specify the sequences that are *not* allowed.
- Assume that no sequences are allowed and then systematically add the allowed sequences.

We have currently opted for the second approach. We recognize that this is different from many AI planning systems, but it follows the approach frequently used in the formal methods community. We currently believe that this leads to a clearer specification, though we recognize that we may be biased by the historic conventions of our discipline.

2.1 Transitions

The transition relation on a timeline is similar to state-transition systems. Here, the states are the actions and a directed edge represents a valid transition between states. We have used the same construction deployed in the Abstract Plan Preparation Language (APPL) [4]. Hence, the transition relation is a set of pairs of actions, which can be declared by listing for each action the (complete) set of its successors, as in the following example:

```
TRANSITIONS

   A0 -> A1 -> A2 -> (A0 | A1 | A3)
   A3 -> A2
```

Action `A0` can only be followed by action `A1`. Action `A1` can only be followed by action `A2`. Action `A2` can be followed by actions `A0`, `A1`, or `A3`. Action `A3` can only be followed by action `A2`.

The flexibility of the language is increased by allowing parametrization of actions. For example, the following

```
A1(x,y: animal): [10,_]
```

defines an action `A1` with two parameters of type `animal` that takes at least 10 units of times. The type `animal` is defined as follows:

```
TYPE animal = {cat, dog, fish, horse, eel, chicken, donkey, snake}
```

and `A1(cat,snake)` is a valid instance of this action. We allow more restrictive forms of transitions to be defined using a simple parameter matching scheme, with implicitly declared variables. For example,

```
A1(cat,yy) -> A2(yy,_)
```

This constraint states that only A1 actions with a first parameter equal to cat are to be followed by an A2 action and that the first parameter of A2, represented by the variable yy, must be equal to the second parameter of A1. Unless explicitly specified on a different constraint, no other transition from A1 is allowed. Similarly,

```
A1(xx,_) -> A2(dog,xx)
```

Here, all A1 actions must be followed by an A2 action where the first parameter equals dog and the second parameter, represented by the variable xx, equals the first parameter of A1. Notice the use of the place holder _ in the second parameter of A1, which matches any possible value of the given type.

Timeline instances are defined using the VARIABLE section as follows:

```
VARIABLES
   t1,t2: A
   t3: B
```

This specification declares two distinct timelines, t1 and t2, defined by OBJTYPE A, and one timeline t3 defined by OBJTYPE B.

The variables of the same OBJTYPE share the transition relation, but might still behave differently, in case specific constraints are declared in the general constraint section. This is beneficial in terms of keeping the model compact, and it is frequently seen in practice. For example, the crew activity model (see Section 6) declares a general timeline for a crew member, which describes the common activities, such as sleeping, eating, etc. Additional constraints referring to non-routine tasks are then expressed by using specific crew member identifiers to enforce, for example, that a given astronaut is disassembling the sleeping unit, while a different one is fixing the power unit.

2.2 Goal Statements and Initialization

In ANMLite, goals can be specified by an action name:

```
GOALS
   t1.A3
   t2.A2
```

This specifies that A3 is scheduled and completes on timeline t1, and that A2 is scheduled and completes on timeline t2. A generic form is also supported:

```
GOALS
   A.A2
```

where A is declared to be a timeline using an OBJTYPE declaration. This expression means that A2 is scheduled and completes on *every* timeline of type A. For example, if we have

```
VARIABLES
      t1,t2,t3,t4: A
```

then the generic goal `A.A2` is equivalent to

```
GOALS
       t1.A2
       t2.A2
       t3.A2
       t4.A2
```

Initial states can also be specified using an `INITIAL-STATE` declaration though they are not necessary. For example,

```
INITIAL-STATE

|-> t1.A0
|-> t2.A1
```

This specifies that `A0` is the first scheduled action on timeline `t1` and that `A1` is the first scheduled action on timeline `t2`. A generic form is also allowed

```
INITIAL-STATE

|-> A.A0
```

This means that on *every* timeline of type `A`, `A0` is the first scheduled action.

2.3 Transition Statement Extension

For specifications where there is a large fan-out from many states, we are experimenting with the following extension to the transition statement

```
A1 -> *
```

which means that `A1` can transition into any of the actions on the timeline including itself. One can restrict the allowed transitions by adding a list of exceptions as illustrated below

```
A1 -> * \ (A2 | A5)
```

This means that `A1` can transition into any of the actions on the timeline except `A2` and `A5`.

3 Constraints

The transition statements are adequate to specify the allowed sequences of actions on a timeline, but they cannot be used to specify constraints between actions on different timelines. The constraint section is used to accomplish this. The ANMLite constraints are built upon a simple but powerful foundation: the start and end times of actions. These time points can be referenced as follows

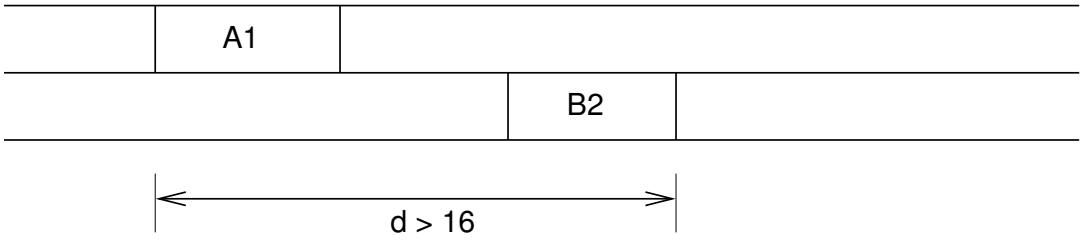
```
A1.start

B2.end
```

Constraints are expressed as simple linear relations between these variables:

```
A1.start + 16 < B2.end
```

This constraint can be visualized on a timeline as follows



The constraints can also be chained as follows

```
A1.start + 4 < B1.start <  C1.end - 10
```

This is equivalent to writing

```
A1.start + 4 < B1.start AND B1.start <  C1.end - 10
```

Restricting the constraint language to these simple linear relationships enables a very natural translation into the SAL model checking language (see Section 5).

3.1 Contains Example

Suppose we want to specify a `contains`-like constraint between actions, inspired from the Allen Temporal Logic [2]. In the New Domain Description Language (NDDL) [5], we would write

```
B1 contains A1
```

and in ANML we might write something like

```
objtype B {
  ...
  action B1 { ... }
}

objtype A {
  action A1 {
      condition over all : B.state == B1;
  }
```

The following diagram illustrates the intended behavior graphically

In ANMLite we would express this constraint as follows:

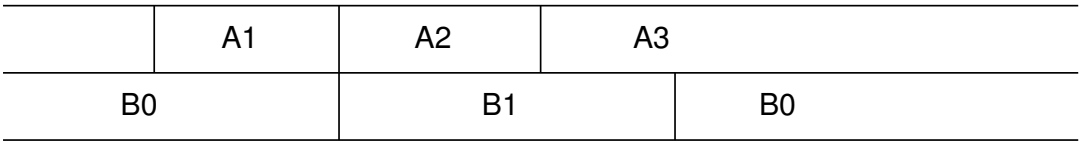
```
B1.start  < A1.start < A1.end < B1.end
```

3.2 Meets Example

Suppose we want to specify a `meets`-like constraint. In NDDL we write

```
A1 meets B1
```

The following diagram illustrates the intended behavior



In ANMLite we would express this constraint simply as:

```
A1.end = B1.start
```

3.3 Repetitive Actions

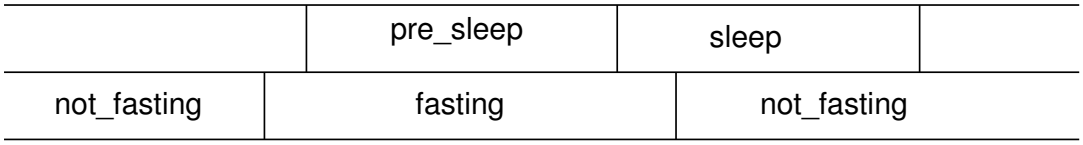
Consider the following fragment of ANML:

```
objtype FastingWindow {
  FastingState fs;
  transition fs {
    "fasting"     -> "not_fasting";
    "not_fasting" -> "fasting"
  };

  action not_fasting()
  {
    change over all { fs  = "fasting" -> "not_fasting" -> "fasting" };
  }
}
```

```
objtype CrewMember {
  FastingWindow fw;
  action pre_sleep {
      condition over all : fw.state == fasting();
  }
}
```

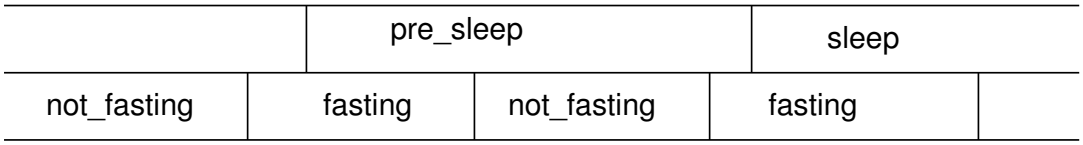
The condition statement asserts that the state of another timeline, called `fw`, must be `fasting` throughout the execution of `pre_sleep`:



We can express this as follows in ANMLite:

```
fasting.start < pre_sleep.start < pre_sleep.end < fasting.end
```

This states that the start of the `fasting` action must take place before the start of `pre_sleep` and the end of the `fasting` action must take place after the end of `pre_sleep`. The question is whether this specification rules out the following scenario:



It does if `fasting.start` and `fasting.end` refer to the first fasting interval and not the second one. So we clearly need to distinguish between these intervals. We could call the first interval `fasting` and the second interval `next fasting`. Now suppose we want to specify that `pre_sleep` must start with `fasting` active and end with `fasting` active and that `not_fasting` executes in between as shown above. This can be expressed as:

```
fasting.start < pre_sleep.start < pre_sleep.end < next fasting.end AND
pre_sleep.start < not_fasting.start < not_fasting.end < pre_sleep.end
```

Another question is whether `not_fasting` should be considered as current or not, that is whether to use `not_fasting` or `next not_fasting`. To disambiguate, we need a reference point from which the current action and the next action can be determined.

We consider two approaches[1]:

- Provide a new construct to establish a reference point.
- Define a default reference point such as the earliest or latest time point in the expression.

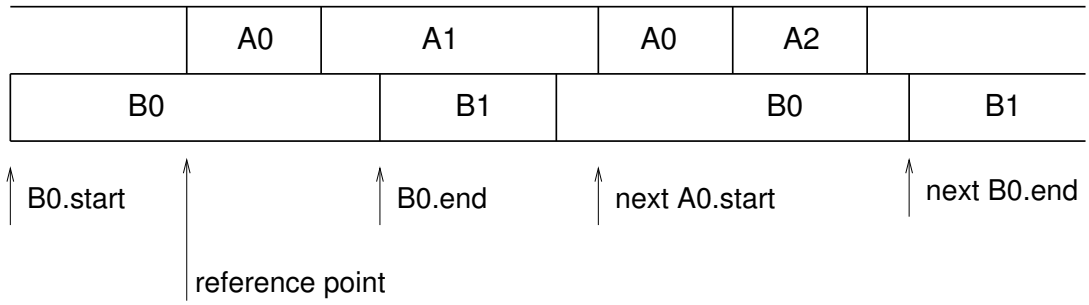
[1]We have also explored the use of logic quantifiers ($\forall, \exists$), but we currently believe that the next operator approach is simpler and easier to understand. However, it is not without some difficulties which will be discussed subsequently.

3.3.1 The `at` Expression

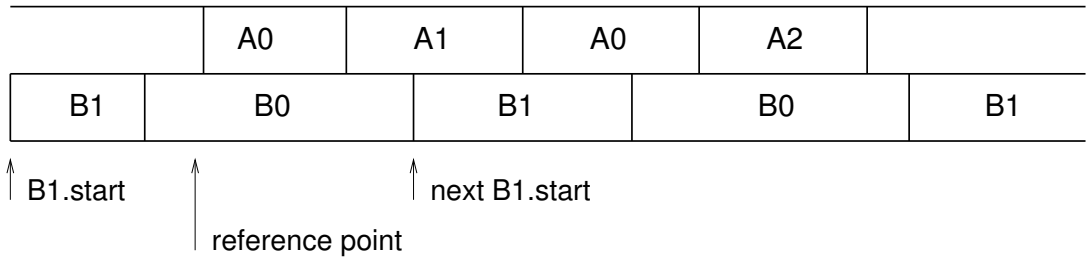
The first approach could be achieved by introducing a new expression, namely an `at` expression, which specifies a reference timepoint. For example

```
at A0.start:  B0.end < next A0.start
```

In this case, all actions that are active at the timepoint `A0.start` are the current ones. The next instance after the completion of the current one is the `next` one. For example



If the action is not active at the reference point, then the "current" one is the last one and the next one is the first occurrence after the reference point. For example

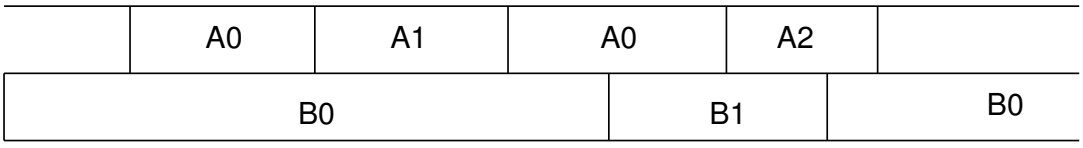


It should also be noted that there is an implicit universal quantifier in every constraint. If the reference point involves action `A1`, e.g., `A1.start`, and `A1` can occur multiple times on a timeline, then this constraint applies every time `A1` is scheduled.

Note that the constraint above, namely

```
at A0.start:  B0.end < next A0.start
```

would rule out the following solution



Because of the flexibility in attaching a reference point and the implicit universal quantifier, it becomes apparent that the same constraint could have a different set of solutions, depending on the location of the `at` expression.

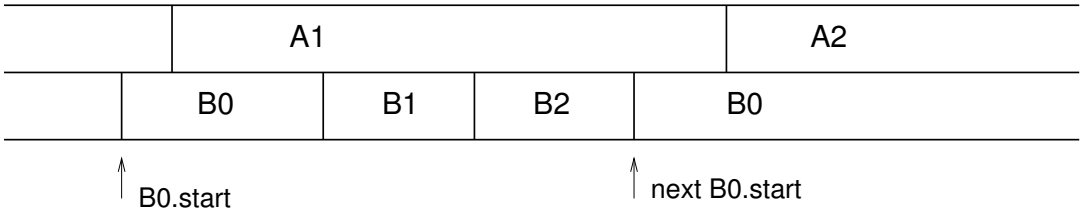
Consider the following constraint

```
B0.start + 5 < next A2.start < next B0.start + 7
```

If the reference point is chosen as `A1.start`

```
at A1.start:  B0.start + 5 < next A2.start < next B0.start + 7
```

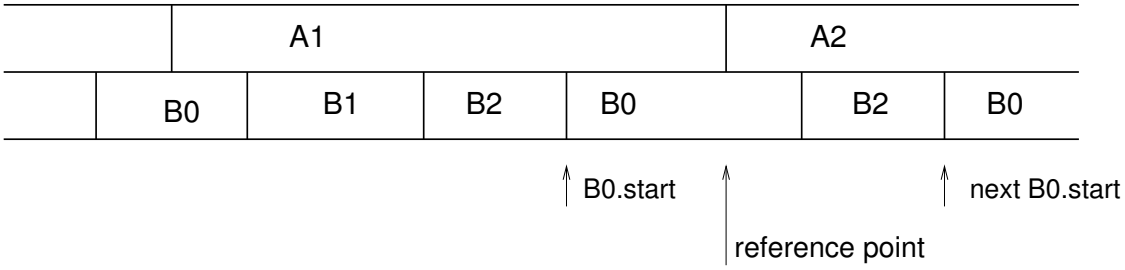
then the following timeline is a potential solution, when `next B0.start + 7 > next A2.start`



If we change the `at` expression as follows:

```
at A1.end:  B0.start + 5 < next A2.start < next B0.start + 7
```

the above timeline no longer satisfies the constraint



because the location of `B0.start` has changed. Therefore, one should use much care in the application of the `at` expression.

3.3.2 Using a Default Time Reference Point

We have also experimented with a default reference point, namely the first or last timepoint that appears in the expression. However, without a restriction on the `at` expression, some ambiguities can arise. These ambiguities will be demonstrated using the following transition graphs:

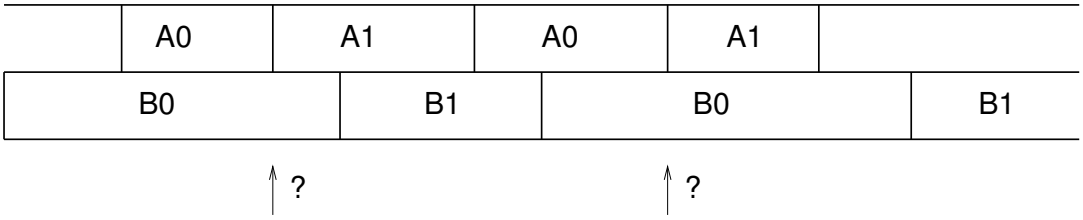
```
A0 -> A1 -> A2 -> A1 -> A3

B0 -> B1 -> B2 -> B1 -> B3
```

Suppose we use the first timepoint that appears in the expression. But how should one interpret the following?

```
next A1.start < B1.end
```

Here the first expression is `next A1.start`. In this case, determining which instance of `A1` is the reference point is problematic:



The same impasse is reached when trying to determine which instance of `B1` is referred to by the `B1.end` term.

Now suppose we make the last timepoint the reference expression. To create an ambiguity, one only needs reverse the constraint

```
A1.start < next B1.end
```

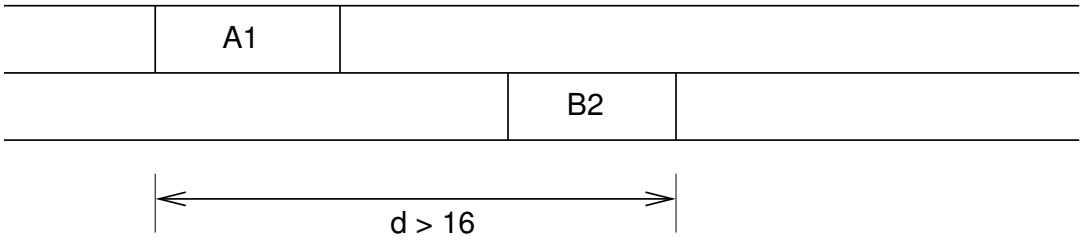
Of course we can disallow the use of `next` in the first position in the first approach (or last position in the second approach), though this is a bit artificial. Nevertheless, this is probably the best approach. We need only decide whether to use the first or last term. If we use the first term as the default, then the first constraint we looked at, namely,

```
A1.start + 16 < B2.end
```

would have to be rewritten as

```
A1.start + 16 < next B2.end
```

if the following timeline is to be a solution:



Therefore, we think that the best approach is to make the default the last term. By making the default reference point the last term, all of the previous terms are earlier in time and hence they are current ones (by definition of the reference point). Therefore, this choice provides a nice seemless unification with the non-repetitive cases. If the last term has a `next` operator, then an explicit `at` expression must be provided[2].

[2]Another attractive alternative is to require an `at` expression whenever there is a repetitive action.

3.4 Timeline Instance Specific Constraints

Constraints can be specialized by using a timeline variable in the constraint. Suppose we have

```
VARIABLES
    t1,t2: A
    t3,t4: B

CONSTRAINTS
     t1.A1.start < t4.B1.end
```

Now this constraint only affects timelines `t1` and `t4`. But the constraint

```
     A1.start < B1.end
```

is equivalent to four constraints:

```
     t1.A1.start < t3.B1.end
     t1.A1.start < t4.B1.end
     t2.A1.start < t3.B1.end
     t2.A1.start < t4.B1.end
```

Mixing qualified and unqualified action names may be problematic. For example

```
     t1.A1.start < B2.end
```

Even though a default interpretation can be envisioned as treating the unqualified term as "for all" timeline instances `tb` of `B`

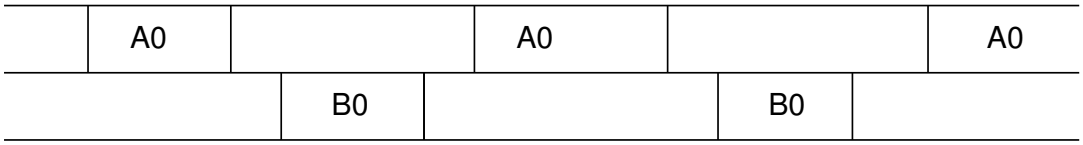
```
     t1.A1.start < tb.B2.end
```

This is however dangerous when the terms are on the same timeline, because it can easily lead to contradictions, such as in

```
     t1.A1.end < t2.A2.start < A1.end
```

3.5 The Interleaving Example

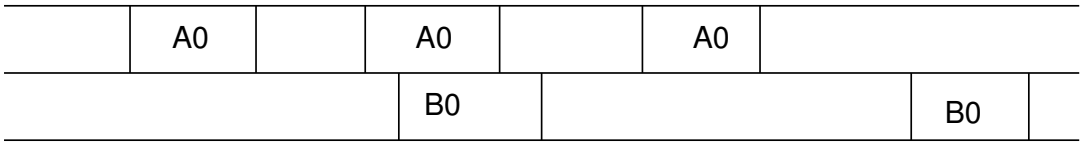
Suppose we need to specify that repetitive actions `A0` and `B0` are interleaved as illustrated below:



Let's try to write some constraints that enforce this behavior

```
A0.end < B0.start
B0.end < A0.start
```

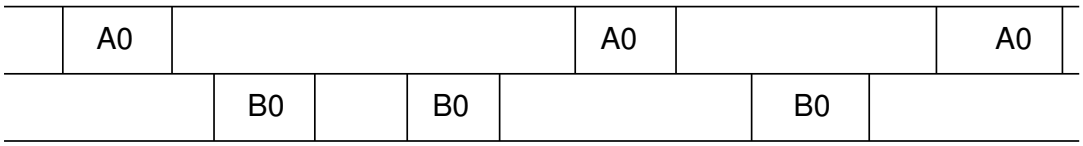
Immediately we encounter a problem. Consider the following timeline:



It satisfies the constraints because there are multiple instances of `A0` and `B0`. In this case, it is possible that `A0` and `B0` overlap but the constraint is still satisfied. If instead we combine the two inequalities in a single chain

```
at B0.start: A0.end < B0.start < B0.end < next A0.start
```

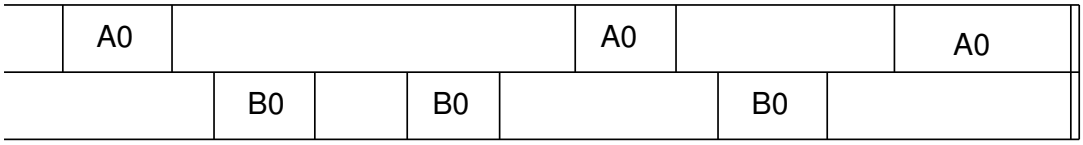
we rule out the possibility of overlap: `A0` has to have ended before `B0` starts and cannot occur again until `B0` ends. However, this is still not the complete specification. This constraint requires that *at least one* instance of `B0` occurs between successive instances of `A0`. But consider the following timeline



Unfortunately this solution meets the specification, even though there are multiple occurrences of `B0` between first and second `A0`. So we need to write one more constraint that is symmetric with respect to `B0`:

```
at A0.start: A0.end < next B0.start < next B0.end < next A0.start
at B0.start: B0.end < next A0.start < next A0.end < next B0.start
```

The previous undesired scenario is no longer a solution. The care needed to solve this simple problem, illustrates that correctly specifying a constraint can be a subtle matter. In fact there is one additional aspect of this solution that should be mentioned. Namely, we have implicitly assumed that we lift the requirements at the end of the planning horizon. The last occurrence of either `A0` or `B0` has no successor, hence the constraint is satisfied by default. Suppose `||` represents the end of the planning horizon on the following timeline.



The final `A0` has no `next A0` action and there is no subsequent `B0` so the constraint

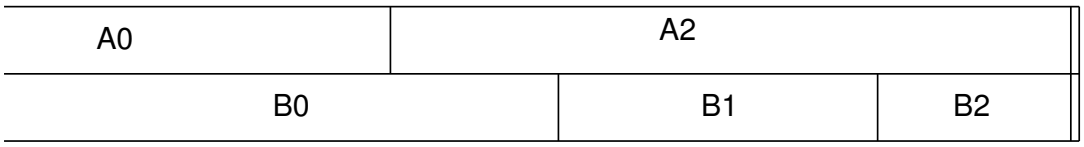
```
at A0.start: A0.end < B0.start < B0.end < next A0.start
```

is not strictly satisfied. Therefore, an exception must be made at the end of the horizon.

Note: we currently allow only one application of `next`. In other words, the following is not allowed: `next next A2`. Allowing multiple instances of `next` will increase the power of the language (and the difficulty of translation), but we are not convinced that this additional power is needed.

3.6 Vacuous solutions

Consider the Allen operator `A1 contains B1`. A constantly debated issue is whether the constraint can be satisfied by the following timeline



Because the Allen operator has the implicit quantifiers `FORALL A1: EXISTS B1: A1 contains B1`, this constraint can be vacuously met in case `A1` is never scheduled. Whether this is desirable or not is a recurring theme in the plan specification domain. A non-ambiguous semantics should be chosen for all these situations.

In ANMLite, the existence of vacuous solutions takes another dimension in the presence of flexible `at` expressions. For example, an equivalent ANMLite specification for the Allen operator `A1 contains B1` is the following:

```
A1.start < B1.start < B1.end < A1.end
```

There could be three types of vacuous solutions, depending on whether `A1` is never scheduled, `B1` is never scheduled, or both. If the `at` expression is associated with `A1`, then there exists a vacuous solution when `A1` is never scheduled, even if `B1` is. If, on the other hand, the `at` expression is associated with `B1`, then no such vacuous solution exists, because if `B1` is scheduled, then the satisfaction of `A1.start < B1.start` immediately requires the existence of a preceding `A1` to `B1`.

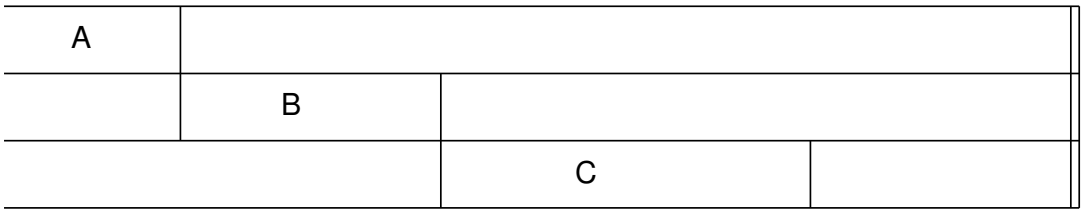
The *implementation* of a verifier by means of a model checker further adds to the complexity of this discussion, because `A1.start < B1.start` could be satisfied not only by the actual scheduling of `A1`, but also by the choice of initializing the variable associated to `A1.start` in the model. Usually, an action that has not been scheduled yet has a distinctive value which is not in the range of the planning horizon. The natural choices are $-\infty$ or ∞. If a negative initial value is chosen, `A1.start < B1.start` is always satisfied when `A1` is not (yet) scheduled before `B1.start`. A positive initial value will rule this situation out. But, of course, the situation is reversed when requiring the symmetric case: `A1.start > B1.start`.

In conclusion, the existence of vacuous solutions is influenced by three factors: the reference point, the implicit quantifiers, and the initialization of variables.

We have added the chained constraint syntax as an alternative to the conjunctive form. For the existence of vacuous solutions the two formats are not equivalent. Consider the following constraint:

```
B.start = A.end < B.end = C.start
```

which is satisfied by the following:



The semantics for this constraint is

```
FORALL C: EXISTS A,B: at C.start: B.start = A.end < B.end = C.start
```

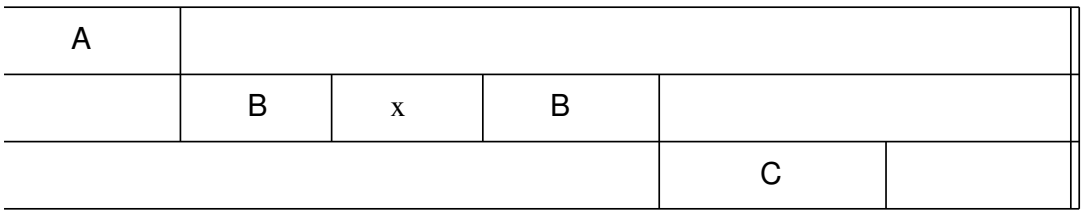
The last term is universally quantified and the other terms are existentially quantified. Therefore if `C` does not execute, the constraint is satisfied, but if `C` does execute, then there must also be executions of `A` and `B` as well. If one breaks up the above constraint into two separate constraints, i.e.,

```
B.start = A.end
B.end = C.start
```

the following defines the meaning:

```
FORALL A: EXISTS B: at A.end: B.start = A.end
FORALL C: EXISTS B: at C.start: B.end = C.start
```

Note that the split version allows the following timeline:



whereas the first version does not. It should be noted that an equality within a constraint is not an equivalence relation, except with respect to the timepoints. Because of the default reference points and default quantifier rules, the timelines defined by `A = B` can be different than those defined by `B = A`. For example,

```
A.end = B.start
B.end = C.start
```

has the following meaning:

```
FORALL B: EXISTS A: at B.start: A.end = B.start
FORALL C: EXISTS B: at C.start: B.end = C.start
```

so that the diagram above with two Bs separated by an x action does not satisfy it. The second B does not have an associated A for which the first constraint is satisifed.

In the future, we would like to look at the implications of eliminating all vacuous solutions or restricting them. Many times vacuous solutions are not desirable solutions at all. It is very easy for the writer of a domain specification to add a constraint with the expectation that this constraint will be required to happen. But if the planner finds a solution that avoids the implicit FORALL variable of the constraint, then it can ignore the stated constraint. In a safety-critical application, this could result in some important action being omitted or an important constraint not being satisfied. Of course, the domain specification can be strengthened by explicitly listing every action that must execute. But, it is not always intuitively obvious that this needs to be done. We believe that the clarity of a specification language is very important. It would be interesting to investigate whether the clarity of ANMLite would be increased by eliminating vacuous solutions or at least making it very clear where vacuous solutions are allowed.

3.7 Summary of Constraint Semantics

There are two major issues that need to be resolved when interpreting a constraint in ANMLite:

- Determination of the time point from which the current and next instances of an action can be disambiguated.
- Determination of which actions are universally quantified and which ones are existentially quantified.

These issues are orthogonal and hence the most general solution allows an independent specification of them. The first issue is handled by the at expression. If the at expression is omitted, the last term determines the time point. For example, given

```
A.end < B.end
```

the reference time point is B.end. All actions that are active at the refererence point are current. If an action is not scheduled at the reference point, the last instance is the current one and the future one is next.

The second issue is handled by a syntactic convention, namely, that the last term in the chain of inequalities determines the universally quantified action. For example, if we have

```
A.start < B.end < C.start < W.end
```

then the universally quantified action is W and the actions A, B, and C are existentially quantified:

```
FORALL W: EXISTS A,B,C: A.start < B.end < C.start < W.end
```

This choice is justified by the way the constraint checking has to performed (efficiently) in the SAL models (see Section 5). The other alternative, of attaching the universal quantifier to the first term, is equally valid from the theoretical point of view.

Also note that these conventions depend upon the restriction that all of the operators must be either `<`, `<=`, or `=`[3].

If the last term contains a `next` operator:

```
at C.start: A.start < B.end < C.start < next B.end
```

then the existence of two instances of B (current and next) are assumed. This constraint is interpreted as follows

```
FORALL B: (EXISTS B': B = next B' =>
                      (EXISTS A,C: at C.start:
                          A.start < B'.end < C.start < B.end))
```

It should be noted that inconsistent expressions can easily be created:

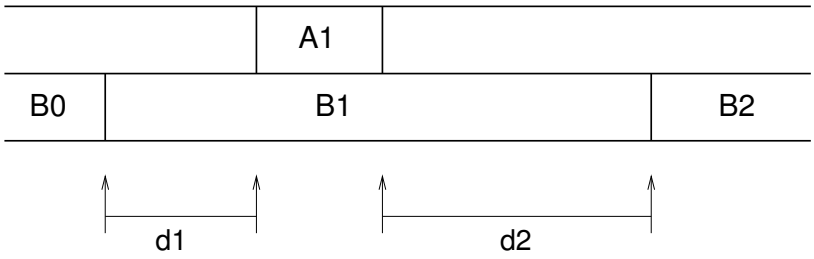
```
A.end < A.start
```

Such expressions are unsolvable and produce plan specifications for which there are no solutions.

3.8 Some More Illustrative Examples

3.8.1 Example: Spread-out Contains

Suppose we want a `B1 contains A1` that is also time-constrained:



We want `d1 > 10` and `d2 > 20`. We do this as follows

[3]We could generalize and also allow sequences of `>`, `>=`, `=` operators and then make the first syntactic term be the universally quantified action. We believe the added flexibility is not worth the potential confusion that could arise from trying to remember the quantification rule. Alternatively, we could add an explicit quantifier to the constraint expression, e.g. `FORALL W: A.start < B.end < C.start < W.end`, but we would then either have to make this mandatory or define a default. At this point we prefer not to add an explicit quantifier to the language.

```
at B1.start: B0.start + 10 < next A1.start < next A1.end < B1.end - 20
```

or more concisely as

```
B0.start + 10 < A1.start < A1.end < B1.end - 20
```

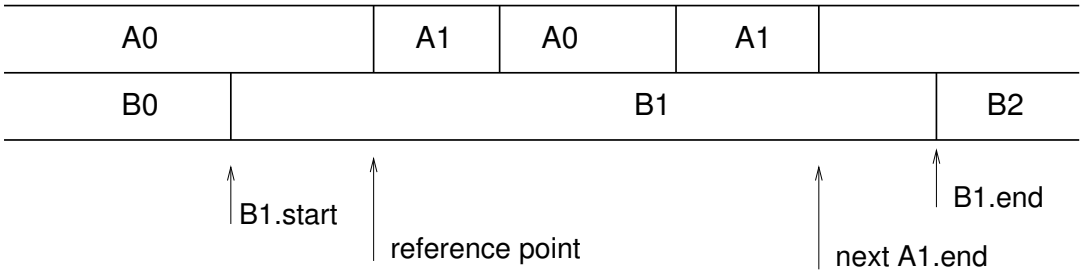
where the default reference point is at `B1.end`.

3.8.2 Example: Stretch It Out

Suppose we want each `B1` action to contain at least two instances of `A1`. This can be achieved with one chained constraint as follows:

```
at A1.start: B1.start < A1.start < next A1.end < B1.end
```

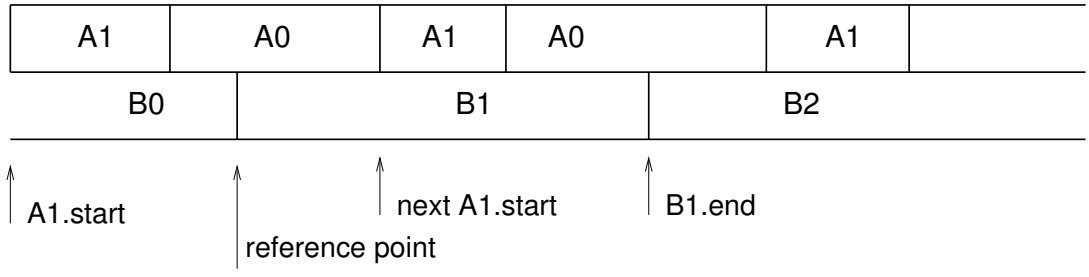
Here is an example solution:



Note that the constraint

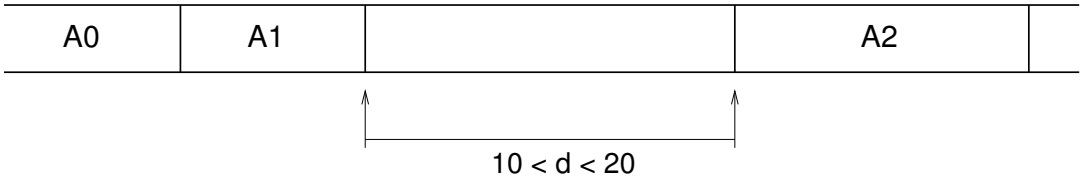
```
at B1.start: B1.start < A1.start < next A1.end < B1.end
```

has no solution. Consider the following diagram to see this:



3.8.3 Example: Forcing a Future Action to Occur Within a Time Interval

Suppose we wish to constrain action A2 to always be scheduled after action A1 and that this must occur between 10 and 20 time units after the end of A1 as illustrated below



which can be written in ANMLite syntax as

```
at A2.start: A1.end + 10 < A2.start < A1.end + 20
```

3.8.4 Delayed Initiation

Suppose that you want to insure that action `A1` does not begin until after 15 time units. The following constraint

```
15 < A1.start
```

accomplishes this. Note that it is more difficult to specify something like "the final action on the timeline is A3 and it starts after 70 time units". If `A3` is not a repetitive action, then this can be accomplished by

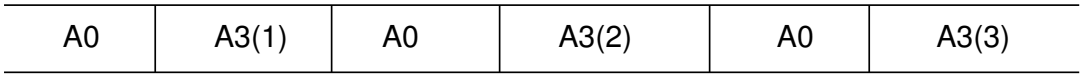
```
CONSTRAINTS

   70 < A3.start

GOALS

   A.A3
```

But what if `A3` is a repetitive task and we just want the last instance of `A3` to start after 70? We could use a parameter and count the instances of `A3` as follows:



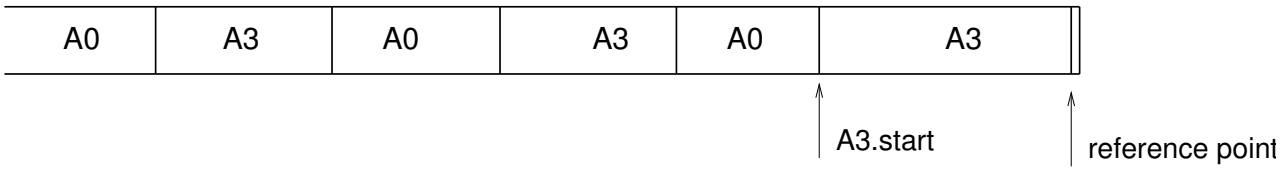
We could then extend the constraint language to allow parameters and constrain a specific instance, say 4, to occur after a specified time as follows

```
70 < A3(4).start
```

But this still does not solve the stated problem. Perhaps we could extend the `at` expression so that it could refer to the horizon beginning or end. Then we could write

```
at end: 70 < A3.start
```

This will work if `A3` is the goal state



3.9 Mandatory `at` Expressions

We have explored the possibility of not having a default `at` reference time point. If there are no cycles in the transition graph then each action can only occur once. In this case, the `next` syntax is not needed and hence there is no need to specify a reference time point. But suppose the user makes a mistake and specifies a constraint without a reference timepoint and there is a cycle in the graph. The translator would need to detect this situation and report an error. This will require a traversal of the graph to see if a cycle is present. This is relatively straight-forward in the absence of parameters. But with parameters, it is possible for there to be cycles for only certain parameter combinations. So what might appear to be a cycle situation, may really be unreachable. If `at` expressions are made optional and there is no default, it seems prudent to make the cycle check in the absence of parameters. This is clearly conservative and will rule out some cases where the `at` expression can be omitted, but the implementation code is much simpler. But, due to the difficulties associated with this approach, we have chosen to define a default reference point whenever an `at` expression is not present.

3.10 Convenient Macros

The ANMLite language can be extended with some macros which emulate some of the Allen operators. For example, the Allen operator `A contains B` can be emulated by

```
A.start < B.start < B.end < A.end
```

We can introduce the notation

```
CONSTRAINTS
    A contains B
```

which is then automatically expanded into

```
A.start < B.start < B.end < A.end
```

Similarly

```
A meets B
```

can automatically be translated into

```
B.start = A.end
```

and

```
A met_by B
```

can automatically be translated into

```
B.end = A.start
```

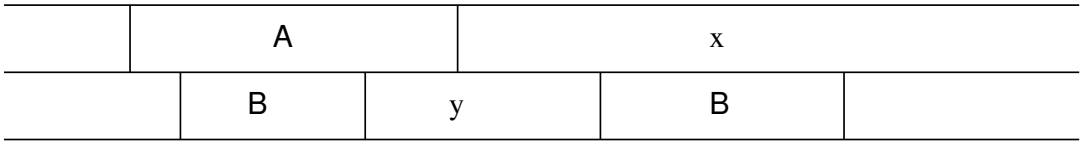
Some of the Allen operators cause some difficulty. Consider `B contained_by A`. At first we might suppose that the same constraint

```
A.start < B.start < B.end < A.end
```

will suffice. But, the semantics of the Allen operator mandates that if `B` executes, then it must be contained by an `A`. In other words, there is an implicit universal quantification for `B` and an implicit existential quantification for `A`, i.e. `FORALL B: EXISTS A: B contained_by A`. However, the semantics of the constraint: `A.start < B.start < B.end < A.end` is

```
FORALL A: at A.end: A.start < B.start < B.end < A.end
```

which is satisfied by



It appears that there is no convenient way to create a `contained_by` macro that exactly matches the semantics of the corresponding Allen Operation using the constructs that we have defined. To obtain exactly the same solutions would require something new like

```
FORALL B: at A.end: A.start < B.start < B.end < A.end
```

Another possibility would be to attach the default `FORALL` to the `at` or to create a special `at!` where this is done:

```
at! B.end: A.start < B.start < B.end < A.end
```

But, we are not sure whether this is a wise extension of the language. Perhaps we should not seek to duplicate the Allen Operation Semantics? Often vacuous solutions are not desirable solutions. It is very easy for the writer of a domain specification to write a constraint expecting that this would insure that the described situation will actually happen. But the constraint can be avoided altogether by the planner by finding a solution where the `FORALL` variable is not scheduled at all. It would be interesting to study the implications of eliminating all or some of the vacuous solutions in ANMLite. See section 3.6 for more information about this issue.

4 Condition and Effects

We are currently experimenting with condition and effects clauses within the definition of an action. This is motivated by similar clauses in the ANML language. However, we have introduced these constructs with some reluctance because they operate through use of global variables. Global variables are usually an attribute of programming languages rather than specification languages. Although this makes it possible to express things in the style that programmers are familiar with, it ultimately leads to a less abstract specification, and hence a more difficult one to formally verify. Nevertheless, one of the main purposes of this work is to develop verification methods suitable for verifying properties of the ANML language under development at NASA Ames.

We augment the body of an action definition, with a `condition` clause and an `effects` clause as illustrated below:

```
A1: [7,_] {
  condition: start > vf + 10;
  effect: vf := end;
}
```

where `vf` is a variable defined in a preceding `VARIABLES` section:

```
VARIABLES
    vf: TM_rng = 0;
```

The keyword `start` refers to the time that the action started execution and the keyword `end` refers to the time that the action finishes execution. The `condition` statement represents a constraint that must hold before the action is allowed to execute. The `effect` statement results in the update of the value of variables when the task terminates. The ANML language is more general in that it allows an arbitrary predicate and the specification of a time interval over which this predicate must hold. This time interval can be in the future, so very complex behaviors can be specified with that construct. However, we have restricted our effects section to be just a sequence of variable updates. The `condition` statement in `A1` above prevents the initiation of `A1` until the time is `vf + 10`. Initially `vf` is 0, so the first execution of `A1` is delayed until after time `10`. At the termination of `A1`, the `effect` statement updates the value of the variable `vf` to the termination time of `A1`. The net effect on subsequent executions of `A1` is to make sure that they are `10` time units apart.

The use of `condition` and `effect` statements is to introduce an algorithmic flavor to the language. If the updates to the variables are in a very restricted and controlled manner, then the intended meaning can be easy to discern. But if variables are updated in multiple places in different actions, then we have all of the subtleties of concurrent programming to deal with. It seems to us that these condition/effect statements can easily lead to complex specifications that are difficult to understand and even more difficult to verify. The more general ANML constructs are even more dangerous.

Note that the above specification can be accomplished more naturally using the following constraints

```
    A1.start > 10
    at A1.start: next A1.start > A1.end + 10
```

The advantage of this latter approach is that the specification is abstract and not algorithmic.

5 Translating ANMLite to SAL

Although using a model checker might not be the most efficient means of finding a solution to a planning problem, building a translator has provided a sanity check on the meaning of the language constructs.

5.1 Simple Example

We will begin our look at the technique for translating ANMLite to SAL with a very simple two timeline example:

```
PLAN ex1

TIMELINE A
ACTIONS

    A0: [2,_]
    A1
    A2

TRANSITIONS

    A0 -> A1 -> A2

END A


TIMELINE B
ACTIONS

    B0: [2,_]
    B1: [1,10]

TRANSITIONS

    B0 -> B1

END B
```

```
INITIAL-STATE

|-> A.A0
|-> B.B0

GOALS

  A.A2
  B.B1

END ex1
```

Timeline `A` has three actions `A0`, `A1`, and `A2` with only one possible sequence. The duration of action `A0` is at least 2 time units, while the other two actions have no restrictions on their duration. Similarly, timeline `B` has two actions `B0` and `B1`, with only one possible sequence. `B0` has a duration of at least 2 time units, while `B1` takes between 1 and 10 units.

Corresponding to these actions, the following types are generated

```
A_actions: TYPE = {A0, A1, A2, A_null};

B_actions: TYPE = {B0, B1, B_null};
```

In addition to the declared actions, a null state is created for each of the timelines. There are two purposes for these extra states:

- They provide a means for the completion of an action when the action has no successor.
- They provide a convenient mechanism for recording when a goal state has been reached on each timeline. As shown below, a transition from a timeline's goal state to the null state is generated by the translator.

The generated SAL model will consist of three modules:

- Module `A_m` which corresponds to timeline `A`.
- Module `B_m` which corresponds to timeline `B`.
- Module `Clock` which advances time.

5.2 Multiple variables

If there are multiple variables of a timeline, say

```
VARIABLES
    t1,t2: A
```

then a variable id type is generated,

```
A_ids: TYPE = {t1,t2};
```

and the module `A_m` is parametrized with the variable id

```
A_m[i: A_ids]  : MODULE =
```

Furthermore, since each instance of the timeline is a separate module, all the local and global variables in the parametrized module have to be arrays. For example, a non-parametrized module `A_m` might include a variable for `A0_start`

```
GLOBAL
  A0_start: TM_rng;
```

The parametrized version has to be

```
GLOBAL
  A0_start: ARRAY A_ids OF TM_rng;
```

This way, the start of `A0` for instance `t1` is referred to as `A0_start[t1]`.

5.3 Modeling Time

Time is governed by the generic clock module. We have experimented with various implementations of this module. The most straightforward approach is to have the clock module increment the current time by one time unit at each step. This approach is very simple but is not scalable, because the system would traverse a very large number of states that are identical with the exception of the clock value. This state explosion problem is exacerbated by problems with large planning horizons. A possible alleviation of problem is to allow the clock to advance by larger amounts. However, this still does not rule out the traversal of multiple states in an interval of time when nothing interesting happens (from the point of view of action change). The best solution in this case is to use the concept of timeouts [6] that model the *event driven* clocks. In this approach, each timeline maintains a future clock value where an event is scheduled to occur, and time jumps directly to the next interesting event. The timeouts are stored in an array of timepoints and the clock module determines the next (minimum value in the future) timeout.

The structure of the module generated for a timeline `A` is:

```
A_m : MODULE =
BEGIN
  INPUT

  OUTPUT

  GLOBAL
```

```
  LOCAL

  INITIALIZATION

  TRANSITION
[
    A0_to_A1:     %% A0 -> A1

  []
    A1_to_A2:     %% A1 -> A2

  []
    A2_to_A_null:     %% A2 -> A_null

  ]
END; %% A
```

Each of these sections is used in the translation:

- `INPUT` is used to select values for parameters of actions.
- `OUTPUT` is used to store timeout values for the clocking mechanism
- `GLOBAL` is used to hold state values and their current parameter values.
- `LOCAL` is used to hold the start time of the currently scheduled action.
- `INITIALIZATION` is used to set initial values.
- `TRANSITION` specifies rules for transitioning from one action to another.

The three modules will be asynchronously composed. In SAL this is specified as follows[4]

```
System: MODULE = A_m [] B_m [] Clock;
```

The SAL tool links the variables named in the `GLOBAL` and `INPUT` sections together. In other words, variables with the same name are equated even though they are specified in different modules.

The SAL model checker will be used to search through all possible sequences of actions on the timelines to find sequences which satisfy all of the constraints specified in the ANMLite model. These constraints fall into three broad categories:

- Timing constraints that impact durations and start/stop times of actions.

[4]It is actually slightly more complicated than this because of the need to coordinate the timeout variables of the modules. This will be discussed later.

- Simple relationships between `start` and `end` variables
- Constraints defined through `condition` and `effect` statements.

The search is started at time 0 and proceeds forward in time until the planning horizon is reached. The stop time is specified via a constant as follows:

```
MAX_TM: int = 30;
TM_rng: TYPE = [0 .. MAX_TM+1];
```

The progress of the "clock" is controlled by the `clock` module:

```
Clock: MODULE =
BEGIN
 INPUT timeout: TIMEOUT_ARRAY
 OUTPUT time: TM_rng
 INITIALIZATION
    time = 0;
 TRANSITION
   [
    time_elapses:
      (EXISTS (i: timeline): time < timeout[i])
        AND (FORALL (i: timeline): time /= timeout[i])
      -->
        time' IN { t: TM_rng | is_min(timeout, t) }
   ]
END;
```

5.4 Model Variables

The `GLOBAL` sections of all of the timeline modules contain variables which record the action that is scheduled during the current time:

```
GLOBAL
  A0_start: TM_rng,
  B0_start: TM_rng,
  B1_start: TM_rng,
  B_state: B_actions,
  A_state: A_actions,
```

The `_state` variables contain the current action and the `_start` and `_end` variables contain the start and end times of the actions.

The durations of the actions are controlled by the use of the `_start` variables. Whenever an action is initiated, the initiation time is stored in this variable. The durations of an action will be controlled through this variable. This will be explained more carefully when we discuss the transitions section.

The initial actions on a timeline can be specified as follows:

```
INITIAL-STATE
|-> A.A0
|-> B.B0
```

If a timeline's initial state is not specified, then the model checker will explore all possible start states. The above initialization results in

```
INITIALIZATION
  A_state = A0;
  A0_start = 0;
```

in the `A_m` module and

```
INITIALIZATION
  B_state = B0;
  B0_start = 0;
  B1_start = MAX_TM+1;
```

in the `B_m` module.

5.5 Transitions

The ANMLite `TRANSITIONS` section is the major focus of the translation process. The SAL `TRANSITIONS` section is constructed from this part of the ANMLite model. For example, the following

```
TRANSITIONS

    A0 -> A1 -> A2
```

is translated into

```
TRANSITION
[
  A_starts:
     A_state = A0
  -->
     timeout' IN {t:TM_rng | t > time}
[]
  A0_to_A1:     %% A0 -> A1
  A_state = A0
  AND time >= A0_start + 2
  -->
    A_state' = A1;
    timeout' IN {t:TM_rng | t > time}
```

```
[]
  A1_to_A2:     %% A1 -> A2
  A_state = A1
  -->
    A_state' = A2;
    timeout' IN {t:TM_rng | t > time}
[]
  A2_to_A_null:     %% A2 -> A_null
  A_state = A2
  -->
    A_state' = A_null;
    timeout' = MAX_TM+1;

]
```

When a transition occurs, an action is completed and another transition is initiated. No empty time slots are allowed. The `TRANSITIONS` section defines three transitions which are labeled as follows:

```
A0_to_A1:       %% A0 -> A1
A1_to_A2:       %% A1 -> A2
A2_to_A_null:   %% A2 -> A_null
```

The first transition is guarded by the following expression:

```
  A_state = A0
  AND time >= A0_start + 2
```

The first conjunct insures that this transition only applies when the current action on the timeline is `A0` and the second conjunct insures that the duration of the action is at least 2 time units. This corresponds to the fact that `A0` was declared as `A0: [2,_]`. The expressions after the `-->` specify that the new state is `A1`. Note that start times are not maintained for `A1` and `A2` because no constraints use them.

The ANMLite `GOALS` statement

```
GOALS
  A.A2
  B.B1
```

lists two actions that need to be reached (where the default meaning of the expression is the logical conjunction of the terms). This statement is translated into the following SAL specification:

```
 sched_sys: THEOREM
    System |- AG(NOT(A_state = A_null AND B_state = B_null));
```

Since the "null" states can only be reached from the goal states (i.e., A2 and B1), these efficiently record the fact that the appropriate goal has been reached on each timeline. Note that the ANMLite goal statement has been negated. Therefore when the model checker is instructed to establish the property, any counterexample provided by SAL will serve as a feasible realization of the plan.

The complete generated SAL model is

```
ex1: CONTEXT =
BEGIN

  MAX_TM: int = 30;
  TM_rng: TYPE = [0 .. MAX_TM+1];

  A_actions: TYPE = {A0,
                     A1,
                     A2,
                     A_null};

  B_actions: TYPE = {B0,
                     B1,
                     B_null};

  timeline: TYPE = {A,B};

  TIMEOUT_ARRAY: TYPE = ARRAY timeline OF TM_rng;

  is_min(x: TIMEOUT_ARRAY, t: TM_rng): bool =
     t >= 0
      AND (FORALL (i: timeline): t <= x[i])
      AND (EXISTS (i: timeline): t = x[i]);

  A_m  : MODULE =
  BEGIN
    INPUT
      time: TM_rng
    OUTPUT
       timeout: TM_rng
    GLOBAL
      A0_start: TM_rng,
      B0_start: TM_rng,
      B1_start: TM_rng,
      B_state: B_actions,
      A_state: A_actions
```

```
  INITIALIZATION
    A_state = A0;
    A0_start = 0;
  TRANSITION
  [
    A_starts:
       A_state = A0
    -->
       timeout' IN {t:TM_rng | t > time}

  []
    A0_to_A1:     %% A0 -> A1
    A_state = A0
    AND time >= A0_start + 2
    -->
      A_state' = A1;
      timeout' IN {t:TM_rng | t > time}
  []
    A1_to_A2:     %% A1 -> A2
    A_state = A1
    -->
      A_state' = A2;
      timeout' IN {t:TM_rng | t > time}
  []
    A2_to_A_null:     %% A2 -> A_null
    A_state = A2
    -->
      A_state' = A_null;
      timeout' = MAX_TM+1;
  ]
END; %% A

B_m  : MODULE =
BEGIN
  INPUT
    time: TM_rng
  OUTPUT
     timeout: TM_rng
  GLOBAL
    A0_start: TM_rng,
    B0_start: TM_rng,
    B1_start: TM_rng,
```

```
    A_state: A_actions,
    B_state: B_actions
  INITIALIZATION
    B_state = B0;
    B0_start = 0;
    B1_start = MAX_TM+1;
  TRANSITION
  [
    B0_to_B1:     %% B0 -> B1
    B_state = B0
    AND time >= B0_start + 2
    -->
      B_state' = B1;
      B1_start' = time;
      timeout' IN {t:TM_rng | t > time}
  []
    B1_to_B_null:     %% B1 -> B_null
    B_state = B1
    AND time >= B1_start + 1
    AND time <= B1_start + 10
    -->
      B_state' = B_null;
      timeout' = MAX_TM+1;
  ]
END; %% B


Clock: MODULE =
BEGIN
 INPUT timeout: TIMEOUT_ARRAY
 OUTPUT time: TM_rng
 INITIALIZATION
    time = 0;
 TRANSITION
   [
    time_elapses:
      (EXISTS (i: timeline): time < timeout[i])
        AND (FORALL (i: timeline): time /= timeout[i])
      -->
        time' IN { t: TM_rng | is_min(timeout, t) }
   ]
END;
```

```
  System: MODULE = (WITH OUTPUT timeout: TIMEOUT_ARRAY
                      (RENAME timeout TO timeout[A] IN A_m)  []
                      (RENAME timeout TO timeout[B] IN B_m)
                    ) [] Clock;

  sched_sys: THEOREM
    System |- AG(NOT(TRUE
                           AND A_state = A_null
                           AND B_state = B_null
                           ));

END %% ex1
```

5.6 Translating Constraints

There are major conceptual differences between *specifying* constraints and *checking* constraints that need to be reconciled. In principle, the specification is declarative by nature and the modeler usually looks "forward" in time in expressing what needs to happen in order for the plan to complete. The checking of the plan is operational by nature, because `start` and `end` variables are assigned values as they occur, hence testing that a constraint is valid cannot be performed until the last timepoint has occurred. Therefore, in the checking of the constraints the modeler has to look "backwards" in time.

For example, the constraint `A.start < B.end < C.start` cannot be established when `A` starts. Even if `B` has not ended yet, its relationship to the start of `C` cannot be established.

The mechanism of checking constraints with a model checker is based on assigning and updating the values of timeline `state` and each action `start` and `end` variables. This is performed at the timepoints when a timeline transitions from one action to another, according to the `TRANSTIONS` section.

Repetitive actions require special care, as multiple occurrences of the same actions will overwrite the values of the corresponding `start` and `end` variables, so only the most recent one is actually available (and possibly the previous occurrence, given that we allow the `next` qualifier).

For example, if there is a transition `A1 -> A2` on timeline `A`, the following updates are necessary:

- `A_state' = A2`
- `A1_end' = time`
- `A2_start' = time`

A constraint is, in principle, applicable to all the transitions that affect the variables present in the constraint expression. That is, a `start` variable is relevant to *entering* an

action, while the `end` variable is relevant to *exiting* an action. Transition guards are generated for the events that are involved.

The general approach of translating constraints into transition guards consists of determining the last timepoint in the chain and substituting that term with the value of the system variable `time`. For example, in the constraint

```
A1.start + 4 < B1.start <  C1.end
```

the last timepoint is `C1_end`. The transitions of relevance to this timepoint are from a predecessor of `C1` to `C1`, i.e., *entering* `C1`. For simplicity, in the following, assume there is only one successor/predecessor of an action, according to its index ($\texttt{X}_n$ precedes $\texttt{X}_{n+1}$, for any action `X` and any index n). Then, the above constraint will results in a transition guard for `C0_to_C1`:

```
TRANSITION
 [
   C0_to_C1:
   C_state = C0
   AND time >= C0_start + d %% any duration constraint here
   AND (A1_start + 4 < B1_start) AND (B1_start < time)
   -->
     C_state' = C1;
     C1_start' = time;
     timeout' IN {t:TM_rng | t > time}
 []
 ...
```

A chained constraint can be broken down into all of its pairs, e.g., treat `A1.start + 4 < B1.start` and `B1.start < C1.end` separately. However, this simple decomposition does not work in the presence of a repetitive action, where both the current and the next instance are involved simultaneously. For example,

```
at B1.end: A1.end < B1.start < B1.end < next A1.start
at A1.end: B1.end < A1.start < A1.end < next B1.start
```

enforcing that `A1.end < B1.start` separately from `B1.end < next A1.start` results in an unwanted scenario: each `B1` is now required to end before the *next* instance of `A1` which practically delays the scheduling of `A1` indefinitely.

5.7 Current and next variables

When both the current and next instances are involved in a constraint, special care is required. Since we check constraints "backwards", the two variables needed have to be labeled `A1_start` and `prev_A1_start`. The current instance is the last in time, so it corresponds to `next A1`, while the start of earlier `A1` is now denoted by `prev_A1_start`, which could be

a little counter-intuitive. When no repetitive instances are involved, no such variable name transformations are necessary.

The generated guard for the first constraint in the above example is

```
TRANSITION
 [
   A0_to_A1:
   A_state = A0
   AND time >= A0_start + d  %% any duration constraint here
   AND (prev_A1_end < B1_start)
   AND (B1_start < B2_end)
   AND (B1_end < time)
   -->
     A_state' = A1;
     A1_start' = time;
     prev_A1_start' = A1_start;
     timeout' IN {t:TM_rng | t > time}
 []
 ...
```

Note that the value of the `prev_` variables are also maintained. A special case is the first occurrence of a repetitive action, which has *no* predecessor. Hence, the complete guard is actually:

```
TRANSITION
 [
   A0_to_A1:
   A_state = A0
   AND time >= A0_start + d      %% any duration constraint here
   (prev_A1_end == MAX_TM+1) OR  %% first occurrence is a special case
   ((prev_A1_end < B1_start) AND (B1_start < B2_end) AND (B1_end < time))
   -->
     A_state' = A1;
     A1_start' = time;
     prev_A1_start' = A1_start;
     timeout' IN {t:TM_rng | t > time}
 []
 ...
```

Further processing is required in case the constraint refers to a particular timeline instance. For example, the constraint

```
at t2.B1.end: t1.A1.end < t2.B1.start < t2.B1.end < next t1.A1.start
```

generates the following transition guard:

```
TRANSITION
 [
   A0_to_A1:
   A_state[i] = A0
   AND time >= A0_start[i] + d      %% any duration constraint here
   (prev_A1_end[i] == MAX_TM+1) OR  %% first occurrence is a special case
   (i==t1 IMPLIES
   ((prev_A1_end[i] < B1_start[t2]) AND
     (B1_start[t2]  < B2_end[t2]) AND (B1_end[t2] < time)))
   -->
     A_state' = A1;
     A1_start' = time;
     prev_A1_start' = A1_start;
     timeout' IN {t:TM_rng | t > time}
 []
 ...
```

6 The Crew Activity Planning Model in ANMLite

The Crew Activity Planning Model is an example problem taken from the ANML distribution. The task is to plan the basic daily routines for several crew members over a period of four days. The activities include, besides physiologic states (sleeping, meals), routine activities, such as regular filter changes, medical conferences, and some payload operations. The ANML specification of this problem is described below.

The definition of an action

```
action pre_sleep {
    condition over all : fw.state == not_fasting();

}
```

is captured in the following ANMLite as follows:

```
not_fasting.start < pre_sleep.start
                  < pre_sleep.end < not_fasting.end
```

The three condition statements in the definition of `meal`

```
action meal {
    condition over all : fw.state == not_fasting();

    // keep separation between meals >= 4 hrs
    condition over [start-240 start] : fw.state == fasting();
    condition over [end end+240] : fw.state == fasting();
```

```
}
```

can be condensed into

```
not_fasting.start + 240 < meal.start < meal.end < not_fasting.end - 240
```

The following ANML specification uses an `effect` statement to specify that the next filter change happens within the next 24 to 48 hours

```
action change_filter(FilterState fs) {
    change over all : fs -> {changed,changing,changed};

    // make sure next filter change happens
    //within the next 24 to 48 hours
    effect in [end+1440 end+2880] : fs == changing;

}
```

This specification relies on the use of an auxiliary variable `fs`, which has no other role than to provide a way of saying that something must occur in the future. Compare the opaqueness of the ANML specification with the elegance of the ANMLite equivalent:

```
at change_filter.end:
 change_filter.end + 1440 < next change_filter.start
                          < change_filter.end + 2880
```

Notice that there is no need for an auxiliary variable and it is far easier to understand. In fact, it would be very difficult to discern the intended meaning of the ANML specification without the presence of the comment.

The crew planning problem is centered around the idea that there is a set of jobs that need to be done by the crew but the allocation of crew to the jobs is left to the planner. The set of jobs is specified using the predicate

```
predicate payload_act_completed(
  int id,
  string desc,
  int length,
  int priority,
  int dueDate,
  int usesComms,
  int isPhysicallyRestraining);
```

The only parameter that is referenced in the current model is dueDate. So at this point in the development of the model, the other parameters are just stubs. We need to augment the ANMLite language to conveniently handle this aspect of the problem. In particular, we need variables which are declared to be arrays. For example:

```
VARIABLES

   payload_act: ARRAY [1..numacts] OF RECORD
                                      completed: boolean,
                                      dueTime: TM_rng
                                      END
                = (false,1440; false,1440; false,1440; false,60; false,60)
```

The first variable records when a payload activity is completed and the second holds the deadline for a particular activity (note that it is a constant). The `payload_activity` action is then defined as follows

```
   pact(id: 1..numacts; desc:StringData;
      length, priority, dueDate, usesComms, isPhysicallyRestraining: myint)
   {
      effect: payload_act[id].completed = true;
   }
```

The following constraint is added to the constraints section

```
    at pact.start:  pact(id, ...).end < payload_act[id].dueTime;
```

and the following goals are added to the goals section:

```
      FORALL myid: payload_act[myid].completed = true;
```

The Crew Planning problem raises another issue with its use of a decomposition construct. This construct represents a hierarchy. There is a high-level action (e.g., `daily_routine`) which "contains" a series of low-level actions, e.g., `post_sleep`, `dpc`, `meal`, etc., which have a partial ordering on them. To accomplish this in NDDL a separate timeline was created with a single action "`perform`" which contains all of the required crew actions. Since `perform` is a goal action, the FORALL-EXISTS semantics forces all of the actions which it contains to also occur.

But, if we use the ANMLite version of `contains`, e.g.,

```
      A contains B
```

namely

```
      A.start < B.start < B.end < A.end
```

we have to worry about the possibility of vacuous solutions. In other words, whether this constraint can be satisfied vacuously with B never executing. The current operational semantics provided by our translator can be summarized as follows: If `A` terminates, then `B` will have to have executed and terminated as well. This is true because the translator initializes the start/end variables to a virtually infinite value. Therefore the guard on the termination of `A` cannot be satisfied unless `B` has executed once and acceptable values are loaded into `B.start` and `B.end`.

So the semantics of the statement

```
A.start < B.start < B.end < A.end
```

can be thought of

```
FORALL A: EXISTS B: A.start < B.start < B.end < A.end
```

So if the last action is required to execute, e.g., because it is in a goal statement, the execution of the other actions in a chain is also required.

So we add a boolean variable

```
RPCM_done: boolean = false;
```

which is initialized to false and add a statement to the final `rec_t_loop` action:

```
rec_t_loop(loopCnt: myint): [ 6,6 ] {
    if loopCnt == 2 then RPCM_done = true;
}
```

and finally add the goal

```
GOAL
   RPCM_done;
```

Then the following constraints should constrain the appropriate sequence of actions:

```
Crew.rem_sleep_stat.end < Crew.rem_pow_src.start <
           Crew.rem_pow_src.end < Crew.replace_rpcm.start <
               Crew.replace_rpcm.end < Crew.place_power_source.start
                   Crew.place_power_source.end < Crew.ass_sl_stat.start

Crew.rec_t_loop.end < Crew.replace_rpcm.start

Crew.replace_rpcm.end  < Crew.rec_t_loop(2).start
```

7 Conclusion

This paper discusses several design issues of a simple language for specifying planning problems. We call that language ANMLite and it is aimed to be compact, expressive, and suitable for formal analysis. We illustrate with several examples that these objectives are not necessarily compatible and that trade-offs between expressiveness, simplicity, and clarity are often necessary. ANMLite is by no means a replacement for more expressive planning languages such as PDDL [7], NDDL, or ANML. However, the semantic issues that arise in this simple language strongly suggests that subtle semantic issues are probably present in these more complex languages as well. We would argue that a clear semantics for the planning languages is essential if planning systems are to be trusted for safety-critical applications.

We are not completely satisfied that we have arrived at the best choice for the definition of the `at` reference point or whether it would be better to define repetitive actions using a `prev` concept instead of a `next` concept. We are not sure whether we have selected the best defaults. Every option that we have explored so far results in some non-intuitive cases or ugliness. We also are concerned about language constructs which allow vacuous solutions. Because we were attempting to emulate the behavior of EUROPA 2, we have allowed these. But we would like to explore whether the elimination of some or all of the vacuous solutions will improve the clarity of the ANMLite specification language. We have sought to elaborate on these issues in this paper. This work suggests that the semantics of a planning language, even a simple one such as ANMLite, is not a simple effort. We strongly believe that a formal semantics is a prerequisite for the use of these languages in critical aerospace applications. The work presented in this paper is a first step in that direction.

Our efforts at using a model checker to solve planning problems have convinced us that this is not an efficient approach that scales well. But, building the translator has forced us to sharpen our thinking about the meaning of the language constructs. An alternative solution approach that we would like to pursue is to apply a constraint satisfaction solver such as Yices [8]. We believe that the translation of the ANMLite constraints into formulas that can be processed by a constraint solver would be fairly straight forward for the non-repetitive cases. However, the repetitive cases result in an indeterminate number of instances of an action. Constraint solvers need a fixed set. This problem can be addressed a manner analogous to bounded model checking. First, the satisfaction solver is given formulas that involve only one instance of an action. If the solver fails to find a solution, then second instances of repetitive actions can be introduced, and then three, and so on. This can be continued until a solution is found or a pre-specified quitting point is reached. A nice consequence of this approach is that a plan with minimal repetitions of an action will be discovered.

References

[1] E.Smith, David; Frank, Jeremy; and McGann, Conor: *The ANML Language*. NASA Ames Unpublished Report, Technical report, 2006.

[2] Allen, James F.; and Ferguson, George: *Actions and Events in Interval Temporal Logic*. University of Rochester, Technical Report TR521, 1994.

[3] de Moura, Leonardo; Owre, Sam; and Shankar, Natarajan: *The SAL Language Manual*. CSL Technical Report, Technical Report SRI-CSL-01-02, 2003. http://sal.csl.sri.com/documentation.shtml.

[4] Butler, Rick W.; and Muñoz, César A.: *An Abstract Plan Preparation Language*. NASA Langley, Report NASA/TM-2006-214518, NASA LaRC,Hampton VA 23681-2199, USA, 2006.

[5] Bedrax-Weiss, Tania; McGann, Conor; Bachmann, Andrew; Edington, Will; and Iatauro, Michael: *EUROPA-2: User and Contributor Guide*. NASA AmesResearch Center, Technical report, Moffett Field, CA, Feb 2005.

[6] Owre, Sam; and Shankar, Natarajan: *Formal Analysis Methods for Spacecraft Autonomy, Final Report*. CSL, Technical Report SRI-17625, 2007.

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

[8] de Moura, Leonardo; and Dutertre, Bruno: *Yices 1.0: An Efficient SMT Solver*. Technical report, 2006. Presented at SM_COMP'06, available at http://yices.csl.sri.com.

A The ANMLite Syntax

A.1 Timeline declarations

```
<anml_def>          ::= PLAN <identifier>
                        ( <type_decl> | <timeline_decl> | <constraints_decl> |
                          <vars_decl> )*
                        [ <inits_decl> ]
                        [ <goals_decl> ]
                        END <identifier>

<type_decl>         ::= TYPE ( <simple_type_decl> | <compound_type_decl> )

<simple_type_decl>::=  <identifier> = <type>

<compound_type_decl>::=  <identifier> <parameters> [ = <type> ]

<type>              ::= <basic_type> | <enumeration> | <interval> | <defined_type>

<basic_type>        ::=  INT | FLOAT |BOOL | STRING |

<enumeration>       ::=  [ <identifiers> ]

<identifiers>       ::=  <identifier> ( , <identifier> )*

<identifier>        ::=  <ID>

<interval>          ::=  [ <add_expression_or_nil> , <add_expression_or_nil> ]

<add_expression_or_nil> ::=  <additive_expression> | <nil>

<defined_type>      ::= <identifier> <arguments>

<arguments>         ::=  [ <strict_arguments> ]

<strict_arguments> ::= "("  <expression_or_nil> ( , <expression_or_nil> )* ")"

<expression_or_nil>::= <expression> | <nil>

<nil>                ::= "_"

<parameters>         ::=  [ strict_parameters() ]
```

```
<strict_parameters>::= "(" <parameter> ( ; <parameter> )* ")"

<parameter>        ::= <identifiers> ":" <type>

<timeline_decls>   ::= OBJTYPE <identifier> <parameters>
                       <actions_decl>
                       <transition_decl>
                       END <identifier>


<actions_decl>     ::= ACTIONS ( action_decl() )+

<action_decl>      ::= <identifier> <parameters> <duration_decl> [ action_body() ]

<duration_decl>    ::=  [ ":" <interval> ]

<transitions_decl> ::=  [ TRANSITIONS ( <transition_decl> )+ ]

<transition_decl>  ::=  <action>
                           ( ( -> <action> )+ ["-|"]  |
                               -> * "\" <action>
                           )

<action>           ::= <simple_action> |
                      ( "(" <simple_action> ( "|" <simple_action> )+ ")" )

<simple_action>    ::= <qualified_id> <arguments>

<start_end_var>    ::= [ NEXT ] <identifier> <start_end>

<start_end>        ::= ".start" | ".end"

<qualified_id>     ::= <ID> ( .<ID> )

<expression>       ::= ..........
```

A.2 Constraints

The proposed BNF for the constraints syntax includes (in)equations involving start/end of actions timepoints, *over* constraints, and quantifiers (a long awaited new feature!).

```
<constraints_decl>  ::= CONSTRAINTS ( <constraint_decl> )+
```

```
<constraint_decl>   ::= ( <at_formula> | <bool_formula> )

<at_formula>        ::= [ <at_expression>] <timepoint> <rel_op>
                        <timepoint> [ <plusinteger> ]

<timepoint>         ::= (<start_end_term> | <integer>)

<start_end_term>    ::= ( next ) <ID> <start_end> (<add_op> <integer>)

<add_op>            ::= +  |  -

<bool_formula>      ::= ( <simp_bool_formula> ("&&" | "||" | "->")+
                          <simp_bool_formula>
                          | "!" <simp_bool_formula>
                          | <simp_bool_formula>
                        )

<simp_bool_formula> ::=  <state_var> "==" <state> |
                         <state_var> "!=" <state>

<bin_logic_op>      ::= "&&" | "||" | "->"


<at_expression>     ::= at <qualified_id> (<start_end>) (<strict_arguments>) :
<state_var>         ::= <ID>
<state>             ::= <ID>

<inits_decl>        ::= INITIAL_STATE ( init_decl )+
<goals_decl>        ::= GOALS ( goal_decl )+
<goal_decl>         ::= <action>
<vars_decl>         ::= VARIABLES ( var_decl )+
<var_decl>          ::= <identifiers> <COLON> <type> ( = <integer>)
<init_decl>         ::= |-> <action>
```

A.3 Condition and Effect Statements

```
<action_body>       ::= "{" [ condition() ]  [ effect() ] "}"

<condition>         ::= "condition:"  <expression>
<effect>            ::= "effect:" <identifier> ":="  <expression>
```

B The Dock-Worker Robots Problem in ANMLite

The book *"Automated Planning"* by Ghallab, Nau and Traverso presents a planning problem that is used throughout the book. The Dock-Worker Robots problem is centered around a set of robots, cranes, and piles. Robots move containers from one location to another, while cranes move containers between piles and robots. The initial state is defined by allocating containers to piles. The final state is another distribution of containers on the piles. In the general case, there can be multiple piles at a location. Here we assume only one pile per location. The following is an ANMLite version of this problem:

```
 PLAN dwr

   TYPE location = [1,3]

   TYPE robot_ix = [1..2]

   TYPE container = {c1,c2,c3,c4,c5,c6,null}

   TYPE pile_ix = [1,3]

VARIABLES

   pile: ARRAY [pile_ix, 1..6] OF container = (4*6*null);
   top: ARRAY[pile_ix] OF [0..6] = (4*0);

   push(p,c) {
      top[p] = top[p] + 1;
      pile[p,top] = c;
   }

   pop(p): container {
      popv: container;
      if top[p] = 0 then popv = null;
      else
         popv = pile[p,top];
         top[p] = top[p] - 1;
      endif
      return popv;
   }


TIMELINE Nav ACTIONS
```

```
   At(x: location)

   Move(x,y: location) WITH
      y = x + 1 || y = x - 1
      %% can only move to adjacent position in 1 step

TRANSITIONS

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

END Nav

TIMELINE Robot ACTIONS

   Unloaded

   Loading(c: container)

   Loaded(c: container)

   Unloading(c: container)

TRANSITIONS

   Unloaded -> Loading(c) -> Loaded(c) -> Unloading(c) -> Unloaded


END Robot

TIMELINE Crane ACTIONS

   // need to reference the pile associated with me, e.g. Cr[2] or Cr[3]
   // need a way of referencing "my_location" or "my_pile"
   // or we can define a local stack variable here
   //
   // pile: ARRAY [1..6] OF container = (6*null);
   // top: [0..6] = 0;
   //
   // rather than as a two-dimensional global variable.
   // If I do it here, it is looking more like an object.

   Empty
```

```
   Take(c:container) {
         condition: top(my_pile) > 0
         effect: c = pop(my_pile);
   }

   Put(c:container) {
       condition: top(my_pile) <= 6;
       effect: push(my_pile,c);
   }

   Load_robot(c:container;  r: robot_ix)

   Unload_robot(c:container;  r: robot_ix)

   Holding(c: container)

TRANSITIONS

   Empty -> (Take(c) | Unload_robot(c,r)) -> Holding(c)

   Holding(c) ->( Put(c) | Load_robot(c,r) ) -> Empty

END Crane

VARIABLES

   Nv: ARRAY robot_ix OF Nav
   Rb: ARRAY robot_ix OF Robot
   Cr: ARRAY location OF Crane

CONSTRAINTS

      Cr[xx].Load_robot(c,rr)   <contained_by> Nv(rr).At(xx)
      Cr[xx].Load_robot(c,rr)   <meets>        Rb[rr].Loaded(c)
      Cr[xx].Unload_robot(c,rr) <contained_by> Nv(rr).At(xx)
      Cr[xx].Unload_robot(c)    <ends>         Robot.Loaded(c)

INITIAL-STATE

|-> Nv[1].At(1)
|-> Nv[2].At(2)
```

```
|-> Rb.Unloaded

|-> "some distribution of containers on the piles"

GOAL

    "another distribution of containers on the piles"

END dwr
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 11 - 2007	Technical Memorandum	

4. TITLE AND SUBTITLE
The ANMLite Language and Logic for Specifying Planning Problems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Butler, Ricky W.; Siminiceanu, Radu I.; and Muñoz, César A.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
699152.04.03.03.04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19405

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-2007-215088

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

We present the basic concepts of the ANMLite planning language. We discuss various aspects of specifying a plan in terms of constraints and checking the existence of a solution with the help of a model checker. The constructs of the ANMLite language have been kept as simple as possible in order to reduce complexity and simplify the verification problem. We illustrate the language with a specification of the space shuttle crew activity model that was constructed under the Spacecraft Autonomy for Vehicles and Habitats (SAVH) project. The main purpose of this study was to explore the implications of choosing a robust logic behind the specification of constraints, rather than simply proposing a new planning language.

15. SUBJECT TERMS
AI Planning; Artificial Intelligence; Formal Methods; Planning Language; Safety; Software 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	54	(301) 621-0390