

# Modeling Hybrid Systems in Action Languages

Richard Watson and Sandeep Chintabathina

Department of Computer Science  
Texas Tech University  
Lubbock, TX 79409, USA  
{rwatson, chintaba}@cs.ttu.edu

**Abstract.** In this paper we present a means of modeling hybrid systems which involve both discrete and continuous time. For the purposes of modeling we use “processes” that are functions dependent on both state and time and show how they can be used on top of existing action languages. To ease computation on current answer set solvers, we use local time instead of global time. The use of such processes in programming is shown by an example.

## 1 Introduction

In the real world we come across many systems that involve both continuous and discrete time. Take for example the behavior of a freely falling body. Suppose the body is held at a certain height from the ground and we drop it at a specific time. The action of dropping the body changes the state of the body from stationary to falling. The body will keep falling until it reaches the ground. The action of dropping the body is discrete in nature in the sense that it happens at some definite time. On the other hand, the height of the body as it descends keeps decreasing with time and is therefore a continuous function of time.

The question is how to model such behavior in a logic programming framework. Approaches to solving this problem have been developed by other researcher such as Reiter [12] and Baral, Son, & Tuan [4]. We present a new approach here for several reasons. The work done by Reiter and his group was based on situation calculus. We use an action language/logic programming [5],[6],[9] approach, we believe has advantages over the situation calculus approach in many cases. None the less, the strong theoretic basis of the situation calculus work gives us a substantially different approach to compare and contrast with. We believe this benefits the science in general. The work by Baral, Son, & Tuan is based on A-Prolog as is our approach. In reading over and applying their work, we found domains we were unable to properly model. We discussed the domain with Tuan and he believed their approach would be unable to model it. We believe the work presented here overcomes those shortcomings,

In order to model hybrid domains, we wished to extend existing action languages (which traditionally have only supported discrete behavior) to allow for continuous behavior as well. In this paper we introduce “Processes” that can be used on top of existing action languages. Processes take the form of recursively

defined function. As will be seen, this provides the ability to represent continuous behavior.

## 2 Language $H$

Action languages are used to model dynamic domains. A domain description (written in an action language) defines a transition diagram for the domain. States of the diagram specify what a rational agent should believe if he was in a situation represented by that state. Arcs of the diagram are labeled by actions. An arc labeled by action,  $A$ , leading from a given state,  $s_0$ , to a state,  $s_1$ , specifies that if the agent performs action  $A$  in the situation represented by  $s_0$ , he will end up in the situation represented by  $s_1$ . Different action languages allow one to represent different types of domains or to reason about the domains in different ways.

In this section we present the action language,  $H$ , which can be viewed as an extension for earlier action languages. For this paper we have extended the action language  $B$  [8], however, the extension is general and should be usable with most other other action languages.

### 2.1 Declarations in $H$

A standard feature of action languages is the *fluent*. Fluents of  $H$  are syntactically the same as in  $B$ , however, the earlier definition of a fluent needs to be clarified. In the past a fluent was normally defined as “a property (of some object in the domain) whose value can change with time.” While this is indeed true, the definition does not traditionally apply to properties whose values change continually with time, but rather those properties which only change when some outside force *causes* them to change. This is what will differentiate between fluents and processes (discussed later).

In terms of the transition diagram, **within a given state the value of a fluent remains constant**. There are two basic types of fluents, *inertial fluents* and *non-inertial fluents*. The difference between these types has to do with their behavior when moving from state to state. The value of a inertial fluent will change if and only if the action (or one of it’s effects) causes it to change. Non-inertial fluents, on the other hand, become undefined in the new state unless that action or its effect cause the fluent to have a value in the new state. In literature you will also find mention of *defined fluents* [7]. They are simply a short-hand for for describing the behavior of specific subclasses of the above types and can always be eliminated.

For use in our action language, and to aid translation, we will declare fluents by giving their name and their type.

$$\text{fluent}(f(c_1, \dots, c_n), \textit{Type}). \quad (1)$$

where a string  $f$  followed by a list,  $c_1, \dots, c_n$ , of zero or more constants is the fluent’s name and  $\textit{Type}$  is the fluent’s type:  $i$  for inertial,  $n$  for non-inertial, or  $d$  for defined.

As mentioned above, the action language descriptions encode the effects of the actions. This defines the arcs of our transition diagram. There are three types of actions: *agent actions*, *exogenous actions*, and *natural actions*. Agent actions are exactly that, actions which can be performed by the agent. Exogenous actions are those actions that can be performed by other entities in domain - either with our without the agent knowing they were performed. Natural actions could be considered to be a type of exogenous action. These are actions that occur automatically as the result of some process. An example of a natural action would be the action of a ball bouncing after it was dropped.

As with fluents, we will declare actions and their types. Such declarations have the form:

$$\text{action}(a(c_1, \dots, c_n), \textit{Type}). \quad (2)$$

where a string  $a$  followed by a list,  $c_1, \dots, c_n$ , of zero or more constants is the action's name and  $\textit{Type}$  is the action's type:  $a$  for agent,  $n$  for natural, or  $ex$  for exogenous. Later we will discuss how the effects of actions are defined.

We now introduce an addition to the standard action language: the *process*. A process is similar to a fluent. It is a property whose value can change over time. It differs, however, in that a process value can change without any other action, fluent, or process causing the change - the change is the caused by the process' normal behavior. If we consider a ball being dropped, the position of the ball would be a process. The position constantly changes within a state and continues from state to state until some action, such as someone catching the ball or the ball bouncing, changes the behavior of the process. Processes are represented by functions which take time, and possible other values, as parameters. Consider the dropping ball example. The position of the ball at a given time in a given state depends on the time since the ball was dropped and the initial position of the ball when it was dropped.

A process' existence is defined by giving the process name:

$$\text{process}(p(c_1, \dots, c_n)). \quad (3)$$

For each process the domain must contain a boolean fluent,  $\text{active}(p(c_1, \dots, c_n))$ , which is true in a state if and only if the process is active in that state.

Constants in  $H$  differ from constants in many previous action languages in that rather than just denoting "names", constants of  $H$  may also denote values. To aid in translation, such "value" constants must be declared. This is done by denoting the name and value:

$$c(c_1, \dots, c_n) = \textit{Value}. \quad (4)$$

Note that a constant's name might contain other constants as "arguments" in its name. For example, if we wanted constants to state that the weight of a specific box,  $box1$ , in our domain was 5 pounds, we could write:  $\text{weight}(box1) = 5$ . If the constant has a boolean value we will simply write  $c(c_1, \dots, c_n)$  or  $\neg c(c_1, \dots, c_n)$ . Integers are considered to be constants whose value is already defined.

As with other action languages, variables will also be allowed in place of constants. To aid translations we will require variables be defined. To define a variable we use the statement:

$$\text{range } q(val_1, \dots, val_n), V_1; \dots; V_m. \quad (5)$$

where  $q$  is a label for the range of values the variables can take,  $val_1, \dots, val_n$ , are the actual values in the range, and  $V_1; \dots; V_m$  are the names of variables which have that range. Variables whose range is a consecutive sequence of integers can be defined by an alternate form:

$$\text{range } q(\text{Min}..Max), V_1; \dots; V_m$$

where  $Min$  and  $Max$  are the minimum and maximum integers in the range. Although not technically required within the action language, when more than one variable name is used, the naming convention for  $V_1; \dots; V_m$  is

$$name; name_1; \dots; name_{m-1}.$$

Note that the use of variable is simply a shorthand. Variables are used in rules of  $H$ , as will be seen below. A rule with variables is a shorthand for the set of rules that is obtained by replacing the variables by each combination of values from variables' ranges.

Together, definitions of the types given above form the *declaration* section of an action description. We now go on to the rules of  $H$ .

## 2.2 Rules of $H$

Before we discuss rules, we must first define some terms that will be used in this section. By definition, we will say a term is a *Regular Function* if it is either a

- 1) number,
- 2) fluent with a numeric value.
- 3) process with a numeric value,
- 4) constant with a numeric value, or
- 5) function of the form  $f(t_1, \dots, t_n)$  where each  $t_1, \dots, t_n$  is a regular function, and  $f$  is an arithmetic function (+, -, \*, /, etc.).

Similarly, a term is a *Boolean Relation* if it is either a

- 1) fluent with a boolean value,
- 2) process with a boolean value,
- 3) the classical negation of an item of type 1 or 2,
- 4) negation as failure applied to an item of type 1 or 2,
- 5) function of the form  $p(t_1, \dots, t_n)$ , where  $p$  is an arithmetic relation (<, >, <=, >=, !=, etc.) and each  $t_1, \dots, t_n$  is a regular function, or
- 6) other functions of the form  $p(t_1, \dots, t_n)$ , where  $p$  is a boolean relation and  $t_1, \dots, t_n$  are fluents, processes, or arguments of the correct type for the relation.

Standard binary functions and relations can be written infix.

We are now ready to continue with rules. The rules of  $H$  can be broken up into three basic types. The three types are rules which state:

- the direct effects of actions,
- constraints on the domain, and
- the behavior of processes.

**Dynamic Causal Laws** We start with those specifying the direct effects of actions - *Dynamic causal laws*. Dynamic causal laws have a similar form to those in  $B$ :

$$A \text{ causes } G = Y \text{ if } W_1, \dots, W_n. \quad (6)$$

where  $A$  is an action,  $G$  is a fluent or process,  $Y$  is a constant value to assign to  $G$ , and  $W_1, \dots, W_n$  are boolean relations. A comma is read as a boolean *and*. If  $n = 0$  then the if can be dropped. If  $Y$  is a boolean value (true, false), then we drop the  $= Y$  and simply write  $F$  or  $\neg F$ . An example of a dynamic cause law is:

$$\text{drop}(\text{glass}) \text{ causes } \text{broken}(\text{glass}) \text{ if } \text{hard\_floor}, \text{height}(\text{glass}) > 5.$$

**Constraints** There are two kinds of constraints: *impossibility constraints* and *state constraints*. Impossibility constraints are used to state conditions under which it is impossible to perform an action. They have the form:

$$A \text{ impossible if } W_1, \dots, W_n. \quad (7)$$

where  $A$  is an action and  $W_1, \dots, W_n$  is a sequence of one or more boolean relations. An example of this form of constraint is:

$$\text{fire\_gun} \text{ impossible if } \neg \text{loaded}.$$

State constraints give conditions which any state must satisfy. State constraints have the form:

$$F = Y \text{ if } W_1, \dots, W_n. \quad (8)$$

where  $Y$  is a value to be assigned to fluent  $F$  and  $W_1, \dots, W_n$  is a sequence of one or more boolean relations. Note: a constant can be considered to be a state constraint with  $n = 0$ . Again, if  $Y$  is boolean, we can drop it as we do with dynamic causal laws. As a special case, we can also write:

$$\text{false if } W_1, \dots, W_n.$$

This form states that any state in which every boolean relation  $W_i$  evaluates to true is not allowed. Examples of state constraints are:

$$\begin{aligned} &\text{lamp\_lit if } \text{bulb\_ok}, \text{power\_on}. \\ &\text{false if } \text{water}(\text{hot}), \text{water}(\text{cold}). \end{aligned}$$

**Processes** Before getting into the rules for defining processes, it will help to first discuss states of the transition diagram. In many previous action languages, a state was simply a complete valuation of the fluents of the domain. A state was a static object. The “real time” was left out. The normal association of state to time was that the state started at the time just after the initial sequence of actions leading to the state was performed. It ended when the next action in the sequence was performed.

We now wish to include processes. Imagine a state that was started by an agent action “drop\_glass” and ended by a natural action “glass\_hits\_floor”. The state has a definite time length which depends on the height of the glass when dropped and the pull of gravity. This is fine in theory, but in practice it is quite difficult to implement. In the example of the dropped glass, it is easy to compute when the natural action occurs. Imagine, however, a different process defined by a complex mathematic formula and a natural action should occur the first time the value drops below a threshold. This can be quite difficult.

We must also take implementation into consideration. As will be seen later, we will translate domain descriptions of  $H$  into A-Prolog logic programs. If we wish to talk about the position of the glass in the air, we have an infinite number of groundings. Alternately, we could leave the process uncomputed until queried, but then we have the problem of when actions occur based on the process.

We chose a compromise which eases computation while still allowing reasonable process definitions. We simply discretized time into equal length segments. This leads to some approximation in the value of functions, but at the same time gives us something easily computable. This choice also leads to a nice form for representing processes. As will be seen below, processes will be represented using recursive functions. The value of the process at some time in the state will be computed based on the value at the previous time segment.

Another point to recall is that for each process,  $P$ , we have a fluent  $active(P)$ . The rules for processes will only be applied if the process is active, i.e. when  $active(P) = true$ , therefore there must be a dynamic causal law or state constraint that make the process active. The process will continue until such time that a dynamic causal law or state constraint makes it inactive.

We will have two types of rules concerning the effects of processes. The first concerns how the value of a process changes from one time segment to the next. Such rules, called *process update rules*, have the form:

$$P = Y \text{ after } W_1, \dots, W_n. \quad (9)$$

Where  $P$  is a process,  $Y$  is a value, and each  $W_i$  is a boolean relation. Again, if  $Y$  is a boolean value the  $= Y$  can be dropped. An example of such a process is a timer that counts down one tick per unit of time until it reaches zero. This can be represented by the rule:

$$timer = X - 1 \text{ after } timer = X, X > 0.$$

Notice that, as is usual, the process occurs on both sides of the rule.

The second type of rule has to do with when a process causes a “natural” action to occur. These rules have the form:

$$\text{initiated } A \text{ if } U(P), W_1, \dots, W_n. \quad (10)$$

where  $A$  is an action,  $U$  is a boolean relation concerning the process  $P$ , and  $W_1, \dots, W_n$  are other boolean relations. Such rules are called *natural action rules*. An example of this rule would be:

$$\text{initiated } \textit{alarm} \text{ if } \textit{timer} = 0.$$

Both the rules above require that the process is active. We do not, however, require that the fluent  $\textit{active}(P)$  occur on the right side of either rule - it is assumed.

### 3 Example Domain Description

We will now give a complete example of a domain description. Imagine we wish to model the behavior of a traffic light. We will have three lights: red, yellow, and green. Each light will have an associated timer. If there is power and a timer for a color is active, the light should be lit (unless it is burnt out).

We will start out with some definitions. First we will designate the maximum time on each timer:

$$\begin{aligned} \text{max\_timer}(r) &= 8. \\ \text{max\_timer}(g) &= 12. \\ \text{max\_timer}(y) &= 2. \end{aligned}$$

Next we will give facts for which color light follows which:

$$\begin{aligned} \text{next}(r,g). \\ \text{next}(g,y). \\ \text{next}(y,r). \end{aligned}$$

Then we define a variable for colors

$$\text{range color}(r,y,g), C.$$

Next we define our fluents. We will have defined fluents to indicate if the light of each color is lit and inertial fluents indicating if system is on, bulbs are burnt out, and the power is on.

$$\begin{aligned} \text{fluent}(\textit{lit}(C),d). \\ \text{fluent}(\textit{system\_on},i). \\ \text{fluent}(\textit{burnt\_out}(C),i). \\ \text{fluent}(\textit{power\_on},i). \end{aligned}$$

In addition, there will also be “ $\textit{active}(P)$ ” fluents for each process.

For actions, the agent can turn the system on or off, reset the power, or change a bulb. There is a natural action of one timer cycling over to the next, and exogenous actions or burning out a bulb, or a power outage occurring.

$$\begin{aligned} \text{action}(\textit{turn\_system\_on},a). \\ \text{action}(\textit{turn\_system\_off},a). \\ \text{action}(\textit{reset\_power},a). \end{aligned}$$

```

action(change_bulb(C),a).
action(cycle_light(C),n).
action(burn_out(C),ex).
action(outage,ex).

```

Our only processes will be a timer for each light color.

```

process(timer(C)).

```

We are now ready for dynamic causal laws. The first three rules are concerned with the system being turned on. When the system is turned on, the red light comes on for the maximum duration. These rules all require the power to be on, if it isn't the action does nothing.

```

turn_system_on causes system_on if power_on.
turn_system_on causes timer(r) = max_timer(r) if power_on.
turn_system_on causes active(r) if power_on.

```

The next two actions state that when the system is turned off, it is 1) not on, and 2) deactivates any active timer.

```

turn_system_off causes -system_on.
turn_system_off causes -active(C).

```

Next, resetting the power causes the power to be on.

```

reset_power causes power_on.

```

Changing a bulb of a certain color causes that bulb to not be burnt out.

```

change_bulb(C) causes -burnt_out(C).

```

The natural action of one timer cycling to the next causes the new timer to be activated and set to its maximum time and the old timer to deactivate.

```

cycle_light(C) causes timer(C) = max_timer(C).
cycle_light(C) causes active(C).
cycle_light(C2) causes -active(C1) if next(C1,C2).

```

And the exogenous actions cause a bulb of a certain color to be burnt out or for the power to turn off.

```

burn_out(C) causes burnt_out(C).
outage causes -power_on.

```

Next we have the following state constraints:

```

reset_power impossible_if system_on.
lit(C) if active(C), power_on, -burnt_out(C).

```

The first says you can't reset the power while the system is on. The second says that if the timer for a color is active, the bulb is not burnt out and there is power, then the bulb will be lit.

Finally, rules for processes. Since the processes are similar for each color, we only need two rules (with variables). The first rule states that if an active timer reaches 0, it will initiate an action to start the next timer in the sequence.

```

initiated cycle_light(C2) if timer(C1) = 0, next(C1,C2).
timer(C) = X-1 after timer(C) = X, X > 0.

```

Second rule states that if an active timer has a value greater than 0, it ticks down by one in the next time step.

This completes the example. We will discuss models of this domain after we give the translation to logic programming.

## 4 The Initial Situation and History

To use a domain description,  $D$ , we must also be able to state what is true initially, what actions the agent knows happened, and when the actions happened. We state the initial values of fluents and processes using statements of the form:

$$\text{initially } G = Y. \quad (11)$$

where  $G$  is a fluent or process and  $Y$  is its initial value. Boolean values are treated as before. To state the agent's knowledge about the occurrence of an action we use a statement:

$$\text{occurs } A \text{ at } T. \quad (12)$$

where  $A$  is an action and  $T$  is a numeric constant stating the number of time steps since the initial situation at which the action occurred. While this second statement has the same syntactic form as languages such as action query language  $Q$  [8], it differs in that in most prior languages,  $T$  represented the number of a state. This initial situation and history is referred to as  $\Gamma$ . What is true with respect to  $D$  and  $\Gamma$  can then be determined by looking at the trajectories (paths) within the transition diagram (obtained from  $D$ ) which satisfy  $\Gamma$ .

## 5 Semantics of $H$

Due to space restrictions, precise semantics of  $H$  will not be given here. The semantics, however, are based on those from [10] and are very similar to those of the language  $B$ . A domain description,  $D$ , of  $H$  defines a transition diagram. Roughly speaking, states of the transition diagram contain valuations for each fluent and process. Arcs of the transition diagram are labeled by actions. An arc from  $s$  to  $s'$  labeled by action  $a$  denotes that performing action  $a$  in the situation corresponding to  $s$  may result in the agent being in the situation corresponding to  $s'$ . The word *may* is used since if there are multiple arcs leading from  $s$  labeled by  $a$  the result of performing  $a$  is non-deterministic.

Given a domain description,  $D$ , and a history  $\Gamma$ , we can define a consequence relation. A fluent or process having a given value is a consequence of  $D$  and  $\Gamma$  if every trajectory (path) in the transition diagram of  $D$  which satisfies  $\Gamma$  ends in a state in which the fluent (process) has that given value. For clear and precise definitions of how such semantics were specified for  $B$  (and other similar languages) see [8].

## 6 Translation into a Logic Program

We will now discuss the translation from  $H$  to A-Prolog. We use SMOBELS [11] to compute answers, so the translation given will be in the syntax of that inference engine. Given a domain description,  $D$ , written in  $H$ , we will create a logic program  $\Pi(D)$  which will allow for computation.

No translation is needed for the first rules - (1), (2), and (3) remain the same in  $\Pi(D)$ . We will, however, add other rules to define variables for each set of names. For each rule of type (1),  $fluent(N, V)$ , we will add a rule

$$fluent\_name(N).$$

and for each rule of type 2,  $action(N, V)$ , we will add:

$$fluent\_name(N).$$

We then add the following rules:

$$\begin{aligned} &\#domain\ fluent\_name(F). \\ &\#domain\ action\_name(A; A1; A2). \\ &\#domain\ process(P; P1; P2). \end{aligned}$$

These will be used as range restricted variable in later rules.

Rules of type (4),  $C(c_1, \dots, c_n) = Value$  will be replaced in  $\Pi(D)$  by a rule:

$$C(c_1, \dots, c_n, Value).$$

and later, when converting other rules, if  $C(c_1, \dots, c_n)$  occurs in a boolean relation, we add to the right-hand-side of the rule,  $C(c_1, \dots, c_n, V)$  (where  $V$  is a new, undefined variable) and replace the occurrence of  $C(c_1, \dots, c_n)$  by  $V$ .

Each type (5) rule, range  $Q(Val_1, \dots, Val_n), V_1; \dots; V_m$  is replaced by rules:

$$\begin{aligned} &Q\_range(Val_1). \\ &\quad \vdots \\ &Q\_range(Val_n). \end{aligned}$$

and

$$\#domain\ Q\_range(V_1; \dots; V_m).$$

In the case where rather than having  $(Val_1, \dots, Val_n)$ , we have  $(Min..Max)$ , instead of the first  $n$  rules we only need one:

$$Q\_range(Min..Max).$$

Before we go onto the rules of type (6), dynamic causal laws, we need to present how instances of fluents, processes, and actions, are treated when they appear in rules. First, we will assume some special, range restricted variables. They are  $S, T, T1$ , and  $T2$ . The range of  $S$  is over state numbers, from 0 to the maximum.  $T, T1$ , and  $T2$  range over time point numbers, from 0 to the maximum number of time points per single state. The range definition for these variables is not fixed within the domain, it is part of the initial situation and history,  $\Gamma$ .

Next we show how fluents, processes, and occurring in rules of  $D$  are handled. Actions are the easiest, when an action,  $A$ , is in a rule being translated, during the translation it is replaced by:

$$o(A, S, T).$$

For fluents and processes, the translation will depend on if it occurs in the head or the body of a rule, if its value is boolean or not, and what type of rule it occurs in. Although not explicitly stated in the domain description, whether the range of a fluent or process in boolean or not can be determined by usage.

First we will look at boolean fluents. If a boolean fluent,  $F$  occurs in the body a rule or in the head of a state constraint, it is replaced by

$$h(F, S)$$

If it occurs in the head of a dynamic causal law, it is replaced by

$$h(F, S + 1)$$

Translation for value based is slightly more complex as they will necessarily occur in some formula. If the fluent occurs in the head of a rule, it must have the form  $F = Y$  where  $Y$  is a constant. We simply replace this by either

$$val(Y, F, S) \text{ or } val(Y, F, S + 1)$$

depending, as above, on if the rule is a dynamic causal law or not. If it occurs in the form  $F = Y$  in the body and  $Y$  is a constant, we again replace it with

$$val(Y, F, S).$$

All other occurrence must occur in some boolean relation,  $p(V_1, \dots, F, \dots, V_n)$ . We add to the right side of the rule,

$$val(W, F, S)$$

where  $W$  is a new variable, and we replace  $F$  in the relation  $p$  by  $W$ .

Processes are translated similarly to except for three things: processes can't appear in the heads of dynamic causal laws, they rely on time as well as state, and, unlike fluents, they do appear in the head of process update rules. The translations are as above except an additional parameter,  $T$ , is added as the last parameter. The one exception is when the process occurs in the head of a process update rule. In this case  $T + 1$  is added instead of  $T$ . All remaining boolean relations are left unchanged in  $\Pi(D)$ .

With these translations in mind, we move on to the translations for the remaining rules. Dynamic causal laws  $A$  causes  $F = Y$  if  $W_1, \dots, W_n$ . is replaced by a new rule

$$X : -o(A, S, T), W'_1, \dots, W'_n,$$

where  $X$  is the translation of  $F = Y$  and each  $W'_i$  is the translation (as above) of  $W_i$ .

Constraints of type (7), are replaced by a rule:

$$: -o(A, S, T), W'_1, \dots, W'_n,$$

where again each  $W'_i$  is the translation of  $W_i$ .

Rules of type (8) and (9),  $G = Y$  if  $W_1, \dots, W_n$  are translated as

$$X : -W'_1, \dots, W'_n,$$

and rules of type (10), initiated  $A$  if  $U(P), W_1, \dots, W_n$ . are translated as

$$o(A, S, T) : -X, W'_1, \dots, W'_n,$$

where  $X$  and  $W'_i$ 's are as with dynamic causal laws.

We also need to translate the rules of the initial situation and history,  $\Gamma$ . For rules of type (11), initially  $G = Y$ , we translate the fluent or process as before except we replace  $S$  and  $T$  by 0.

Finally, rules of type (12), occurs  $A$  at  $Val$  are replaced by

$$occ(A, Val).$$

## 6.1 Domain independent axioms

Simply translating the program is not enough to allow us to use automated computation methods. What follows will be domain independent axioms which will be added to any domain being translated. Due to there domain independent nature, no changes are needed to the code.

**End of State Axioms** We start with some axillary actions which will be used in later rules. These axioms record the time step when each state ends.

The first rule states that, if nothing else, at state can always end at the last possible time unit in the state.

$$0end(S, num\_time\_units)1.$$

Next, a rule that states that an action ends at the time unit an action was performed in the stale.

$$end(S, T) : -o(A, S, T).$$

Then we have a rule that states that a state cannot have more than one end time.

$$: -end(S, T1), end(S, T2), neq(T1, T2).$$

Finally, two rules which state that: 1) a state ends if it has an end time and 2) every state must end.

$$ends(S) : -end(S, T).$$

$$: -notends(S).$$

**Inertia Axioms** Inertia axioms are common in translations from action languages to logic programming. These rules are used to handle those things that don't change when moving from one state to the next.

The first two rules are for boolean fluents. They state that, for any inertial fluent, unless the value must change (because of an action or state constraint) then it remains the same in the next state.

$$\begin{aligned} h(F,S+1) &:- \text{fluent}(F,i), h(F,S), \text{not } \neg h(F,S+1). \\ \neg h(F,S+1) &:- \text{fluent}(F,i), \neg h(F,S), \text{not } h(F,S+1). \end{aligned}$$

We have two similar rules for processes. These rules say that the value of a boolean process at the beginning of a new state is the same as it was at the end of the previous state unless something changed it.

$$\begin{aligned} h(P,S+1,0) &:- h(P,S,T), \text{end}(S,T), \text{not } \neg h(P,S+1,T). \\ \neg h(P,S+1,0) &:- \neg h(P,S,T), \text{end}(S,T), \text{not } h(P,S+1,T). \end{aligned}$$

For non-boolean fluents and processes we only have one rule each. The meaning of these rules is similar to the ones above. For fluents the rule is

$$\text{val}(Y,P,S+1) :- \text{val}(Y,P,S), \text{not } \neg \text{val}(Y,P,S).$$

The rule for processes is

$$\text{val}(Y,P,S+1,0) :- \text{end}(S,T), \text{val}(Y,P,S,T), \text{not } \neg h(\text{active}(P),S+1).$$

**Real Time Axioms** A set of real time axioms are included. These convert statements about occurrences of actions based on number of time steps since the initial situation to action occurrence based on a state/time pair.

This first rule actually isn't a domain independent rule. It is one required to be added to the initial situation by the user. It defines the maximum number that can be used for a "real time".  $\#constrealJen = Val$ .

where  $Val$  is a number. Although smaller values are often sufficient in practice, the largest possible value needed for  $Val$  is the number of states multiplied by the maximum number of time points per state.

The next two rules define the range that a real time can fall within and define  $R$  as a variable in that range.

$$\begin{aligned} &\text{realtime}(0..realJen). \\ &\#domainrealtime(R). \end{aligned}$$

The following 2 rules compute the time until the occurrence of a real time action from the beginning of the next state given the time from the beginning of the current state and the end time of the current state. The first of the rules is for cases when the time is too far in the future to occur in the current state. The second rule handles cases when the time until the occurrence is short enough to occur, but the state ends before that time.

$$\begin{aligned} \text{occ}(A, R - T, S + 1) &:- \text{occ}(A, R, S), R > \text{num\_times}, \text{end}(S, T). \\ \text{occ}(A, T1 - T2, S + 1) &:- \text{occ}(A, T1, S), T1 > T2, \text{end}(S, T2). \end{aligned}$$

The following 3 rules together state that "if the occurrence of a real time action can occur in the current state then it must occur in the state."

$$\begin{aligned} \text{Oo}(A, S, T)1 &:- \text{occ}(A, T, S), T \leq \text{num\_times}. \\ &:- \text{occ}(A, T1, S), \text{end}(S, T2), T1 < T2. \\ &:- \text{occ}(A, T, S), \text{end}(S, T), \text{noto}(A, S, T). \end{aligned}$$

## 7 Results of Computation

A sample initial situation was created in which the system was immediately turned on, there were 10 steps and 10 time points maximum per step. The total possible “real time” was set at 100. The program was then run on a couple different platforms. One was a Sparc Ultra 10 running Solaris 8 and the other was a PC with 512 MB ram, an Athlon XP 1800+ processor, and running Red Hat Linux version 9.0. Both machines were using the latest versions of `lparse` (1.0.13) and `S MODELS` (2.27). The answer set returned from the program was as expected. Run times for both machines was in the 25 to 29 second range.

## 8 Conclusions and Future Work

In this paper we have seen how one can model continuous behavior using processes. These processes can be used on top of many existing action languages. There is a simple and elegant transformation to A-Prolog and efficient computation under `S MODELS`. There is still work to be done though.

One area for future work is diagnostics. We have already performed some preliminary tests using methods from [1] and [2]. Results were good. In the traffic light example, diagnosing possible causes for a light being out when it should have been lit were found in less than half a minute. There are some open questions however. Whether using a diagnostic module or consistency restoring rules, it is unclear how to perform diagnosis in some situations. With continuous functions it may be possible that the only diagnoses are actions which must occur at a time other than one of the discrete times.

Another area for further research is delayed grounding. If functional values and times could be grounded only if and when needed it could reduce the program size and allow spacing of time points to fluctuate as needed.

We need to work on more examples to find out if there are any pitfalls which were not encountered in our brief trials so far. A more in-depth analysis needs to be done between this approach, Reiter’s approach, and Baral’s approach. Michael Gelfond also has another, yet unpublished, approach to this problem. We will be working with him to compare the approaches and hopefully converge on a single approach combining the best ideas from each.

## 9 Acknowledgments

First and foremost, the authors would like to thank Michael Gelfond. This work grew out of numerous discussions we had together. The authors also wish to thank United Space Alliance and both NASA Ames Research Center and NASA Johnson Space Center whose research grants, in part, funded this research.

## References

1. [BG03] M. Balduccini and M. Gelfond. Diagnostic reasoning with A-Prolog. In *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4-5):425-461, Jul 2003.
2. [BG03a] M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In *AAAI Spring 2003 Symposium*, 2003.
3. [BG00] C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In Minker, J., ed., *Logic-Based AI*, Kluwer Academic publishers, (2000), 257-279.
4. [BST02] C. Baral, T. Son and L. Tuan. A transition function based characterization of actions with delayed and continuous effects. In *Proceedings of KR'02*, pages 291-302.
5. [GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming, In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, 1988, pp. 1070-1080.
6. [GL98] M. Gelfond and V. Lifschitz. Action Languages. In *Electronic Transactions on Artificial Intelligence*, 3(6), 1998.
7. [GW98] M. Gelfond and R. Watson. On Methodology of Representing Knowledge in Dynamic Domains. In *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, pp. 57-66, 1999.
8. [Lif97] V. Lifschitz, Two components of an action language, In *Annals of Mathematics and Artificial Intelligence*, Vol. 21, 1997, pp. 305-320.
9. [Lif99] V. Lifschitz. Action languages, Answer Sets and planning. In *The Logic Programming Paradigm: a 25 year perspective*. 357-373, Springer Verlag, 1999.
10. [MT95] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proceedings of IJCAI'95*. pp1978-1984, 1995.
11. [NS97] I. Niemela and P. Simons. Smodels - an implementation of the stable model and well founded semantics for normal logic programs. In *Proceedings of LPNMR'97*, pages 420-429, 1997.
12. [Rei96] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2-13, Cambridge, Massachusetts, U.S.A., November 1996.