NASA/TM-2007-215089

Solving the AI Planning + Scheduling Problem Using Model Checking via Automatic Translation From the Abstract Plan Preparation Language (APPL) to the Symbolic Analysis Laboratory (SAL)

Ricky W. Butler
Langley Research Center, Hampton, Virginia

César A. Muñoz and Radu I. Siminiceanu
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-2007-215089

Solving the AI Planning + Scheduling Problem Using Model Checking via Automatic Translation From the Abstract Plan Preparation Language (APPL) to the Symbolic Analysis Laboratory (SAL)

Ricky W. Butler
Langley Research Center, Hampton, Virginia

César A. Muñoz and Radu I. Siminiceanu
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 Introduction

The Abstract Plan Preparation Language (APPL) [2] is a planning language inspired by the New Domain Description Language (NDDL), a powerful planning language developed at NASA Ames [4] which has evolved from the Planning Domain Description Language (PDDL)[5][1]. APPL is built around the idea that not all constraints are alike in the specification of an AI planning problem. In particular, the APPL language provides a special notation for the actions which must be mutually exclusive on a particular timeline. This feature enables a more compact specification of actions and states that is similar to the specification of state transition systems.

Because the syntax and semantics of APPL are simpler than those of NDDL, APPL is more suitable for formal verification, static analysis, and automated test generation. However, the language is much less expressive than NDDL.

This paper describes the algorithms and techniques used to translate APPL to SAL[3]. This translator was developed under the Spacecraft Autonomy for Vehicles and Habitats (SAVH) project, whose purpose is to develop verification & validation technology that can enable the application of AI planning to man-rated space applications. The long-term goal of our research is to develop techniques to verify the correctness of the domain knowledge that is typically expressed in planning languages. The key idea is that once the planning domain knowledge has been captured in a SAL model, many different properties of that domain can be analyzed using the SAL model checker. However, at this point in the SAVH program, a target domain has not been developed, so our efforts have centered on the development of basic technology. In particular, we have focused on the mechanisms for expressing and translating constraints, which are central to the NDDL language. To enable the assessment of the power and usefulness of our translators, one verification property is generated. This property states that no plan exists that satisfies all of the constraints. The model checker is then deployed on this property. If a plan does exist, then the model checker will produce a counterexample that is a viable plan that satisfies the constraints. We do not envision that this approach will be nearly as efficient or powerful as EUROPA 2. We do believe that this translation technology could be useful for establishing that key properties of the domain knowledge are true. We think that this approach may one day provide a powerful means for debugging the specifications of domain knowledge that are used in safety-critical planning systems.

2 Simple Example

We will begin our look at the technique for translating APPL to SAL with a very simple two timeline example[2]:

[1] PDDL is the planning language of the Extensible Universal Remote Operations Architecture 2 (EUROPA 2). EUROPA 2 is a component-based software library for constructing highly-tailored, domain-specific planners.

[2] We assume a prior knowledge of SAL syntax throughout this document.

```
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 consists of three modules:

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

The structure of the module generated for timeline A is:

```
A_m : MODULE =
BEGIN
  INPUT

  GLOBAL

  LOCAL

  INITIALIZATION

  TRANSITION
[
    A0_to_A1:     %% A0 -> A1

  []
    A1_to_A2:     %% A1 -> A2

  []
    A2_to_A_null:     %% A2 -> A_null
```

```
  ]
END; %% A_m
```

Each of these sections is used in the translation:

- `INPUT` is used to select values for parameters of actions.
- `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

```
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 is then used to search through all possible sequences of actions on the timelines to find sequences which satisfy all of the constraints specified in the APPL model. These constraints fall into two broad categories:

- Timing constraints that impact durations and start/stop times of actions.
- Allen operations [1] that specify relationships between time intervals of actions.

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 = 150;                %% Always generated
TM_rng: TYPE = [0 .. MAX_TM];     %% Always generated
```

The progression of time is controlled by the `clock` module:

```
Clock: MODULE =
BEGIN
INPUT
  tm_intv: [1 .. 10]
OUTPUT
  time: TM_rng
INITIALIZATION
  time = 0;
TRANSITION
```

```
[
  advance_clock:
    time + tm_intv <= MAX_TM  --> time' = time + tm_intv;
]
END;
```

This module advances the clock by a value in the range of `1 ..10`. The clock is stored in the variable `time`. Note that since the modules are asynchronously composed, the clock module can effectively advance time at any point. Both the `A_m` and the `B_m` modules include the variable `time` in their `INPUT` sections.

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

```
GLOBAL
  B_state: B_actions,
  A_state: A_actions
```

The durations of the actions are controlled by the use of a `start` variable declared as follows

```
LOCAL
  start: TM_rng
INITIALIZATION
  start = 0;
```

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

```
TRANSITIONS

    A0 -> A1 -> A2
```

is translated into the SAL code:

```
TRANSITION
[
  A0_to_A1:     %% A0 -> A1
  A_state = A0
  AND time >= start + 2
  -->
    A_state' = A1;
    start' = time;
[]
  A1_to_A2:     %% A1 -> A2
  A_state = A1
  -->
    A_state' = A2;
    start' = time;
[]
  A2_to_A_null:     %% A2 -> A_null
  A_state = A2
  -->
    A_state' = A_null;
]
```

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[3] 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 >= 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` and that the `start` variable is re-initialized.

The APPL `GOALS` statement

[3]We have adopted a constructive semantics of allowed transitions: only the transitions that are explicitly specified are allowed. The EUROPA 2 system takes the opposite approach. All possible transitions are allowed unless specifically ruled out by a constraint. In future versions of the APPL to SAL translator, we may add an option that allows one to choose between constructive or destructive semantics.

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

lists the 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 APPL 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. If a specific ordering of goal achievement is desired, we can add a constraint using Allen operations in the `CONSTRAINTS` section to accomplish this. For example, if goal `A2` must be reached before `B1`, then the following constraint can be added:

```
    A.A2 WITH before B.B1
```

Allen operations will be discussed in greater detail in Section 3.

The complete generated SAL model (ex1.sal) is:

```
%% File ex1.sal
%% Generated from ex1.appl
%% On Fri Oct 20 14:14:39 EDT 2006
%% By APPL-2.b (08/14/06)

ex1: CONTEXT =
BEGIN

  INFNTY: int =  1000;              %% Always generated
  MAX_TM: int = 150;                %% Always generated
  TM_rng: TYPE = [0 .. MAX_TM];     %% Always generated

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

  B_actions: TYPE = {B0, B1};

  A_m : MODULE =
  BEGIN
    INPUT
      time: TM_rng
    GLOBAL
```

```
    B_state: B_actions,
    A_state: A_actions
  LOCAL
    start: TM_rng
  INITIALIZATION
    start = 0;
    A_state = A0;
  TRANSITION
  [
    A0_to_A1:     %% A0 -> A1
    A_state = A0
    AND time >= start + 2
    -->
      A_state' = A1;
      start' = time;
  []
    A1_to_A2:     %% A1 -> A2
    A_state = A1
    -->
      A_state' = A2;
      start' = time;
  []
    A2_to_A_null:     %% A2 -> A_null
    A_state = A2
    -->
      A_state' = A_null;


  ]
END; %% A_m

B_m : MODULE =
BEGIN
  INPUT
    time: TM_rng
  GLOBAL
    A_state: A_actions,
    B_state: B_actions
  LOCAL
    start: TM_rng
  INITIALIZATION
    start = 0;
```

```
    B_state = B0;
  TRANSITION
  [
    B0_to_B1:     %% B0 -> B1
    B_state = B0
    -->
      B_state' = B1;
      start' = time;
  []
    B1_to_B_null:     %% B1 -> B_null
    B_state = B1
    AND time >= start + 1
    AND time <= start + 10
    -->
      B_state' = B_null;


  ]
END; %% B_m


Clock: MODULE =
BEGIN
INPUT
  tm_intv: [1 .. 10]
OUTPUT
  time: TM_rng
INITIALIZATION
  time = 0;
TRANSITION
[
  advance_clock:
    time + tm_intv <= MAX_TM  --> time' = time + tm_intv;
]
END;

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

sched_sys: THEOREM
  System |- AG(NOT(A_state = A_null
```

```
                    AND B_state = B_null
                    ));

END %% ex1
```

3 Constraints Expressed Through Allen Operations

The allowed transitions on a timeline can be further restricted through use of Allen operations. The following Allen operations are allowed in the APPL language:

- `contains`
- `contained_by`
- `meets`
- `met_by`
- `starts`
- `equals`
- `ends`
- `before`
- `after`
- `overlaps`

In the generated SAL model, these constraints result in additional guards on the transitions and additional variable updates.

3.1 Approach

In this section, we will explain the translation approach without consideration of action parameters. In a later section, we will describe how the action parameters are handled. In the first pass of the translator, all of the Allen operations are collected in a data structure which stores five key fields:

- `tml_A`: Timeline of the first action.
- `act_A`: First action.
- `allop`: The Allen operation.
- `tml_B`: Timeline of the second action.

- `act_B`: Second action.

For example, the Allen operation

```
Nav.X2 starts Instr.Y3
```

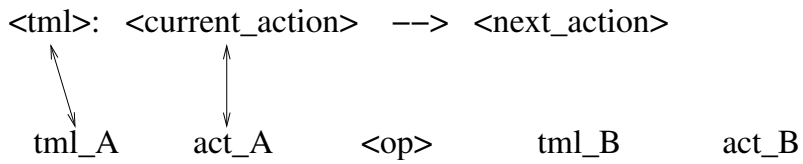
results in `tml_A = Nav, act_A = X2, allop = starts`, `tml_B = Instr`, `act_B = Y3`. In the second pass of the translator, all of these constraints are matched against the transition that is being generated. We will use the following notation to represent the transition under construction:

```
<tml>: <current_action> -> <next_action>
```

The current timeline is `<tml>` and we are processing the transition from `<current_action>` to `<next_action>`. We will introduce some notation to indicate when there is a "match." A constraint will be denoted as follows:

```
tml_A  act_A   <op>   tml_B   act_B
```

where `<op>` is one of the Allen ops (e.g. `meets`, `contains`, etc). We will express the matching requirement by drawing lines between items that must be equal:



This diagram expresses the requirement that `<tml>` = `tml_A` and `<current_action>` = `act_A`.

The basic approach is as follows. For each transition being generated, the translator searches through all of the constraints looking for matches against patterns associated with each type of Allen operation. The patterns are used to specify:

- guards on the transition which occur on the left or **pre** side of the `-->`
- variable updates which occur on the right or **post** side of the `-->`

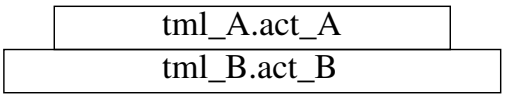
These "patterns" are described in the next subsections. We will not include the routine update of the `time` variables in the following sections, to simplify the presentation.

3.2 The `contained_by` Allen Operator: `A contained_by B`

There are two matching patterns required for the `contained_by` operation. Suppose we have a constraint

```
tml_A.act_A contained_by tml_B.act_B
```

which can be illustrated as follows:



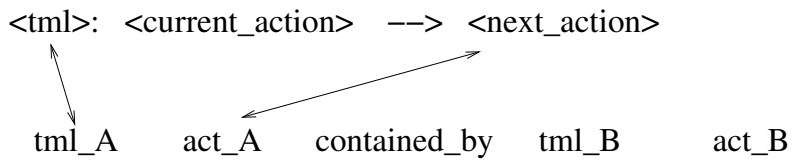
While generating any transition of the form `<current_action> -> act_A` for timeline `tml_A`, a check needs to be made to insure that `act_B` is already executing on the `tml_B` timeline. Therefore, the following guard is added to the `<current_action> -> act_A` transition:

```
tml_B_state = act_B
```

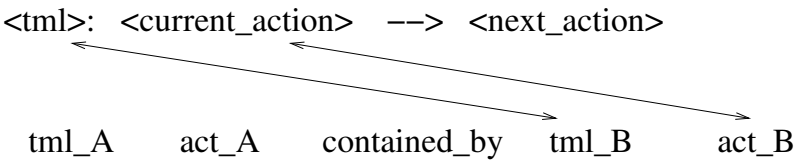
The translator searches through the list of all of the constraints while processing a `<current_action> -> <next_action>` transition. If it finds a constraint with `<op>` = `contained_by`, `tml_A` = `<tml>` and `act_A` = `<next_action>`, then this guard is added and the net result is:

```
    tml_A_state = <current_action>;
    tml_B_state = act_B
-->
    tml_A_state' = act_A;              %% = <next_action>
```

This particular match is illustrated below:



It is also necessary to keep `act_B` from terminating while `act_A` is still executing. This is accomplished using another pattern:



In other words for transitions of the form `tml_B: act_B -> <next_action>` (i.e. where `<tml>` = `tml_B` and `<current_action>` = `act_B`), we must add the following guard:

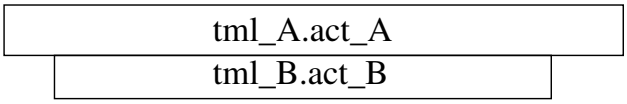
```
AND tml_A_state /= act_A
```

The net result is

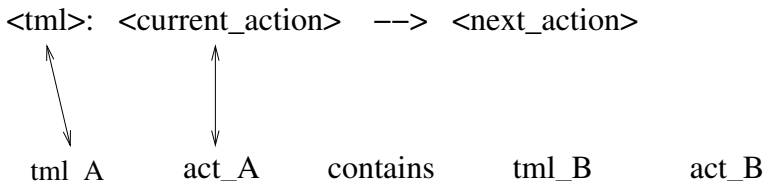
```
    tml_B_state = <current_action>;
    AND tml_A_state /= act_A
-->
    tml_B_state = act_B                %% = <next_action>;
```

3.3 The `contains` Allen Operation: `A contains B`

There are three matching patterns required for the `contains` operation:



The first pattern is



Whenever there is a match, shown in this illustration, the following additional guard is added to the `<current_action> -> <next_action>` transition:

```
    AND tml_B_state /= act_B
```

The net result is:

```
    tml_A_state = act_A;                    %% = <current_action>;
    AND tml_B_state /= act_B
-->
    tml_A_state' = <next_action>;
```

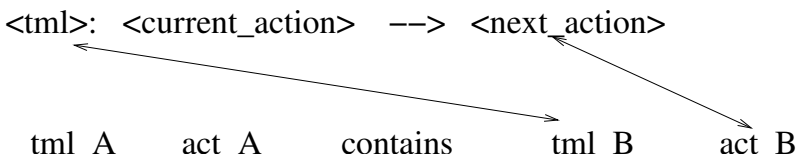
The reason for this guard is simple. Because `tml_A.act_A` contains `tml_B.act_B`, `act_A` cannot be allowed to terminate as long as `act_B` is still active. But what if `act_B` never executed at all? We do not want the action `act_A` to terminate until after an action `act_B` has started and completed. So, we need to keep track of this event using a state variable. We add a variable `act_B_while_act_A` to record this information. We then add a test on this variable to the transition guard:

```
    tml_A_state = act_A;                    %% = <current_action>;
    AND tml_B_state /= act_B
    AND act_B_while_act_A
-->
    tml_A_state' = <next_action>;
    act_B_while_act_A = false;
```

The variable `act_B_while_act_A` is reset to `false` in preparation for a future execution of the action. The initial value of this variable is set as:

```
act_B_while_act_A = (tml_A_state = act_A) AND
                    (tml_B_state = act_B);
```

The second pattern is



When a transition into state `act_B` occurs, we need to update the value of the variable `act_B_while_act_A`. If action `act_A` is executing on the other timeline, then the variable `act_B_while_act_A` is set to true as follows:

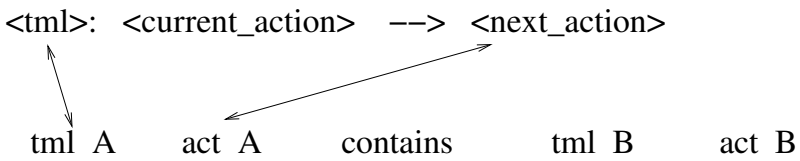
```
IF tml_A_state = act_A THEN
   act_B_while_act_A' = true;
ENDIF;
```

The net result is

```
    tml_B_state = <current_action>;
-->
    tml_B_state' = act_B                    %% = <next_action>;
    act_B_while_act_A' = IF tml_A_state = act_A THEN
                            true
                         ELSE
                            act_B_while_act_A
                         ENDIF;
```

where the update syntax is rewritten in a form suitable for SAL.

The third pattern is



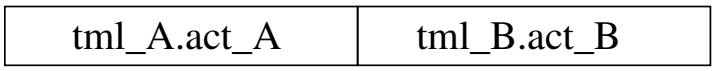
When a transition into state `act_A` occurs, we need to update the value of the variable `B_while_A` if the time has not advanced. If action `act_B` has just been initiated on timeline `tml_B`, then the variable `B_while_A` is set to true as follows:

```
    tml_A_state = <current_action>;
-->
    tml_A_state' = act_A                %% = <next_action>;
    act_B_while_act_A' = IF tml_B_state = act_B AND B_start = time THEN
                            true
                         ELSE
                            act_B_while_act_A
                         ENDIF;
```

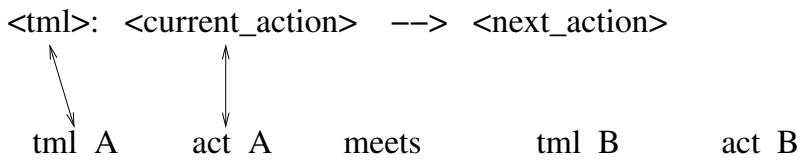
It should be noted that this allows equal start times for actions `A` and `B`. If a strict "contains" is desired, one should leave out the code generated from this last pattern.

3.4 The `meets` Allen Operator: `A meets B`

There are two matching patterns required for the `meets` operation:



The first one occurs when `tml_A.act_A` terminates:



Whenever there is a match shown in this figure, the following variable update is added to the `<current_action> -> <next_action>` transition:

```
tml_B_state' = act_B;
```

The net result is

```
    tml_A_state = act_A;                    %% = <current_action>;
-->
    tml_A_state' = <next_action>;
    tml_B_state' = act_B;
```

Thus, whenever `act_A` terminates, `act_B` is initiated on timeline `tml_B`. But, what if the currently executing action on timeline `tml_B` is not an allowed predecessor of `act_B`? To handle that case, we need an additional guard on the `<current_action> -> <next_action>` transition resulting in:

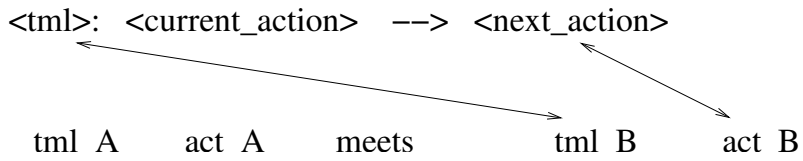
```
    tml_A_state = act_A;                    %% = <current_action>;
    ( tml_B_state = act_PB1 OR
      tml_B_state = act_PB2 OR
                ⋮
      tml_B_state = act_PBn )
-->
    tml_A_state' = <next_action>;
    tml_B_state' = act_B;
```

where `act_PB1, act_PB2, ... act_PBn` are all the allowed predecessor actions of `act_B`. It is also necessary to check that the duration constraints on all of these predecessor actions are met. These necessitate the use of global start variables: `A_start` and `B_start`. If there were two predecessor actions to `act_B`, say `B0` and `B1`, we would end up with a guard similar to the following:

```
(  ( B_state = B0
     AND time >= B_start + 0
     AND time <= B_start + 10
   ) OR ( B_state = B1
          AND time >= B_start + 5
        )
)
```

It is also necessary to make sure that any constraints on the initiated action on the timeline are satisfied. In the above example it is necessary to examine the `B0 -> act_B` and `B1 -> act_B` against all of the Allen operators in the database. This is accomplished by a recursive call to the procedure we are describing in this section.

The semantics of the constraint can be strengthened by matching a second pattern:



When processing the transitions into the `act_B` action on the `tml_B` timeline, we update the `tml_A` timeline as well:

```
tml_B_state = <current_action>
AND tml_A_state = act_A;
-->
tml_B_state' = act_B                    %% = <next_action>
tml_A_state' IN act_NA_i, ... , act_NA_n;
```

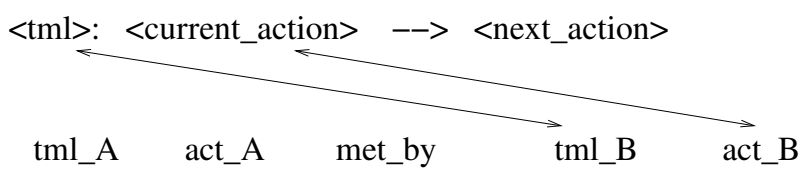
where `act_NA_i`, ..., `act_NA_n` are the successor actions of `act_A`. And, of course, we must check that the durations and constraints are satisfied for this transition as before.

3.5 The `met_by` Allen Operator: `A met_by B`

There are two matching patterns needed for the `met_by` operation:

tml_A.act_B	tml_B.act_A

Whenever the translator finds the following match:



the following variable update is added to the <current_action> -> <next_action> transition:

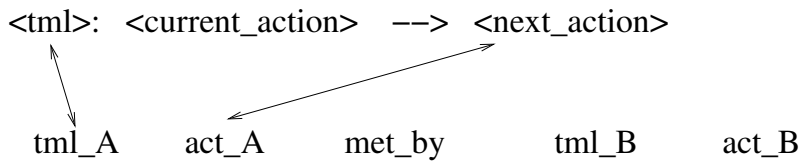
```
tml_A_state' = act_A;
```

In other words, when act_B is terminated on timeline tml_B, action act_A is started on the other timeline. This of course, necessitates a check that a predecessor action of act_A is currently executing. The net result is:

```
tml_B_state = act_B
( tml_A_state = act_PA1 OR
  tml_A_state = act_PA2 OR
          ⋮
  tml_A_state = act_PAn
)
-->
tml_B_state' = <next_action>
tml_A_state' = act_A;
```

where act_PA1, act_PA2, . . . , act_PAn are the predecessor actions of act_A.

Also it is necessary to match transitions on the other timeline as well:



In other words, we can allow entrance into state act_A, if action act_B is currently executing. Therefore, the following guard must be added:

```
tml_B_state = act_B
```

Since act_A is met by act_B, it is necessary that act_B be terminated with an update to tml_B_state:

```
tml_B_state' IN {act_NB1, .., act_NBn};
```

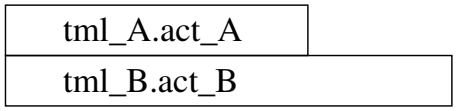
where NB_i, .., NB_n are the successor actions of act_B. The net result is

```
    tml_A_state = act_A;                  %% = <current_action>;
    tml_B_state = act_B
-->
    tml_A_state' = <next_action> ;
    tml_B_state' IN {act_NB1, act_NB2, ...act_NBn} ;
```

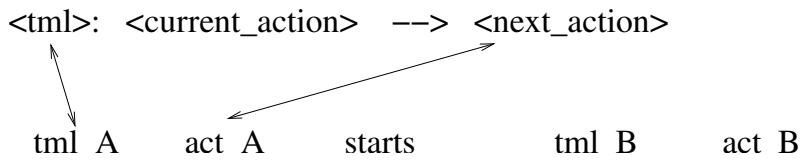
So whenever act_A is terminated, act_B is initiated. And of course we must check that the durations and constraints are satisfied for this transition as in the meets operator.

3.6 The `starts` Allen Operator: `A starts B`

There is just one matching pattern needed for the `starts` operation:



Whenever there is a match



the following variable update is added to the `<current_action> -> <next_action>` transition:

```
tml_B_state' = act_B;
```

The net result is

```
        tml_A_state = <current_action>;
        ( tml_B_state = act_PB1 OR
          tml_B_state = act_PB2 OR
                 ⋮
          tml_B_state = act_PBn
        )
    -->
        tml_A_state' = act_A;                %% = <next_action>
        tml_B_state' = act_B;
```

where `act_PB1, act_PB2, ... act_PBn` are all of the allowed predecessor actions of `act_B`. So, whenever `act_A` is initiated, `act_B` is also initiated, and this transition is guarded with the requirement that a predecessor action of `act_B` is executing.

There is a semantics issue that arises here: whether or not `act_B` can be scheduled without `act_A` being started. For example, does the following timeline trace

```
  --------------
A:  A1   |  A3 |
  --------------
B:  B1   |  B2 |
  --------------
```

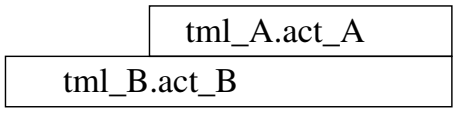
satisfy the constraints

```
A1 -> (A2 | A3 )

B1 -> B2

A.A2 starts B.B2
```

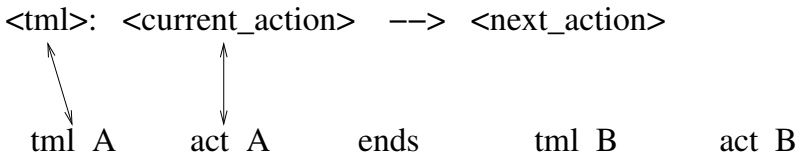
In this trace, `A2` does not execute. Since the semantics of `A op B` is `FORALL A EXISTS B`, the above trace represents a vacuous solution and hence does meet the constraint.

3.7 The `ends` Allen Operator: `A ends B`

There is just one matching pattern needed for the `ends` operation:



Whenever there is a match



action `act_B` has to be terminated along with action `act_A`. The net result is:

```
        tml_A_state = act_A;                    %% = <current_action>
        tml_B_state = act_B
    -->
        tml_A_state' = <next_action>;
        tml_B_state' IN act_NB1, act_NB2, ..., act_NBn;
```

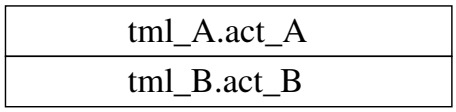
where `act_NB1`, `act_NB2`, ..., `act_NBn` are all of the allowed successor actions of `act_B`. So, whenever `act_A` is terminated `act_B` is also terminated and a successor action is initiated. If there is no successor state for `act_B`, then the guard is falsified with the addition of

```
        AND FALSE
```

If the guard is falsified, then `act_A` is never allowed to terminate.

3.8 The `equals` Allen Operator: `A equals B`

The `equals` operator



is implemented as the combination of `starts` and `ends`. In other words, the Allen operation `A equals B` is internally translated into `A starts B` and `A ends B`.

3.9 The `before` Allen Operator: `A before B`

The remaining three operators – `before`, `after`, and `overlaps` – require that additional variables be introduced in the SAL model in order to enforce the desired behavior. For example, to ensure that action `act_1` occurs before `act_2`, a variable must be used to record the fact that `act_1` has occurred some time in the past. One extra variable is needed for each `before` constraint, because the variable that enforces `act_1 before act_2` does not help in establishing `act_3 before act_4`.

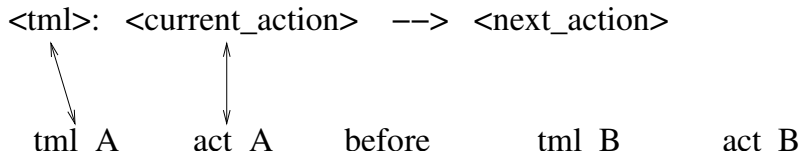
The model generator has a mechanism for creating unique variable names, so that no name collision is possible when dealing with multiple constraints. The operator names and the actions' names and parameters that are involved are part of this identifier.

There are two matching patterns required for the `before` operation:

For example, the `before` constraint

```
tml_A.act_A before tml_B.act_B
```

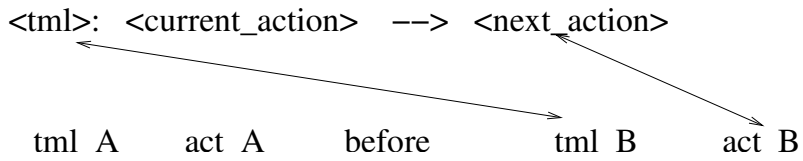
a Boolean variable `tml_A_act_A_before_tml_B_act_B` is declared and initialized as `FALSE`. Then, whenever there is a match:



the fact that `tml_A.act_A` has occurred is recorded:

```
    tml_A_state = act_A;                    %% = <current_action>
-->
    tml_A_state' = <next_action>;
    tml_A_act_A_before_tml_B_act_B' = true
```

Also, action `act_B` cannot be executed unless action `act_A` has executed in the past. So whenever there is a match:



the following code is generated:

```
    tml_B_state = <current_action>
    AND tml_A_act_A_before_tml_B_act_B
-->
    tml_B_state' = act_B;
```

Another semantics issue is raised here. The mechanism for recording the occurrence of `act_A` can be modified so that it is matched repeatedly (by resetting the history variable) or just once (without resets). These two interpretations correspond to two distinct views of the Allen operator semantics. There is no clear indication as to which approach is to be preferred. The current implementation accepts a timeline where action `act_B` never occurs. This is a weaker constraint than requiring that for each `act_A`, there should necessarily be a matching occurrence of `act_B`. The current implementation is also strict. In other words, if `A meets B`, then `A before B` is not true.

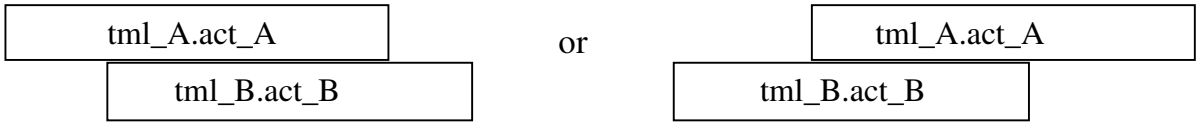
3.10 The `after` Allen Operator: `A after B`

The `after` operator

is the dual of the `before` operator, i.e. the code generated for the constraint `act_A before act_B` is identical to that for `act_B after act_A`. Hence, the code generation is very similar to the above: an additional history variable is needed to record the occurrence of `act_A`, when leaving the state `act_A`, this fact is recorded in the history variable. Any transition into `act_B` is then guarded by the condition that the history variable is `true`.

3.11 The `overlaps` Allen Operator: `A overlaps B`

The constraint `act_A overlaps act_B`

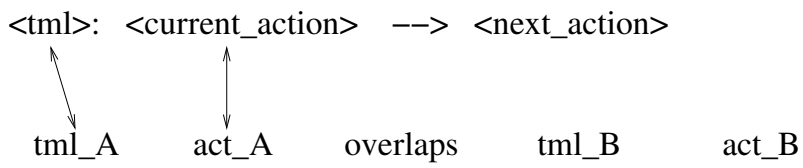


can be satisfied by two scenarios:

- `act_A` occurs first, then the system needs to traverse the following sequence of states and no other: `act_B` starts, `act_A` ends, and afterwards `act_B` ends.
- `act_B` occurs first, then the only allowed sequence is: `act_A` starts, `act_B` ends, and then `act_A` ends.

Monitoring the correct succession of states can be enforced by using one additional variable that records either one of the above initial events and then the subsequent intermediate states that are traversed. Alternatively, we can use two Boolean variables to record which one of the initial conditions has occurred: has `act_A` happened first or `act_B`?

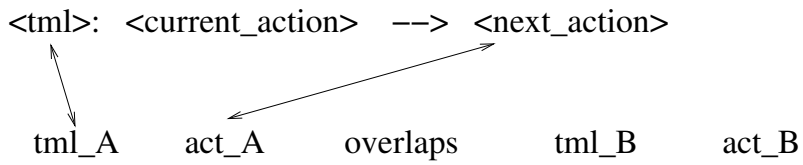
There are four separate match patterns that generate code for an `overlaps` constraint. More precisely we can be either entering or leaving either `act_A` or `act_B`. The first pattern is



The code generated for the first pattern is:

```
    tml_A_state = act_A                    %% = <current_action>
    AND act_A_first_overlaps_act_B
-->
    tml_A_state' = <next_action>;
```

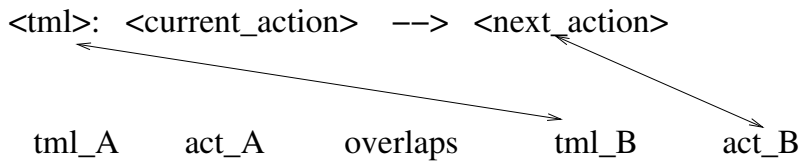
The second pattern is:



The code generated for the second pattern is:

```
    tml_A_state = <current_action>
-->
    tml_A_state' = act_A                     %% = <next_action>
    act_A_first_overlaps_act_B' = NOT act_B_first_overlaps_act_A
```

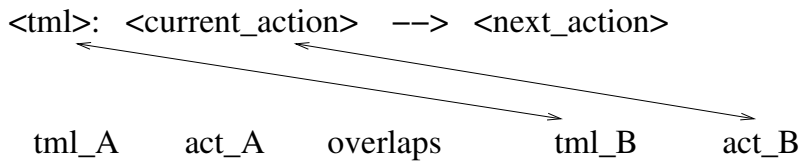
The third pattern is



The code generated for the third pattern is:

```
    tml_B_state = <current_action>
-->
    tml_B_state' = act_B;                    %% = <next_action>
    act_B_first_overlaps_act_A' = NOT act_A_first_overlaps_act_B
```

The fourth pattern is



The code generated for the fourth pattern is:

```
        tml_B_state = act_B
        AND act_B_first_overlaps_act_A
    -->
        tml_B_state' = <next_action>;
```

4 Transition Parameters

The processing of action parameters results in additional guards and variable updates. Consider the following example transition

```
A1(xx,yy,_,ww) -> A2(_,yy,ww)
```

where the matching of parameter names indicates a constraint on the transition. For each matching argument, it is necessary to generate a variable to hold the value of that matched parameter. The translator creates a variable name by concatenating the parameter identifier to the action name. For example, if the `A1` and `A2` actions were defined as follows:

```
A1(p1: type1, p2: type2, p3: type3, p4: type4)

A2(p1: type1, p2: type2, px: type4)
```

the translator will generate the following `GLOBAL` variables:

```
A1_p2: type2;
A1_p4: type4;
A2_p2: type2;
A2_px: type4;
```

Note that variables are only created where needed, namely where there is a match. The transition statement equates these parameters as follows:

```
A1_p2 = A2_p2
A1_p4 = A2_px
```

These parameter matchings can be generated on the **pre** side, such as

```
A1_p2 = A2_p2' AND A1_p4 = A2_px'
```

or the **post** side

```
A2_p2' = A1_p2;
A2_px' = A1_p4;
```

The primes indicate that these are the values of the variables after the update. The advantage of the **post** form is that it directly selects the new values for the parameters after the transition. However, in the presence of multiple constraints it is possible to have multiple updates on the same parameter:

```
A1_p2' = A2_p2;
A1_p2' = B3_p2;
```

This construction is not allowed in SAL, while the **pre** version is allowed:

```
A1_p2' = A2_p2 AND A1_p2' = B3_p2
```

Although it is necessary to actually update the variables, the **pre** form is deployed as much as possible to avoid illegal updates.

5 Constraint Parameters

The impact of parameters on the Allen operation constraints is very complicated. In fact, many subtle semantics issues arise here. These will not be dealt with in detail in this paper. Instead, we will present a few examples of how parameter processing is done, without attempting to be complete.

For convenience, we will restrict our attention to two timelines A and B with actions `A1`, `A2`, ... and `B1`, `B2`, ... For simplicity, we will not address the processing of the action durations in the following subsections. Also for simplicity we will assume that all actions have the following parameter identifiers: `x: animals`, `y: animals`, `z: animals`. For example:

```
A1(x: animals, y: animals, z: animals)
A2(x: animals, y: animals)
B4(x: animals, y: animals)
```

5.1 Example 1 – Parameter Matching

Suppose we have the following transitions with matching parameters:

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

B1(aa,_) -> B2(_,aa)

A.A1(aaa) WITH meets B.B2(fish,aaa)
```

When processing the `A1(xx) -> A2(_,xx)` transition, we must first handle the matching parameter as follows:

```
A_state = A1
-->
  A_state' = A2;
  A2_y' = A1_x;
```

Next, we need to make sure that the `meets` constraint is satisfied. Therefore, the `B2` action must be initiated as well.

```
A_state = A1
AND B_state = B1
-->
  A_state' = A2;
  B_state' = B2;
  A2_y' = A1_x;
```

Notice the presence of the guard `AND B_state = B1`, which insures that the appropriate predecessor state of `B2` is executing. Next, we need to make sure that the parameter matchings of the `meets` constraint are satisfied:

```
A_state = A1
AND B_state = B1
    AND B2_x' = fish
    AND B2_y' = A1_x
-->
  A_state' = A2;
  B_state' = B2;
  A2_y' = A1_x;
```

These guards are indented for easy recognition. Finally, we must make sure that the `B1(aa,_) -> B2(_,aa)` transition rules are observed:

```
A_state = A1
AND B_state = B1
    AND B2_x' = fish
    AND B2_y' = A1_x
-->
  A_state' = A2;
  B_state' = B2;
  B2_y' = B1_x;
  A2_y' = A1_x;
```

The `A.A1(aaa) WITH meets B.B2(fish,aaa)` constraint must also be considered when processing the transition from `B1 -> B2`. Clearly, the system should be able to make this transition when the action on the `A` timeline is `A1`. But what if the current action on the `A` timeline is not `A1`? Although we can envision solutions where `B1` proceeds to `B2` and the `A` timeline remains in state `A1` awaiting a future `B2` to meet, why not take advantage of this situation and transition `A1` to `A2` along with the `B1` to `B2` transition (and thus satisfy this Allen op constraint immediately)?

```
B1_to_B2:     %% B1 -> B2
```

```
B_state = B1
AND (A_state = A1 => ( TRUE
     AND A1_x' = A2_y'
     AND B2_x' = fish
     AND B2_y' = A1_x ))
-->
  B_state' = B2;
  IF A_state = A1 THEN
    A2_y' = A1_x;
    A_state' = A2;
  ENDIF
  B2_y' = B1_x;
  B2_x' IN {x: animals | true};
```

Notice that the guard contains the following conditional `(A_state = A1 => ...`. If this conditional is true, then the constraints associated with the transition from `A1` to `A2` are enforced by some further equalities after the conditional and with some variable updates inside an `IF-THEN-ELSE`. Note that if the Allen operation contains some constant parameters in the first action (e.g. `A.A1(fish) WITH meets B.B2` , this is handled by adding a term to the conditional as follows `(A_state = A1 AND A1_x = fish) => ...` .

5.2 Example 2 – Constant Parameters

```
A1(xx,_,_) -> A2(_,xx) -> A3

B0 -> B1 -> B2(dog,_,_)

A.A2 starts B.B2(_,cat,donkey)
```

when processing `A1 -> A2`, the following will be generated:

```
A_state = A1
AND B_state = B1
AND B2_y' = cat AND B2_z' = donkey
-->
  A_state' = A2;

  B_state' = B2;
    B2_x' = dog;
    B2_y' IN {x: animals | true}
    B2_z' IN {x: animals | true}
  A2_y' = A1_x;
  A2_x' IN {x: animals | true}
```

The guard `AND B2_y' = cat AND B2_z' = donkey` insures that the constant parameters of `B.B2` in the Allen operation are properly set. The variable update `B2_x' = dog` insures that the B timeline transition constraints are met. The last two variable updates insure that the `A` timeline constraints are met. Note that the `A2_x' IN {x: animals | true}` update occurs because of the `"_"` in the first parameter position of the `A2` action. Also note that although the `B2_y` and `B2_z` parameters could be updated as follows:

```
B2_y' = cat;
B2_z' = donkey;
```

If the following additional constraint were present

```
B1(mm) -> B2(dog,mm,_)
```

we would obtain two different updates of `B2_y`:

```
B2_y' = cat;
B2_y' = B1_x;
```

which is not allowed in SAL.

5.3 Example 3 – Constant Parameters

```
A1(mm) -> A2(_,mm) -> A3

B0 -> B1 -> B2 -> B3

A.A2(dog,ww) WITH contains B.B2(ww,_)
```

When processing `A2 -> A3`, the following is generated:

```
A_state = A2
AND (A2_x = dog
       => (NOT (B_state = B2
                AND B2_x = A2_y))
           AND B2_while_A2 )
-->
  A_state' = A3;
```

The first thing to note is that the guard has an implication (`=>`). Thus, the requirement that `A2` contains `B2` is only enforced when the first parameter of `A2` is `dog`. Therefore if an `A2` is executing with a first parameter that is not equal to `dog`, there is no constraint that it must contain `B2`. If the first parameter of `A2` is equal to `dog`, then the transition is not allowed to occur as long as `B_state = B2 AND B2_x = A2_y` is true.

If the `A` timeline transitions were modified to:

```
A1(mm) -> A2(dog,mm) -> A3
```

then the following would be generated

```
A_state = A2
AND A2_x = dog
AND (A2_x = dog
       => (NOT (B_state = B2
                AND B2_x = A2_y)
           )
    )
AND B2_while_A2
-->
  A_state' = A3;
```

This is logically equivalent to

```
A_state = A2 AND A2_x = dog
AND (NOT (B_state = B2 AND B2_x = A2_y)
AND B2_while_A2
-->
  A_state' = A3;
```

but the translator does not perform this optimization.

The `B1 -> B2` transition is

```
B_state = B1
-->
  B_state' = B2;
  IF A_state = A2 AND A2_x = dog AND B2_x' = A2_y THEN
     B2_while_A2' = true;
  ENDIF;
  B2_x' IN {x: animals | true};
```

Another issue that arises is whether the `contains` operation should be strict containment or whether equal starting or end times should be allowed. The mechanisms described previously for contains, have allowed equal start or end times.

5.4 Example 4 – Constant Parameters

```
A0(xx) -> A1(xx,_) -> A2(_,xx) -> A1(_,xx)
A2(dog,_) -> A3

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

A.A2(dog,qq) meets B.B2(qq)
```

When generating a transition from A2 to A1 we get:

```
A_state = A2
AND (A2_x = dog => (B_state = B1
                    AND B2_x' = A2_y))
-->
  A_state' = A1;
  A1_y' = A2_y;
  A1_x' IN {x: animals | true}
  IF A2_x = dog THEN
      B_state' = B2;
      B_start' = time;
  ENDIF;
```

This IF THEN ENDIF construct is not supported in SAL. Therefore, it is translated into the following in a third pass of the translator.

```
  B_state' = IF A2_x = dog THEN B2 ELSE B_state ENDIF;
```

The IF THEN ENDIF version is available in an auxiliary *.sl1 output file. This file is often easier on human eyes than the final *.sal file.

When generating a transition from B1 to B2 we get:

```
B_state = B1
AND ((A_state = A2 AND A2_x = dog) =>
       (     ( A_state' = A1 => A2_y' = A1_y'  )
        AND ( A_state' = A3 => A2_x = dog     )
        AND B2_x' = A2_y
    ))
-->
  B_state' = B2;
  IF A_state = A2 AND A2_x = dog THEN
     IF A_state' = A1 THEN
       A1_y' = A2_y;
     ENDIF
     A_state' IN {A1, A3};
  ENDIF
  B2_x' IN {x: animals | true};
```

6 Processing IF THEN ELSE expressions

The specification of an action allows a WITH clause with an "if then else" expression. In each branch, there can be different constraints. But let's start with a simple case:

```
A2(x,y: animals):
  WITH
    if x = gorilla then
      starts B.B2(fish,x,y)
    else
      starts B.B2(cat,horse,fish)
    endif
```

Now suppose the transitions are constrained as follows:

```
A1(xx,_,_) -> A2(xx,_) -> A1  -> A3

B0 -> B1 -> B2
```

Then the following is generated for the `A1` to `A2` transition

```
  A_state = A1
  AND (A2_x = gorilla
       => (B_state = B1 AND B2_x' = fish
           AND B2_y' = A2_x'
           AND B2_z' = A2_y')
          )
  AND (NOT A2_x = gorilla
       => (B_state = B1
           AND B2_x' = cat AND B2_y' = horse AND B2_z' = fish)
  -->
    A_state' = A2;
    start' = time;
    B_state' = B2
    A2_x' = A1_x;
    A2_y' IN {x: animals | true}
```

Nested "if then else" expressions are a bit more tricky to handle. Consider the following example:

```
A1(x,y: animals)
  WITH
    if x = cat then
        contains B.B1(_,dog)
    else
       if y = dog then
          contains B.B2(cat,_,fish)
       else
          meets B.B2(_,horse,cat)
       endif
```

```
    endif

A2(x,y: animals)
```

with the following transitions

```
A0 -> A1(dog,y) -> A2(_,y)

B0 -> B1(aa,_)  -> B2(dog,aa,aa)
```

The transition from `A1` to `A2` is generated (omitting timing) as follows:

```
  A_state = A1
  AND A1_x = dog
  AND (A1_x = cat
          => (NOT (B_state = B1 AND B1_y = dog ))
      )
  AND (A1_y = dog AND NOT A1_x = cat
           => (NOT (B_state = B2 AND B2_x = cat AND B2_z = fish ))
      )
  AND (NOT A1_y = dog AND NOT A1_x = cat
       => (B_state = B1 AND
             (A1_y = dog AND NOT A1_x = cat
               => (A_state = A1 AND B2_x' = cat AND B2_z' = fish)
             )
             AND B1_x = B2_y'
             AND B1_x = B2_z'
             AND B2_y' = horse AND B2_z' = cat
           )
      )
  -->
    A_state' = A2;
    A_start' = time;
      IF NOT A1_y = dog AND NOT A1_x = cat THEN
        B_state' = B2;
        B2_x' = dog;
        B2_y' = B1_x;
        B2_z' = B1_x;
      ENDIF;
    A2_y' = A1_y;
```

7 Semantic Subtleties

In the development of the translator, several subtle issues associated with the exact meaning of the Allen operations were discovered. In this section, we will explore some of these.

7.1 Non-scheduled Actions and Vacuous Solutions

Suppose we have two timelines `A` and `B` with the following actions and allowed sequences:

```
A0 -> (A1 | A2)

A1 -> A2

B0 -> B1 -> B2

B0 -> B2
```

The two transitions from `B0` represent alternatives and are equivalent to `B0 -> (B1 | B2)` in APPL. Now consider the following Allen operation:

```
A1 WITH ends B1
```

The following time sequences clearly satisfy the `ends` constraint:

```
-------------------------------------------------
  A0  | A1              |  A2       |
-------------------------------------------------
  B0     | B1           |  B2               |
-------------------------------------------------
```

Now suppose that `A1` never executes. Is the Allen operation satisfied in this case? Based upon preliminary experimentation, it appears that the EUROPA 2 planner allows such "vacuous" solutions.

Now suppose that `B1` is not executing at the time `A1` is executing as in the following timeline sequence:

```
-------------------------------------------------
  A1                    |  A2       |
-------------------------------------------------
  B0   |   B2               |
-------------------------------------------------
```

Does this satisfy the constraint? Based upon preliminary experimentation, the EUROPA 2 planner requires that the `B1` action be explicitly scheduled and must end with `A1`. In fact, if there are multiple `A1`s, each must end its own `B1`.

Therefore, given an Allen operation `A op B`, we can express the meaning as follows:

```
FORALL A: EXISTS B:  A op B
```

7.2 Multiple Invocations with `After` Constraints

Consider the following Allen Operation

```
A1 before B2
```

Does the following timeline meet this condition?

```
--------------------------------
|   A1  | A2                   |
--------------------------------
|   B1    | B2   |  B3  |  B2  |
--------------------------------
```

or must there be an `A1` before every `B2`?

```
-----------------------------------------------
|   A1  | A2             | A1 |    A2        |
-----------------------------------------------
|   B1    | B2   |  B3          |  B2  |
-----------------------------------------------
```

Based upon our experimentation with EUROPA 2, the first timeline appears to be adequate. This is consistent with the FORALL-EXISTS semantics given above.

7.3 Terminal States

Suppose we have the following Allen operation:

```
A1 meets B2
```

If `A1` never transitions to `A2` (i.e. it is still active at HorizonEnd), is the following an acceptable solution?

```
-------------------------------------------------------|
  A1(horse)                                            |
-------------------------------------------------------|
  Idle | B1(horse,_)          |  B2(dog,horse)         |
-------------------------------------------------------|
```

EUROPA 2 accepts this as a valid plan. It should be noted that this is a violation of the FORALL-EXISTS semantics. Therefore, the user of the Allen operators needs to remember that the FORALL-EXISTS semantics does not apply at the beginning and end of the timelines.

7.4 Parameters

The presence of parameters significantly complicates the semantics of the Allen operators. There are three categories to consider:

- constant parameters, e.g. `A1(horse)`
- matching parameters, e.g. `A1(mmm) meets B4(mmm)`
- wild card parameters, e.g. `A1(_)`

and combinations of these.

7.4.1 Constant Parameters

Suppose we have the following Allen operation:

```
A1(cat) WITH ends B1
```

Does the following timeline satisfy this specification?

```
-------------------------------------------------------|
  A1(horse)                  | A2                      |
-------------------------------------------------------|
  Idle | B1(horse)           |  B2(dog,horse)          |
-------------------------------------------------------|
```

In other words, is this an allowable vacuous solution? In EUROPA 2, the answer is yes. The constant parameter can be thought of as a name extension, e.g. `A1_cat`. The semantics is thus:

```
FORALL A_cat: EXISTS B:  A_cat op B
```

Therefore, given two actions

```
A(p1,p2,...,pn: type)
B(q1,q2,...,qm: type)
```

and the following Allen operation

```
A(a1,a2,...,an) op B(b1,b2,...,bm)
```

where all of the arguments are constant parameters, the meaning is as follows:

```
FORALL A: p1 = a1 AND p2 = a2 AND ... AND pn = an
       -> EXISTS B, b1, b2, ..., bm:
             A[p1,...,pn](a1,a2,...,an) op B[p1,...,pn](b1,b2,...,bm)
```

7.4.2 Matching and Wild Card Parameters

Matching parameters do not restrict the domain of application of the Allen operation. Instead they constrain the allowed values of an acceptable plan. Therefore given two actions

```
A(p1,p2,...,pn: type)
B(q1,q2,...,qm: type)
```

and the following Allen operation

```
A(xx,_,yy_, ... ) op B(_,yy,_,_,xx, ...)
```

where the `xx` and `yy` are matching parameters. The meaning is as follows:

```
FORALL A: EXISTS B, xx, yy:
                  A(xx,_,yy_, ... ) op B(_,yy,_,_,xx, ...)
```

7.5 Loss of Allen Op Symmetry

When the definitions of the Allen Operations are presented graphically as in Figure 1, one would expect that there would be a symmetry between many of the Allen ops. For example, `A contains B` if and only if `B is contained_by A`. While this is true of any *single pair* of action intervals, the symmetry does not hold when used to describe a domain model. The reason is that there is an implicit quantification over many instances of these actions. For example, consider the following specification:

```
A1 meets B2
```

The following plan satisfies this specification:

```
-------------------------------------------------------------------
   A1                                        | A2
-------------------------------------------------------------------
   Idle | B1        |  B2    | B1            | B2
-------------------------------------------------------------------
```

But, this plan is not satisfied if the above Allen operation is replaced by its dual:

```
B2 met_by A1
```

This observation is consistent with the FORALL-EXISTS semantics discussed previously.

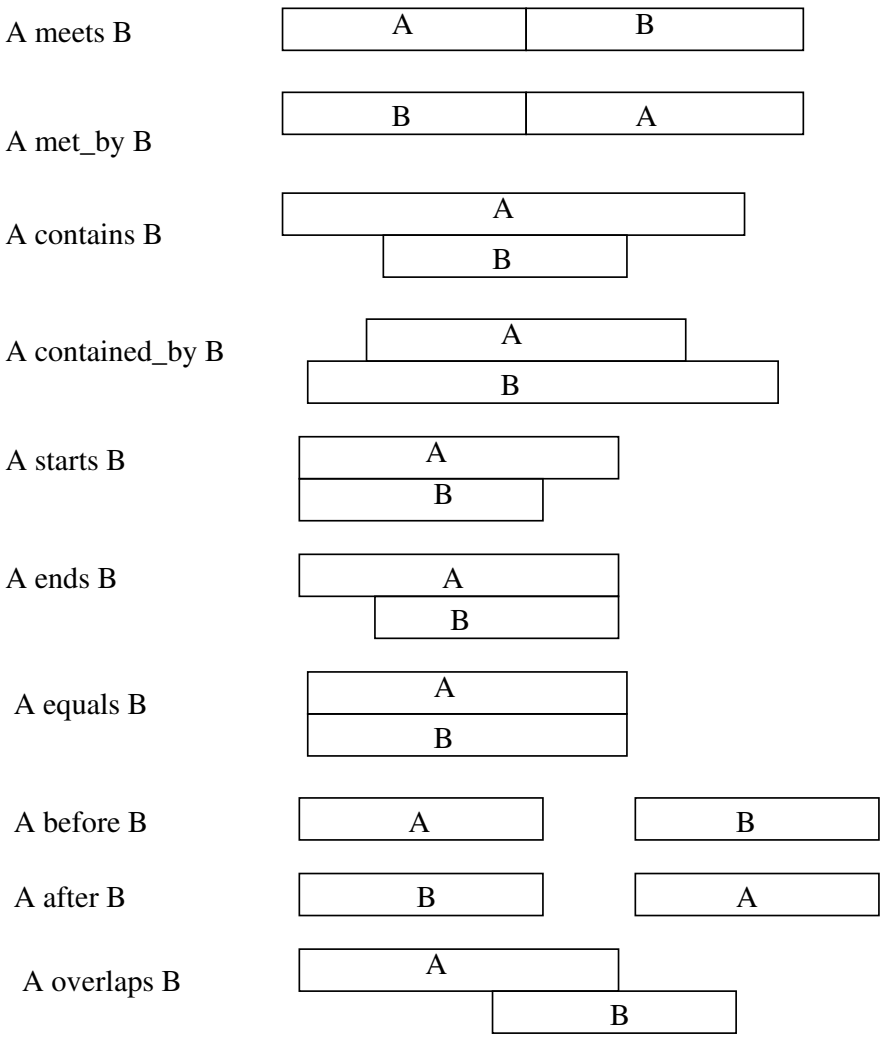


Figure 1: Graphical Definition of Allen Operations

8 Observations and Conclusions

The goal of this work was to develop a capability to translate planning problems specified in the APPL language into the input language of the SAL model checker. This capability was intended to be used in conjunction with another translator from the APPL language to the input language of the EUROPA planning system. Together these translators should enable the formal verification of key properties of a plan specification, especially the specification of a generic domain model.

During the development of this translator, our understanding of plan specification using Allen Operations has greatly increased. We believe we are now in the position to make some observations about the advantages and disadvantages of specifying planning problems using a language built around Allen Operations.

The first observation is that it is easy to construct plan specifications that have no solution. For example, consider the following constraints

```
A meets B
C met_by A
```

These constraints require that `A` must be followed by both `B` and `C`, which is impossible. This might be obvious if the constraints are relatively close to each other in a plan specification, but what if they are several pages apart? Would this contradiction be detected? What might be intended by the above specification is that in some situations one constraint must hold and in another situation the other must hold. This can be accomplished using an if-then-else structure as follows

```
if (condition_1) then
   A meets B
else
   C met_by A
endif
```

Of course, this requires bringing these previously separate specifications together in one place. This, in turn, may make the local specifications less clear.

The second observation is that the FORALL-EXISTS semantics is somewhat non-intuitive. When one first encounters the Allen operations, one would naturally expect that `A meets B` and `B met_by A` are duals. In other words, they specify the same thing. But as shown in Section 7, this is not the case. Therefore, one must be very careful when using Allen ops to keep in mind the FORALL-EXIST meaning and not rely on the English language meaning. Also, the fact that a constraint can be satisfied vacuously can be non-intuitive. If you have

```
A(dog) meets C
```

you can easily fall into the trap of thinking that a solution plan will have an `A(dog)` action that is followed by a `C` action. However, it is perfectly acceptable for the planner to deliver a solution with `A(cat)` and never schedule an `A(dog)`. Furthermore, there are some exceptions

to this FORALL-EXISTS semantics, which complicate our understanding. In particular, at the beginning and end of the planning horizon, the `EXISTS <constraint>` clause need not apply. A good example of this is

```
At meets Going
Going meets At
```

The last `Going` or `At` on a timeline need not satisfy this constraint. Similarly the first `Going` or `At` on a timeline need not satisfy the following constraint:

```
At met_by Going
Going met_by At
```

The third observation is that the use of multiple Allen Operations can have complex interactions. For example:

```
A starts B2
B2 met_by B1
```

Here we declare that every time an `A` starts, a `B2` must also start. But the second constraint requires that `B2` be immediately preceded by `B1`. So the second constraint interacts with the first one as a kind of guard: `A` cannot be scheduled unless `B1` is already executing, and as soon as `A` starts, `B1` must terminate. Note that the declaration

```
A starts B2
B1 meets B2
```

which looks very similar, has a very different kind of interaction. Here, `A` is free to initiate at any time and if `B1` is executing at the time that `A` starts, then the planner could either satisfy the second constraint at this point (by terminating `B1` and starting `B2`) or by letting `B1` continue until a second `B2` executes. All of these possibilities must be considered when writing a domain specification.

The fourth observation is that Allen Operations do not provide a good means of designating a precise sequence of actions. Consider the following APPL construct:

```
A1 -> A2 -> A3 -> A4
```

This declaration provides a concise way of saying exactly what sequence of actions can occur. This can be partially accomplished by the following Allen Operations:

```
A1 meets A2
A2 meets A3
A3 meets A4
```

However, one problem remains. The fact that there can be no successor to `A4` is not specified. Furthermore, merely making `A4` a goal state does not accomplish this because

```
A1 -> A2 -> A3 -> A4 -> A3 -> A4
```

is a solution to the above Allen op specification. In EUROPA one can designate this terminal state by stating that the `A4` must always be at the end of the timeline. But note that this is not a solution that uses Allen ops alone.

The fifth observation is that Allen Operations do not provide a good means of designating a restricted sequence of actions, because they can have non-local interactions and can be spread throughout a specification. Consider the following APPL construct:

```
A1 -> (A2 | A3 | A4) -> A5
```

The following Allen ops can accomplish this

```
if x = 1 then
   A1 meets A2
elsif x = 2 then
   A1 meets A3
else
   A1 meets A4
endif
A2 meets A5
A3 meets A5
A4 meets A5
```

Now suppose we specify, in a different place, that the timeline must begin with `A0` and that `A1` is `met_by A0`:

```
A1 met_by A0
```

Together, one might expect that this specifies that exactly one of the following sequences must occur:

```
A0 -> A1 -> A2 -> A5

A0 -> A1 -> A3 -> A5

A0 -> A1 -> A4 -> A5
```

But this specification does not rule out

```
A0 -> A3 -> A5

A0 -> A4 -> A5

A0 -> A5
```

Of course, the addition of `A0 meets A1` would solve the problem, but this demonstrates how careful one must be when using Allen ops to specify a restricted sequence of operations.

The development of the `APPL -> SAL` translator was driven by the need to match the `APPL -> NDDL` semantics. Several things made this difficult. First, we did not have a formal semantics for NDDL, so we had to infer the meaning of NDDL by executing EUROPA on test problems. The FORALL-EXISTS semantics and end-case exceptions were discovered as we built the translator. Second, we discovered that the EUROPA semantics did not match our original understanding, so there is a structural mis-match between APPL and EUROPA. In particular, we had not originally envisioned that EUROPA employs a kind of *inclusive*[4] approach to specifying the allowed set of actions on a timeline. In EUROPA, any action can be followed by any other action, unless it is prohibited from doing so by a constraint (e.g. an Allen Operation). However, APPL was designed around the idea of a state machine where all of the transitions are enumerated. In this *constructive* approach, an action can only be followed by another action if it is specifically included in the specification. In the end, we were able to bring these two translators fairly close together with the addition of null states, but we are not completely satisfied with our current solution. Third, the interaction of Allen Operations was much more complex than we envisioned. These interactions led to far more complicated code than we originally expected. Furthermore, we often had to rework the translation scheme as we discovered how EUROPA handled these interactions. For that reason, we do not have confidence that the translator is sound. At best, we have produced a rapid prototype which can serve as a basis for future developments. The authors hope that the techniques described in this paper will be useful to future developers who seek to connect planning languages to symbolic model checkers.

References

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

[2] Rick W. Butler and César A. Muñoz. An abstract plan preparation language. Report NASA/TM-2006-214518, NASA Langley, NASA LaRC,Hampton VA 23681-2199, USA, 2006.

[3] Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, CSL Technical Report, 2003.

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

[4] In other words, everything is included except that which is explicitly ruled out by the constraints.

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

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

Solving the AI Planning + Scheduling Problem Using Model Checking via Automatic Translation From the Abstract Plan Preparation Language (APPL) to the Symbolic Analysis Laboratory (SAL)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Butler, Ricky W.; Muñoz, César A.; and Siminiceanu, Radu I.

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-19395

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-2007-215089

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES

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

14. ABSTRACT

This paper describes a translator from a new planning language named the Abstract Plan Preparation Language (APPL) to the Symbolic Analysis Laboratory (SAL) model checker. This translator has been developed in support of the Spacecraft Autonomy for Vehicles and Habitats (SAVH) project sponsored by the Exploration Technology Development Program, which is seeking to mature autonomy technology for the vehicles and operations centers of Project Constellation.

15. SUBJECT TERMS

AI Planning; Autonomy; Formal Methods; Model Checking; 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	47	(301) 621-0390