Modeling Hybrid Domains Using Process Description Language

Sandeep Chintabathina

Master's Thesis

December 15, 2004

ACKNOWLEDGEMENTS

I would like to thank my parents and my sister for their support and good wishes. They have always been my reason to do well. Mother and Father, I would like to express to you my gratitude in your support as I pursue my college education. And Akka, I have learned a lot of things from you over the years. Thank you for everything.

I would like to thank Dr. Gelfond and Dr. Watson for making this thesis possible. You guys are good at what you do and also nice to work with. You have constantly pushed me and got the best out of me. Thank you for your guidance and support.

Dr. Gelfond working with you is an unforgettable experience. I am amazed by your ability to concentrate. Your work ethic and moral values will always inspire me. I have learned so much from you. You always make time for me even when you are so busy. You worked almost everyday with me to finish this thesis. I really appreciate it. Thank you for all your suggestions and comments.

Dr. Watson you are a very open-minded and down to earth person. You hired me as a research assistant at a time when I desperately needed funding. Thank you for giving me an opportunity. I am glad that I will be one of your first students to graduate. I am looking forward to working with you in future research projects.

And to all my friends at Texas Tech University, a big thank you for cheering me up whenever I am feeling down.

Ricardo you are my best friend ever. Whether it is working with you or playing with you or just hanging out I enjoy every bit of it. Thank you for being there whenever I needed you.

A big thanks to Sunil and all my ex-roommates for bearing with me. Sunil, we had a great time as roommates, I will never forget that.

I would like to thank all the KR lab members. Each and everyone has contributed in some way or an other towards this thesis. Its nice to be part of a very active and smart group of researchers. I enjoy working with you all.

I would like to thank the faculty and staff of computer science department for their support and cooperation. Finally, I would like to thank NASA Johnson space center for funding our research projects.

TABLE OF CONTENTS

ABSTRACT

Researchers in the field of knowledge representation and logic programming are constantly trying to come up with better ways to represent knowledge. One of the recent attempts is to model dynamic domains. A dynamic domain consists of actions that are capable of changing the properties of objects in the domain, for example the blocks world domain. Such domains can be modeled by action theories - collection of statements in so called action languages specifically designed for this purpose. In this thesis we extend this work to allow for *continuous processes* - properties of objects that change continuously with time. For example the height of a freely falling object. In order to do this we adopt an action language/logic programming approach.

A new action language called *process description language* is introduced that will be useful to model systems that exhibit both continuous and discrete behavior (also called *hybrid systems*). An example of a hybrid domain is the domain consisting of a freely falling object. A freely falling object is in the state of $falling$, which is a discrete property that can be changed only by actions (also called fluent) while its *height* is a continuous process.

The syntax, semantics, and translation of the statements of the language into rules of a logic program will be discussed. Examples of domains that can represented in this language will be given. In addition, some planning and diagnostic problems will be discussed. Finally, the language will be compared with other languages used for similar purposes.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Designing an intelligent agent capable of reasoning, planning and acting in a changing environment is one of the important research areas in the field of AI. Such an agent should have knowledge about the domain in which it is intended to act and its capabilities and goals. In this thesis we are interested in agents which view the world as a dynamical system represented by a transition diagram whose nodes correspond to possible physical states of the world and whose arcs are labeled by actions. A link, (s_0, a, s_1) of a diagram indicates that action a is executable in s_0 and that after the execution of a in s_0 the system may move to state s_1. Various approaches to representation of such diagrams [3; 6; 9] can be classified by languages used for their description. In this thesis we will adopt the approach in which the diagrams are represented by action theories - collections of statement in so called action languages specifically designed for this purpose. This approach allows for useful classification of dynamical systems and for the methodology of design and implementation of deliberative agents based on answer set programming.

Most of this work, with the notable exception of [4; 14], deals with discrete dynamical systems. A state of such a system consists of a set of *fluents* - properties of the domain whose values can only be changed by actions. An example of a fluent would be the position of an electrical switch. The position of the switch can be changed only when an external force causes it to change. Once changed, it stays in that position until it is changed yet again. The corresponding transitions in the diagram are shown in Figure 1.1.

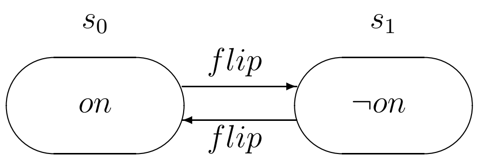


Figure 1.1: Transitions caused by $flip$

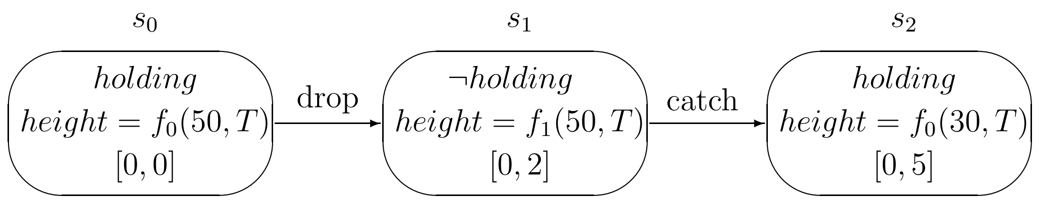
Action languages will describe the diagram in Figure 1.1 by so called dynamic causal laws of the form:

$$flip \;\; causes \;\; \neg on \;\; if \;\; on. \tag{1.1}$$

$$flip \;\; causes \;\; on \;\; if \;\; \neg on. \tag{1.2}$$

(1.1) says that performing the action $flip$ causes the position of the switch to be off if it was *on*. (1.2) says that performing the action $flip$ causes the position of the switch to be *on* if it was off.

In this thesis we focus on the design of action languages capable of describing dynamical systems which allow *continuous processes* - properties of an object whose values change continuously with time. Example of such a process would be the function, *height* of a freely falling object. Suppose that a ball, 50 meters above the ground is dropped. The height of the ball at any time is determined by Newton's laws of motion. The height varies continuously with time until someone catches the ball before it reaches the ground. Suppose that the ball was caught after 2 seconds. Assume that there is only one arm that drops and catches the ball. The corresponding transition diagram will contain transitions of the form:



where f_0 and f_1 are defined as:

$$f_0(Y, T) = Y. \qquad f_1(Y, T) = Y - \tfrac{1}{2}gT^2.$$

Figure 1.2: Transitions caused by *drop* and *catch*

Notice that states of this diagram are represented by mapping of values to the symbols *holding* and *height* over corresponding intervals of time. For example in state s_1, *holding* is mapped to false and *height* is defined by the function $f_1(50, T)$ where T ranges over the interval $[0, 2]$.

Intuitively, the time interval of a state s denotes the time lapse between occurrences of actions. The lower bound of the interval denotes start time of s which is the time at which an action initiates s. The upper bound denotes the end time of s which is the time at which an action teminates s. We assume that actions are instantaneous that is the actual duration is negligible with respect to the duration of the units of time in our domain. For computability reasons, we assign local time to states. Therefore, the start time of every state s is 0 and the end time of s is the time since the start of s till the occurrence of an action terminating s. For example, in Figure 1.2 the action *drop* occurs after a time lapse of 0 units since the start of state s_0. Therefore, the end time of s_0 is 0. The action *catch* occurs after a time lapse of 2 units since the start of state s_1. Therefore the end time of s_1 is 2.

The state s_2 in Figure 1.2 has the interval $[0, 5]$ associated with it. This interval was chosen randomly from an arbitrary collection of intervals of the form [0,n] where $n \geq 0$. Therefore, any of the intervals $[0, 0]$ or $[0, 1]$ or $[0, 2]$ and so on could have been associated with s_2. In other words, performing *catch* leads to an infinite collection of states which differ from each other in their durations. The common feature among all these states is that *height* is defined by $f_0(30, T)$ and *holding* is true. We do not allow the interval $[0, \infty]$ for any state. We assume that every state is associated with two symbols - 0 and *end*. The constant 0 denotes the start time of the state and the symbol *end* denotes the end time of the state. We will give an accurate definition of *end* when we discuss the syntax of the language.

We assume that there is a global clock which is a function that maps every local time point into global time. Figure 1.3 shows this mapping. Notice that this mapping allows one to compute the *height* of the ball at any global time, $t \in [\mathbf{0}, \mathbf{7}]$. This is not necessarily true for the value of *holding*. According to our mapping global time $\mathbf{0}$ corresponds to two local times: 0 in state s_0 and 0 in state s_1. Since the values of *holding* in s_0 and s_1 are *true* and *false* respectively, the global value of *holding* at

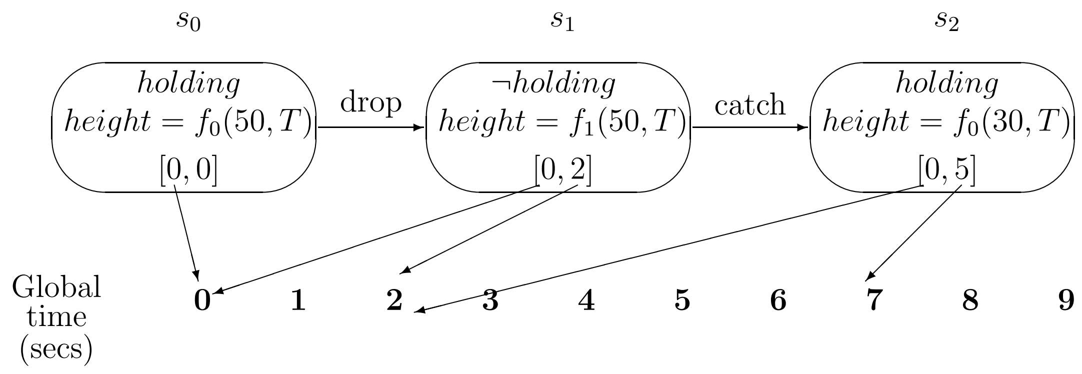


Figure 1.3: Mapping between local and global time

global time **0** is not uniquely defined. Similar behavior can be observed at global time **2**. The phenomena is caused by the presence of (physically impossible) instantaneous actions in the model. It indicates that **0** and **2** are the points of transition at which the value of *holding* is changed from *true* to *false* and *false* to *true* respectively. Therefore, it is $false$ at **1** and *true* during the interval [**3**,**7**].

Since the instantaneous actions *drop* and *catch* do not have a direct effect on *height*, its value at global time **0** and **2** is preserved, thereby resulting in unique values for *height* for every $t \in [\mathbf{0}, \mathbf{7}]$.

Our new action language H, also called as *process description language*, will describe these transitions by defining the continuous process *height*, fluent *holding*, functions $f_0(Y,T)$, $f_1(Y,T)$, and actions *drop* and *catch*. The effects of the action *drop* will be given by the causal law:

$$drop \;\; causes \;\; \neg holding. \tag{1.3}$$

which says that performing the action *drop* at time *end* in a state, s, causes *holding* to be *false* in the successor state of s. This explains the change in the value of *holding* from s_0 to s_1. The executabilty conditions will have the form:

$$impossible \;\; drop \;\; if \;\; \neg holding. \tag{1.4}$$

which says that the ball cannot be dropped at time *end* in a state, s, if holding the ball is false. Therefore, the action *drop* cannot be performed in the state s_1.

$$impossible \;\; drop \;\; if \;\; height(end) \;\; = \;\; 0. \tag{1.5}$$

says that the ball cannot be dropped at time *end* in a state, s, if it is on the ground at *end*. $height(end)$ is a special fluent corresponding to the continuous process *height* that denotes the *height* at the *end* of a state. The effects of the action *catch* are given by the causal law:

$$catch \;\; causes \;\; holding. \tag{1.6}$$

(1.6) explains why there is a change in the value of *holding* from s_1 to s_2. The executablity conditions will have the form:

$$impossible \;\; catch \;\; if \;\; holding. \tag{1.7}$$

(1.7) explains why the action *catch* cannot be performed in the states s_0 and s_2. *height* is defined by the following statements:

$$\begin{aligned} height = f_0(Y,T) \;\; if \;\; & height(0) = Y, \\ & holding. \end{aligned} \tag{1.8}$$

From Figure 1.3 it is obvious that the value of *height* is determined depending on whether *holding* is true or not. Statement (1.8) requires that in any state in which *holding* is true, *height* does not change with time. *height*(0) is a special fluent corresponding to continuous process *height* that denotes the *height* at time 0 of a state. If *holding* is *false* then *height* is defined as follows:

$$\begin{aligned} height = f_1(Y, T) \;\; if \;\; & height(0) = Y, \\ & \neg holding. \end{aligned} \tag{1.9}$$

Statement (1.9) requires that in any state in which *holding* is false, *height* is defined by Newtonian equations.

In the next chapter we will discuss the syntax and semantics of the language and see some more examples.

CHAPTER 2

SYNTAX AND SEMANTICS

2.1 Syntax

To define our language we first need to fix a collection Δ of time points. Normally Δ will be equal to the set, R^+, of non-negative real numbers, but we can as well use integers, rational numbers, etc. We will use the variable T for the elements of Δ. We will also need a collection, $\mathcal{G}$, of functions defined on Δ, which we will use to define continuous processes. Elements of $\mathcal{G}$ will be denoted by lower case greek letters α, β, etc.

A process description language, $H(\Sigma, \mathcal{G}, \Delta)$, will be parameterized by Δ, $\mathcal{G}$ and a typed signature Σ. Whenever possible the parameters Σ, $\mathcal{G}$, Δ will be omitted. We assume that Σ contains regular mathematical symbols including $0, 1, +, <, \leq, \geq, \neq, *, etc.$ In addition, it contains two special classes, $\mathcal{A}$ and $\mathcal{P} = \mathcal{F} \cup \mathcal{C}$ of symbols called *actions* and *processes*.

Elements of $\mathcal{A}$ are elementary actions. A set $\{a_1, \ldots, a_n\}$ of elementary actions performed simultaneously is called a *compound* action. By actions we mean both elementary and compound actions. Actions will be denoted by a's. Two types of actions - *agent* and *exogenous* are allowed. *agent* actions are performed by an agent and *exogenous* actions performed by nature.

Processes from $\mathcal{F}$ are called *fluents* while those from $\mathcal{C}$ are referred to as *continuous processes*. Elements of $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{C}$ will be denoted by (possibly indexed) letters p's, k's and c's respectively.

$\mathcal{F}$ contains a special functional fluent *end* that maps to Δ. *end* will be used to denote the end time of a state. We assume that for every continous process, $c \in \mathcal{C}$, $\mathcal{F}$ contains two special fluents, $c(0)$ and $c(end)$. For example, the fluents $height(0)$ and $height(end)$ corresponding to *height*.

Each process $p \in \mathcal{P}$ will be associated with a set $range(p)$ of objects referred to as the *range* of p. E.g. $range(height) = R^+$.

Atoms of $H(\Sigma, \mathcal{G}, \Delta)$ are divided into *regular* atoms, *c-atoms* and *f-atoms*.

- *regular* atoms are defined as usual from symbols belonging to neither $\mathcal{A}$ nor $\mathcal{P}$.

  E.g. mother(X,Y), sqrt(X)=Y.

- *c-atoms* are of the form $c = \alpha$ where $range(c) = range(\alpha)$.

  E.g. $height = 0$, $height = f_0(Y, T)$, $height = f_0(50, T)$.

  Note that α is strictly a function of time. Therefore, any variable occurring in a *c-atom* other than T is grounded.

  E.g. $height = f_0(Y, T)$ is a schema for $height = \lambda T.f_0(y, T)$ where y is a constant. $height = 0$ is a schema for $height = \lambda T.0$ where $\lambda T.0$ denotes the constant function 0.

- *f-atoms* are of the form $k = y$ where $y \in range(k)$. If k is boolean, i.e.

$range(k) = \{\top, \bot\}$ then $k = \top$ and $k = \bot$ will be written simply as k and $\neg k$ respectively. E.g. *holding*, height(0)=Y, height(end)=0. Note that $height(0) = Y$ is a schema for $height(0) = y$.

The atom $p = v$ where v denotes the value of process p will be used to refer to either a *c-atom* or an *f-atom*. An atom u or its negation $\neg u$ are referred to as *literals*. Negation of $=$ will be often written as $\neq$. E.g. $\neg holding$, $height(0) \neq 20$.

Definition 2.1 An *action description* of H is a collection of statements of the form:

$$l_0 \text{ if } l_1, \ldots, l_n. \tag{2.1}$$

$$a_e \text{ causes } l_0 \text{ if } l_1, \ldots, l_n. \tag{2.2}$$

$$\text{impossible } a \text{ if } l_1, \ldots, l_n. \tag{2.3}$$

where a_e and a are elementary and arbitrary actions respectively and l's are literals of $H(\Sigma, \mathcal{G}, \Delta)$. The l_0's are called the *heads* of the statements (2.1) and (2.2). The set $\{l_1, \ldots, l_n\}$ of literals is referred to as the *body* of the statements (2.1), (2.2) and, (2.3). Please note that literals constructed from *f-atoms* of the form $end = y$ will not be allowed in the heads of statements of H.

A statement of the form (2.1) is called a *state constraint.* It guarantees that any state satifying $l_1, \ldots, l_n$ also satisfies l_0. A *dynamic causal law* (2.2) says if an action a were executed in a state s_0 satisfying literals $l_1, \ldots, l_n$ then any successor state s_1 would satisfy l_0. An *executability condition* (2.3) states that action a cannot be executed in a state satisfying $l_1, \ldots, l_n$. If $n = 0$ then if is dropped from (2.1), (2.2), (2.3).

Example 2.1 Let us now construct an action description AD_0 describing the transition diagram from Figure 1.2. Let $\mathcal{G}_0$ contain functions

$$f_0(Y, T) = Y.$$

$$f_1(Y, T) = Y - \frac{1}{2}gT^2.$$

where $Y \in range(height)$, g is acceleration due to gravity, and T is a variable for *time* points.

The description is given in language H whose signature Σ_0 contains actions *drop* and *catch*, a continuous process *height*, and fluents *holding*, $height(0)$ and $height(end)$. *holding* is a boolean fluent; $range(height)$ is the set of non-negative real numbers. It is easy to see that statements (1.3) and (1.6) are dynamic causal laws while statements (1.4), (1.5) and (1.7) are executability conditions and statements (1.8) and (1.9) are state constraints.

Example 2.2 This example is simplied version of the example used by Reiter in [14]. Consider an elastic ball moving with uniform velocity on a frictionless floor between two walls, w_1 and w_2. Assume that the floor is the X-axis and the walls are parallel to the Y-axis. We expect the ball to *bounce* indefinitely between the two walls. The bouncing causes *velocity* of the ball to change discontinuously. And as long as the *velocity* is not zero, *position* changes continuously with time.

Let us now construct an action description AD_1 of $H(\Sigma_1, \mathcal{G}_1, \Delta)$ that will enable us to define the *velocity* and *position* of the ball. Signature Σ_1 contains the action $bounce(W)$ which denotes the ball bouncing against wall W, a continuous process *position*, and fluents *velocity*, $position(0)$ and $position(end)$.

Since *velocity* is uniform and is a changed only by *bounce* we treat it as a fluent. The $range(velocity)$ is the set of real numbers and the $range(position)$ is the set of non-negative real numbers. Let $\mathcal{G}_1$ contain the function

$$f_2(Y_0, V, T) = \begin{cases} Y_0 & \text{if } \ T = 0. \\ f_2(Y_0, V, T-1) + V & \text{if } \ T > 0. \end{cases}$$

where $Y_0 \in range(position)$ and $V \in range(velocity)$. On hitting a wall, the ball changes direction. This is defined by the causal law:

$$bounce(W) \ \ causes \ \ velocity = -V \ \ if \ \ velocity = V. \tag{2.4}$$

Statement (2.4) says that if the ball moving with velocity V bounces against the wall W at time *end* in a state, s, then its new velocity is $-V$ in any successor state of s. *position* will be defined by the state constraint.

$$\begin{aligned} position = f_2(Y_0, V, T) \ \ if \ \ & position(0) = Y_0, \\ & velocity = V. \end{aligned} \tag{2.5}$$

Statement (2.5) says that *position* is defined by Newtonian equations in any state. The occurrence times of the *bounce* action is determined by Newtonian equations. One way to represent such an action is to write statements called *action triggers* that include these Newtonian equations. In general, action triggers describe the effects of processes or actions on other actions. We will not address the issue of how to write triggers in this thesis because it is not the purpose of this thesis. Our future work may involve extending language H to include triggers.

2.2 Semantics

The semantics of *process description language*, H, is similar to the semantics of action language B given by McCain and Turner [10; 11]. An action description AD of H, describes a transition diagram, *TD(AD)*, whose nodes represent possible states of the world and whose arcs are labeled by actions. Whenever possible the parameter AD will be omitted.

Definition 2.2 An *interpretation*, I, of H is a mapping that assigns (properly typed) values to the processes of H such that for every continuous process, c, $I(c(end)) = I(c)(I(end))$ and $I(c(0)) = I(c)(0)$.

A mapping I_0 below is an example of an interpretation of action language of Example 2.1.

$$I_0(end) = 0,$$

$$I_0(holding) = \top,$$

$$I_0(height(0)) = 50,$$

$$I_0(height(end)) = 50,$$

$$I_0(height) = f_0(50, T).$$

Definition 2.3 An atom $p = v$ is *true in interpretation* I (symbolically $I \models p = v$) if $I(p) = v$. Similarly, $I \models p \neq v$ if $I(p) \neq v$.

An interpretation I is closed under the state constraints of AD if for any state constraint (2.1) of AD, $I \models l_i$ for every i, $1 \leq i \leq n$ then $I \models l_0$.

Definition 2.4 A *state*, s, of a *TD(AD)* is an interpretation closed under the state constraints of AD.

It is easy to see that interpretation I_0 corresponds to the state s_0 in Figure 1.2. Whenever convenient, a state, s, will be represented by a set $\{l : s \models l\}$ of literals. For example, in Figure 1.2, the state s_0 will be the set

$$s_0 = \{end = 0,\ holding,\ height(0) = 50,\ height(end) = 50,\ height = f_0(50, T)\}$$

Please note that only atoms are shown here. s_0 also contains the literals $holding \neq \bot$, $height(0) \neq 10$, $height(0) \neq 20$ and so on.

Definition 2.5 Action a is *executable* in a state, s, if for every non-empty subset a' of a, there is no executability condition

$$\text{impossible}\ \ a'\ \ \text{if}\ \ l_1, \ldots, l_n.$$

of AD such that $s \models l_i$ for every i, $1 \leq i \leq n$.

Let a_e be an elementary action that is executable in a state s. $E_s(a_e)$ denotes the set of all direct effects of a_e, i.e. the set of all literals l_0 for which there is a dynamic causal law

$$a_e \ causes \ l_0 \ if \ l_1, \ldots, l_n$$

in AD such that $s \models l_i$ for every i, $1 \leq i \leq n$. If a is a compound action then $E_s(a) = \bigcup_{a_e \in a} E_s(a_e)$.

A set L of literals of H is closed under a set Z of state constraints of AD if L includes the head, l_0, of every state constraint

$$l_0 \ \text{ if } \ l_1, \ldots, l_n$$

of AD such that $\{l_1, \ldots, l_n\} \subseteq L$. The set $Cn_Z(L)$ of consequences of L under Z is the smallest set of literals that contains L and is closed under Z.

A transition diagram TD=$\langle \Phi, \Psi \rangle$ where

1. Φ is a set of states.

2. Ψ is a set of all triples $\langle s, a, s' \rangle$ such that a is executable in s and s' is a state which satisfies the condition

$$s' = Cn_Z(\ E_s(a) \ \cup \ (s \cap s' \ ) \ ) \tag{2.6}$$

where Z is the set of state constraints of AD. The argument to $Cn(Z)$ in (2.6) is the union of the set $E_s(a)$ of the "direct effects" of a with the set $s \cap s'$ of facts that are "preserved by inertia". The application of $Cn(Z)$ adds the "indirect effects" to this union. In Example 2.1, the set $E_{s_0}(drop)$ of direct effects of *drop* will be defined as

$$E_{s_0}(drop) = \{\neg holding\}.$$

The instantaneous action *drop* occurs at global time **0** and has no direct effect on the value of *height* at **0**. This means that the value of *height* at the *end* of s_0 will be preserved at time 0 of s_1. Therefore,

$$s_0 \cap s_1 = \{height(0) = 50\}.$$

The application of $Cn(Z)$ to $E_{s_0}(drop) \cup (s_0 \cap s_1)$ gives the set

$$Q = \{\neg holding, height(0) = 50, height = f_1(50, T)\}$$

where Z contains the state constraints (1.8) and (1.9).

The set Q will not represent the state s_1 unless *end* is defined. Let us suppose that $s_1(end) = 2$. Therefore, we get

$$s_1 = \{end = 2, \neg holding, height(0) = 50, height(end) = 30, height = f_1(50, T)\}.$$

Please note that only atoms are shown here.

2.3 Specifying history

In addition to the action description, the agent's knowledge base may contain the domain's *recorded history* - observations made by the agent together with a record of its own actions.

The recorded history defines a collection of paths in the diagram which, from the standpoint of the agent, can be interpreted as the system's possible pasts. If the agent's knowledge is complete (e.g., it has complete information about the initial state and the occurrences of actions, and the system's actions are deterministic) then there is only one such path.

The *Recorded history*, Γ_n, of a system up to a current moment n is a collection of *observations*, that is statements of the form:

$$obs(v, p, t, i).$$

$$hpd(a, t, i).$$

where i is an integer from the interval $[0, n)$ and time point, $t \in \Delta$. i is an index of the trajectory. For example, $i = 5$ denotes the step 5 of the trajectory reached after performing a sequence of 5 actions.

$obs(v, p, t, i)$ means that process p was observed to have value v at time t of step i. $hpd(a, t, i)$ means that action a was observed to have happened at time t of step i. Observations of the form $obs(y, p, 0, 0)$ will refer to the initial state of the system.

Definition 2.6 A pair $\langle AD, \Gamma \rangle$ where AD is an action decription of H and Γ is a set of observations, is called a *domain description*.

Definition 2.7 Given an action decription AD of H that describes a transition diagram TD(AD), and recorded history, Γ_n, upto moment n, a path

$$\langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n \rangle$$

in the TD(AD) is a *model* of Γ_n with respect to the domain description, $\langle AD, \Gamma_n \rangle$, if for every i, $0 \leq i \leq n$ and $t \in \Delta$

1. $a_i = \{a : hpd(a, t, i) \in \Gamma_n\}$;
2. if $obs(v, p, t, i) \in \Gamma_n$ then $p = v \in s_i$.

CHAPTER 3

TRANSLATION INTO LOGIC PROGRAM

In this chapter we will discuss the translation of a domain description written in language H into rules of an *A-Prolog* program. *A-Prolog* is a language of logic programs under the answer set semantics [5] for representing agent's knowledge about the domain and formulating the agent's reasoning tasks. Since we use SMODELS [12] to compute answer sets of the resulting A-Prolog program, the translation will comply with the syntax of the SMODELS inference engine.

Given a domain description, $\mathcal{D} = \langle AD, \Gamma \rangle$ where AD is an action description of H and Γ is a set of observations, we will construct the logic program, $\alpha_0(\mathcal{D})$ by mapping statements of $\mathcal{D}$ into rules of A-Prolog.

$\alpha_0(\mathcal{D})$ contains two parts. The first part contains declarations for actions and processes and the second part contains translations for the statements of H and the observations in Γ.

3.1 Declarations

Let us look at a general way of declaring actions and processes:

$$action(action\_name, action\_type).$$

$$process(process\_name, process\_type).$$

action_name and *action_type* are non-numeric constants denoting the name of an action and its type respectively. Similarly, *process_name* and *process_type* are non-numeric constants denoting the name of a process and its type respectively. For instance in Example 2.1 the actions and processes are declared as follows:

$$action(drop, agent).$$
$$action(catch, agent).$$

$$process(height, continuous).$$
$$process(holding, fluent).$$

Now let us see how the range of a process is declared. There are a couple of ways of doing this. The range of *height* from Example 2.1 is the set of non-negative real numbers. In terms of logic programming this means infinite groundings. Therefore, we made a compromise and chose integers ranging from 0 to 50.

$$values(0..50).$$
$$range(height, Y) :- values(Y).$$

holding is a boolean fluent. Therefore, we write

$$range(holding, true).$$
$$range(holding, false).$$

Suppose we have a switch that can be set in three different positions, the range of the process *switch_position* is declared as:

$$range(switch\_position, low).$$

$$range(switch\_position, medium).$$

$$range(switch\_position, high).$$

In order to talk about the values of processes and occurrences of actions we have to consider the *time* and *step* parameters.

Integers from some interval $[0, n]$ will be used to denote the *step* of a trajectory. I's will be used as variables for *step*. Now every *step* has a duration associated with it. Therefore, integers from some interval $[0, m]$ will be used to denote the *time* points of every *step*. In this case, m will be the maximum allowed duration for any *step*. T's will be used as variables for *time*. Therefore, we write

$$step(0..n).$$

$$time(0..m).$$

Assume that n and m are sufficiently large for our applications. Then we add the rules

$$\#domain\ step(I; I1).$$

$$\#domain\ time(T; T1; T2).$$

for declaring the variables I, $I1$, T, $T1$ and, $T2$ in the language of SMODELS. The first domain declaration asserts that the variables I and $I1$ should get their domain from the literal $step(I)$.

3.2 General translations

Let us look at a general translation of an action description of H into rules of A-prolog. If a is an elementary action occurring in a statement that is being translated, it is translated as

$$o(a, T, I)$$

which is read as "*action a occurs at time T of step I.*" If a is a compound action then for each elementary action $a_e \in a$, we write $o(a_e, T, I)$.

If l is a literal of H occurring in the any part of the statement that is being translated, other than the head of a dynamic causal law then it will be written as

$$\alpha_0(l, T, I).$$

$\alpha_0(l, T, I)$ is a function that denotes a translation of literal l. A literal, l, occurring in the head of a dynamic causal law will be written as

$$\alpha_0(l, 0, I + 1).$$

In this thesis we limit ourselves to translating action descriptions of H in which the heads of dynamic causal laws are either *f-atoms* or their negations. The general translations look as follows:

Statement (2.1) will be translated as

$$\begin{aligned} \alpha_0(l_0,T,I) :-\ & \alpha_0(l_1,T,I), \\ & \ldots\ldots\ldots, \\ & \alpha_0(l_n,T,I). \end{aligned} \tag{3.1}$$

Statement (2.2) will be translated as

$$\begin{aligned} \alpha_0(l_0,0,I+1) :-\ & o(a_e,T,I), \\ & \alpha_0(l_1,T,I), \\ & \ldots\ldots\ldots, \\ & \alpha_0(l_n,T,I). \end{aligned} \tag{3.2}$$

Statement (2.3) will be translated as

$$\begin{aligned} :-\ & o(a,T,I), \\ & \alpha_0(l_1,T,I), \\ & \ldots\ldots\ldots, \\ & \alpha_0(l_n,T,I). \end{aligned} \tag{3.3}$$

In statement (2.3) if a is the non-empty compound action $\{a_1,\ldots,a_m\}$ then $o(a,T,I)$ in rule (3.3) will be replaced by $o(a_1,T,I),\ldots,o(a_m,T,I)$. We will not translate (2.3) when a is empty.

$\alpha_0(l,T,I)$ will be replaced by

- $val(V,c,0,I)$ if l is an atom of the form $c(0)=v$.

$val(V, c, 0, I)$ is read as "*V is the value of process c at time 0 of step I.*"

E.g. $height(0) = Y$ will be translated as $val(Y, height, 0, I)$.

- $-val(V, c, 0, I)$ if l is of the form $c(0) \neq v$.

  $-val(V, c, 0, I)$ is read as "*V is not the value of process c at time 0 of step I.*"

- $val(V, p, T, I)$ if l is an atom of the form $p = v$ other than $c(0) = v$.

  $val(V, p, T, I)$ is read as "*V is the value of process p at time T of step I.*"

  E.g. $height(end) = 0$ will be translated as $val(0, height, T, S)$.

- $-val(V, p, T, I)$ if l is of the form $p \neq v$ other than $c(0) \neq v$.

  $-val(V, p, T, I)$ is read as "*V is not the value of process p at time T of step I.*"

$\alpha_0(l, 0, I+1)$ will be replaced by

- $val(V, p, 0, I+1)$ if l is of the form $p = v$.

- $-val(V, p, 0, I+1)$ if l is of the form $p \neq v$.

Note that when translating the *f-atom*, $end = y$ we will not follow the above conventions. Instead we translate it as $end(T, I)$ where T denotes the *end* of step I. Observations of the form $obs(v, p, t, i)$ and $hpd(a, t, i)$ are translated as facts of A-Prolog programs. Before we look at some examples we will discuss domain independent axioms.

3.3 Domain independent axioms

Domain independent axioms define properties that are common to every domain. We will denote such a collection of axioms by Π. Given a domain description $\mathcal{D}$, and $\alpha_0(\mathcal{D})$ that maps $\mathcal{D}$ into rules of A-prolog, we will construct $\alpha(\mathcal{D})$ that adds Π to $\alpha_0(\mathcal{D})$. Therefore,

$$\alpha(\mathcal{D}) = \Pi \cup \alpha_0(\mathcal{D}).$$

Let us look at the axioms constituting Π.

End of state axioms

These axioms will define the *end* of every state s. The end of a state is the local time at which an action terminates s. When it comes to implementation we talk about the *end* of a *step* instead of state. Therefore, we write

$$end(T, I) :- o(A, T, I). \tag{3.4}$$

If no action occurs during a *step* then *end* will be the maximum time point allowed for that *step*. This is accomplished by using the choice rule

$$\{end(m, I)\}1. \tag{3.5}$$

The consequence of the rule (3.5) is that the number of end(m,I) that will be true is either 0 or 1. A *step* cannot have more than one *end*. This is expressed by (3.6).

$$\begin{aligned} :-\ & end(T1, I), \\ & end(T2, I), \\ & neq(T1, T2). \end{aligned} \tag{3.6}$$

Every *step* must end. Therefore, we write

$$ends(I) \; :- \; end(T, I). \tag{3.7}$$

$$:- \; not \;\; ends(I). \tag{3.8}$$

Every *step*, i, is associated with an interval $[0, e]$ where 0 denotes the start time and e denotes the end time of i. We will use the relation *out* to define the time points, $t \notin [0, e]$ and *in* to define the time points, $t \in [0, e]$.

$$\begin{aligned} out(T, I) :- \; & end(T1, I), \\ & T > T1. \end{aligned} \tag{3.9}$$

$$in(T, I) :- not \; out(T, I). \tag{3.10}$$

By using these relations in our rules we can avoid computing process values at time points, $t \notin [0, e]$.

Inertia axiom

The inertia axiom states that *things normally stay as they are.* It has the following form:

$$\begin{aligned} val(Y, P, 0, I+1) \; :- \; & val(Y, P, T, I), \\ & end(T, I), \\ & not \; -val(Y, P, 0, I+1). \end{aligned} \tag{3.11}$$

Intuitively, rule (3.11) says that actions are instantaneous. In example (2.1), *height* at global time **0** is 50 when the instantaneous action *drop* occurs at **0**.

Reality check axiom

This axiom guarantees that the agent's predictions match with his observations.

$$\begin{aligned} :-\ & obs(Y,P,T,I), \\ & \neg val(Y,P,T,I). \end{aligned} \tag{3.12}$$

Other axioms

The axiom

$$o(A,T,I) : -hpd(A,T,I). \tag{3.13}$$

says that if action A was observed to have happened at time T of step I then it must have occurred at time T of step I. And we have

$$val(Y,P,0,0)\ \ :-\ obs(Y,P,0,0). \tag{3.14}$$

for defining the initial values of processes. A fluent remains constant throughout the duration of a *step*. This is expressed by the axiom (3.15).

$$\begin{aligned} val(Y,P,T,I) :-\ & val(Y,P,0,I), \\ & process(P,fluent), \\ & in(T,I). \end{aligned} \tag{3.15}$$

Axiom (3.16) says that no process can have more than one value at the same time.

$$\begin{aligned} -val(Y1,P,T,I)\ \ :-\ & val(Y2,P,T,I), \\ & neq(Y1,Y2). \end{aligned} \tag{3.16}$$

3.4 Example translations

Now let us refer back to Examples 2.1 and 2.2 and see how the corresponding causal laws will be translated. In Example 2.1 the dynamic causal law

$$drop \;\; causes \;\; \neg holding.$$

is translated as

$$val(false, holding, 0, I+1) : -o(drop, T, I).$$

$$catch \;\; causes \;\; holding.$$

is translated as

$$val(true, holding, 0, I+1) : -o(catch, T, I).$$

Next we look at the executability conditions.

$$impossible \;\; drop \;\; if \;\; \neg holding.$$

is translated as

$$\begin{aligned} :- \;\; & o(drop, T, I), \\ & val(false, holding, T, I). \end{aligned}$$

$$impossible \;\; drop \;\; if \;\; height(end) = 0$$

is translated as

$$\begin{aligned} :-\ & o(drop, T, I),\\ & \quad val(0, height, T, I). \end{aligned}$$

$$impossible\ \ catch\ \ if\ \ holding$$

is translated as

$$\begin{aligned} :-\ & o(catch, T, I),\\ & \quad val(true, holding, T, I). \end{aligned}$$

Next we look at state constraints.

$$\begin{aligned} height = f_0(Y, T)\ \ if\ \ & height(0) = Y,\\ & holding. \end{aligned}$$

is translated as

$$function\ f0.$$

$$\begin{aligned} val(f0(Y, T), height, T, I) :-\ & val(Y, height, 0, I),\\ & val(true, holding, T, I),\\ & in(T, I),\\ & range(height, Y). \end{aligned}$$

The function $f0$ is a user defined function that is linked to lparse. Such functions are meant to be called directly from logic programs. Note that the function has to be declared before it appears in any rule. For more information on how to use them refer to the lparse user manual.

$$\begin{aligned} height = f_1(Y,T) \;\; if \;\; & height(0) = Y, \\ & \neg holding. \end{aligned}$$

is translated as

$$function \;\; f1.$$

$$\begin{aligned} val(f1(Y,T), height, T, I) :- \;\; & val(Y, height, 0, I), \\ & val(false, holding, T, I), \\ & in(T, I), \\ & range(height, Y). \end{aligned}$$

where $f1$ is also a user defined function that is linked to lparse. The value returned by the function $f1$ given Y and T, will determine the value of *height* at T.

Now let us look at the translations for the causal laws in Example 2.2. The dynamic causal law

$$bounce(W) \;\; causes \;\; velocity = V \;\; if \;\; velocity = -V.$$

is translated as

$$
\begin{aligned}
val(V1, velocity, 0, I+1) :- \; & o(bounce(W), T, I), \\
& val(V, velocity, T, I), \\
& V1 = -1 * V, \\
& wall(W), \\
& range(velocity, V), \\
& range(velocity, V1).
\end{aligned}
$$

And the state constraint

$$
\begin{aligned}
position = f_2(Y_0, V, T) \;\; if \;\; & position(0) = Y_0, \\
& velocity = V.
\end{aligned}
$$

is translated as

$$
\begin{aligned}
val(f2(Y0, V, T), position, T, I) :- \; & val(Y0, position, 0, I), \\
& val(V, velocity, T, I), \\
& in(T, I), \\
& range(position, Y0), \\
& range(velocity, V).
\end{aligned}
$$

where $f2$ is a user defined function. The following hypothesis establishes the relationship between the theory of actions in H and logic programming.

Hypothesis

Given a domain description $\mathcal{D} = \langle AD, \Gamma_n \rangle$ where AD is an action description of $H(\Sigma, \mathcal{G}, \Delta)$ and Γ_n is the recorded history upto moment n; if the initial situation of Γ_n is *complete*, i.e. for any process p of Σ, Γ_n contains $obs(v, p, 0, 0)$ then M is a model of Γ_n iff M is defined by some answer set of $\alpha(\mathcal{D})$.

A proof of the above hypothesis will not be presented in this thesis. If proven it means that our translations are indeed correct. In the next chapter we will look at a complex example and some experimental results.

CHAPTER 4

EXAMPLE DOMAIN

In this chapter we will look at an example domain and show how our language can be used to model it. First we will understand the physics of the system and then construct an action description describing the system. Later we will translate the statements of the action description into rules of A-prolog. Finally, we will look at some sample scenarios and experimental results.

Example 4.1 Consider a rectangular water tank with a faucet on the top and a drain at the bottom. The faucet is the source of water to the tank and the drain is an outlet. The faucet can be opened and closed. We are interested in predicting the volume of water in the tank. Let us first understand the physics of the system.

Assume that the velocity at which the water flows out of the faucet into the tank (called inflow rate) is approximately 3 m/sec when the faucet is open and 0 when it is closed. The volume of water flowing into the tank per second, denoted by V_{in}, is determined by the following equation:

$$V_{in} = inflow\_rate * cf. \tag{4.1}$$

where cf is the cross section area of the faucet opening. Now we will define the outflow rate which is the velocity at which the water flows out of the drain. We apply Bernoulli's equation of law of conservation of energy to an open tank under atmospheric pressure to derive

$$outflow\_rate = \sqrt{2 * g * h}. \tag{4.2}$$

where g is acceleration due to gravity and h is the height of the water level in the tank. Now we can define the volume of water flowing out of the tank per second, denoted by V_{out}, as follows

$$V_{out} = outflow\_rate * cd. \tag{4.3}$$

where cd is the cross section area of the drain opening. If V_t is the volume of the tank at current time, t, then

$$V_{t+1} = V_t + V_{in} - V_{out}. \tag{4.4}$$

Now let us construct an action description AD_2 describing the above system. Signature Σ_2 contains the actions $turn(open)$ and $turn(close)$ for opening and closing the faucet, continuous processes $volume$ and $outflow\_rate$ and fluents $open$, $inflow\_rate$, $volume(0)$, $volume(end)$, $outflow\_rate(0)$ and $outflow\_rate(end)$. The $range(volume)$ and $range(outflow\_rate)$ is the set of non-negative real numbers; $open$ is a boolean fluent; range(inflow) contains 0 and 3. Let $\mathcal{G}_2$ contain functions

$$f_3(Y,N,T) = \begin{cases} Y & \text{if } \ T = 0. \\ f_3(Y,N,T-1) + N * cf - f_4(Y,N,T-1) * cd & \text{if } \ T > 0. \end{cases}$$

$$f_4(Y,N,T) = \sqrt{2 * g * f_3(Y,N,T)/(l * b)}.$$

where $Y \in range(volume)$, $N \in range(inflow\_rate)$ and the constants cf, cd, g, l and b denote the cross section area of the faucet, cross section area of the drain, acceleration due to gravity, length and breadth of the tank respectively. Now let us look us at the causal laws. The effects of the action $turn(open)$ will be given by the causal law

$$turn(open) \;\; causes \;\; open. \tag{4.5}$$

which says that opening the faucet at time *end* in a state s causes *open* to be true in any successor state of s. The executability condition will have the form

$$impossible \;\; turn(open) \;\; if \;\; open. \tag{4.6}$$

which says that it is not possible to open the faucet at time *end* in a state s if it is already *open*. The effects of the action $turn(close)$ are given by the causal law

$$turn(close) \;\; causes \;\; \neg open. \tag{4.7}$$

which says that closing the faucet at time *end* in a state s causes *open* to be false in any successor state of s. The executability condition will have the form

$$impossible \;\; turn(close) \;\; if \;\; \neg open. \tag{4.8}$$

which says that it is not possible to close the faucet at time *end* in a state s if it is already *closed*. The fluent $inflow\_rate$ is defined by the state constraints

$$inflow\_rate = 3 \;\; if \;\; open. \tag{4.9}$$

which says that in any state, s, $inflow\_rate$ is 3 when the faucet is open and

$$inflow\_rate = 0 \;\; if \;\; \neg open. \tag{4.10}$$

which says that in any state, s, $inflow\_rate$ is 0 when the faucet is closed. The process $volume$ is defined by the state constraint

$$\begin{aligned} volume = f_3(Y, N, T) \;\; if \;\; & volume(0) = Y, \\ & inflow\_rate = N. \end{aligned} \tag{4.11}$$

which says that in any state, s, $volume$ is defined by the function $f_3(Y, N, T)$ where Y is volume at time 0 and N is the $inflow\_rate$. The definition of f_3 is obtained by rewriting equation (4.4). The process $outflow\_rate$ is defined by the state constraint

$$\begin{aligned} outflow\_rate = f_4(Y, N, T) \;\; if \;\; & volume(0) = Y, \\ & inflow\_rate = N. \end{aligned} \tag{4.12}$$

which says that in any state, s, $outflow\_rate$ is defined by the function $f_4(Y, N, T)$ where Y is the $volume$ at time 0 and N is the $inflow\_rate$. The definition of f_4 is obtained by rewriting the equation (4.2). In equation (4.2), the height of water level, h, is obtained by dividing the volume of water in the tank by the length and breadth of the tank. For example, if the length and breadth of the tank are 3 and 4 meters respectively and the volume of water in the tank is 36 cubic meters, then the height of water level is 3 meters. Therefore, h is substituted by $f_3(Y, N, T)/(l * b)$ in the definition of f_4.

Let α be a mapping from action description AD_2 into rules of A-prolog. $\alpha(AD_2)$ contains the following rules:

Statement (4.5) is translated as

$$val(true, open, 0, I+1) :- \ o(turn(open), T, I). \tag{4.13}$$

Statement (4.6) is translated as

$$\begin{aligned} :- \ & o(turn(open), T, I), \\ & val(true, open, T, I). \end{aligned} \tag{4.14}$$

Statement (4.7) is translated as

$$val(false, open, 0, I+1) :- o(turn(close), T, I). \tag{4.15}$$

And (4.8) is translated as

$$\begin{aligned} :- \ & o(turn(close), T, I), \\ & val(false, open, T, I). \end{aligned} \tag{4.16}$$

Statement (4.9) is translated as

$$val(3, inflow\_rate, T, I) :- val(true, open, T, I). \tag{4.17}$$

And (4.10) is translated as

$$val(0, inflow\_rate, T, I) :- val(false, open, T, I). \tag{4.18}$$

Statement (4.11) contains a complex recursive function f_3 which in turn calls the function f_4. One way of implementing such functions is to link them to lparse. Lparse uses pointer arithmetic to deal with the arguments of the user defined functions. It is capable of handling simple recursion but fails to give expected results when functions interact recursively with each other. Therefore, we simplify these functions considerably, so that lparse can handle them. Therefore the translations of (4.11) and (4.12) contain modified versions of f_3 and f_4. We will call these modified versions as f_3' and f_4' respectively.

The following equation defines the relationship between f_3 and f_3'.

$$f_3'(Y, N, T) = f_3(Y, N, 1) \;\; if \;\; f_3(Y_0, N, T-1) = Y.$$

Therefore, (4.11) is translated as

$$function \;\; f3'.$$

$$\begin{aligned} val(f3'(Y0, N), volume, T+1, I) :- \; & val(Y0, volume, T, I), \\ & val(N, inflow\_rate, T, I), \\ & in(T+1, I), \\ & range(inflow\_rate, N), \\ & range(volume, Y0). \end{aligned} \tag{4.19}$$

The following equation defines the relationship between f_4 and f_4'.

$$f_4'(Y, T) = f_4(Y, N, 0) \;\; if \;\; f_3(Y_0, N, T) = Y.$$

Therefore, (4.12) is translated as

$$function \;\; f4'.$$

$$\begin{aligned} val(f4'(Y), outflow\_rate, T, I) :- \; & val(Y, volume, T, I), \\ & range(volume, Y). \end{aligned} \tag{4.20}$$

For a complete listing of the translations along with declarations and domain independent axioms please refer to Appendix A.

Sample scenario and results

Let Γ be the collection of observations:

$$obs(25, volume, 0, 0).$$

$$obs(false, open, 0, 0).$$

$$hpd(turn(open), 0, 0).$$

$$hpd(turn(close), 3, 1).$$

The program $\alpha(AD_2, \Gamma)$ is obtained by adding Γ to $\alpha(AD_2)$. In order to run this program the variables and processes are declared as usual. For instance, I and T take integer values from the intervals [0,2] and [0,6] respectively. $range(volume)$ is the set of integers from the interval [0,60].

The program was run on Sparc Ultra 10 running Solaris 8 using the 1.0.9 version of lparse and 2.26 version of SMODELS. The corresponding answer set returned from the program was as expected. The average run time was 7.2 seconds of which SMODELS took 3.3 seconds and lparse and SMODELS together took 7 seconds. SMODELS directives are used to get a better looking output. The resulting answer set is:

$$\% \;\; process\_name(Value, Local\_time, Step).$$

$outflow\_rate(6,0,0).$	$volume(25,0,0).$	$inflow\_rate(0,0,0).$
$outflow\_rate(6,0,1).$	$volume(25,0,1).$	$inflow\_rate(3,0,1).$
$outflow\_rate(6,1,1).$	$volume(28,1,1).$	$inflow\_rate(3,1,1).$
$outflow\_rate(7,2,1).$	$volume(31,2,1).$	$inflow\_rate(3,2,1).$
$outflow\_rate(7,3,1).$	$volume(33,3,1).$	$inflow\_rate(3,3,1).$
$outflow\_rate(7,0,2).$	$volume(33,0,2).$	$inflow\_rate(0,0,2).$
$outflow\_rate(6,1,2).$	$volume(26,1,2).$	$inflow\_rate(0,1,2).$
$outflow\_rate(5,2,2).$	$volume(20,2,2).$	$inflow\_rate(0,2,2).$
$outflow\_rate(4,3,2).$	$volume(15,3,2).$	$inflow\_rate(0,3,2).$
$outflow\_rate(4,4,2).$	$volume(11,4,2).$	$inflow\_rate(0,4,2).$
$outflow\_rate(3,5,2).$	$volume(7,5,2).$	$inflow\_rate(0,5,2).$
$outflow\_rate(2,6,2).$	$volume(4,6,2).$	$inflow\_rate(0,6,2).$

The following answer set was obtained when local time was mapped into global time. Additional rules were added to the translated program in order to do this mapping. We will not talk about these rules in this thesis.

$\% \; process\_name(Value, \; Global\_time).$

$outflow\_rate(6,0).$	$volume(25,0).$	$inflow\_rate(changed(0,3),0).$
$outflow\_rate(6,1).$	$volume(28,1).$	$inflow\_rate(3,1).$
$outflow\_rate(7,2).$	$volume(31,2).$	$inflow\_rate(3,2).$
$outflow\_rate(7,3).$	$volume(33,3).$	$inflow\_rate(changed(3,0),3).$

$outflow\_rate(6,4)$.	$volume(26,4)$.	$inflow\_rate(0,4)$.
$outflow\_rate(5,5)$.	$volume(20,5)$.	$inflow\_rate(0,5)$.
$outflow\_rate(4,6)$.	$volume(15,6)$.	$inflow\_rate(0,6)$.
$outflow\_rate(4,7)$.	$volume(11,7)$.	$inflow\_rate(0,7)$.
$outflow\_rate(3,8)$.	$volume(7,8)$.	$inflow\_rate(0,8)$.
$outflow\_rate(2,9)$.	$volume(4,9)$.	$inflow\_rate(0,9)$.

As we can see $inflow\_rate$ is not uniquely defined at global time **0** and **3**. It changes from 0 to 3 at global time **0** and from 3 to 0 at global time **3**.

CHAPTER 5

PLANNING AND DIAGNOSIS

In this chapter we will look at some examples on how to do planning and diagnosis in a hybrid domain. The existing theories for planning and diagnosis can be applied to hybrid domains.

5.1 Planning

The ability to plan is an important characteristic of an agent. A plan is a sequence of actions that satisfies the agent's goal. By goal we mean a finite set of literals of H the agent wants to make true.

A planning problem can be solved in different ways. Answer set programming techniques is one of them. In this approach, the answer sets of an A-prolog program encode possible plans. These plans are generated by so called 'planning modules'.

In our examples the planning module is often a simple choice rule. Answer set planning does not require any specialized planning algorithms. The reasoning mechanism used for planning is the same as the one used for deducing answer sets. Let us now look at an example that involves planning.

Example 5.1 Consider a car moving along X-axis with uniform velocity. The domain consists of two actions $start(V)$ and $stop$. $start(V)$ causes the car to move with a *velocity*, V and *stop* causes the *velocity* to be 0. *position* of the car changes continuously with time as long as *velocity* is not zero.

Let us construct an action description AD_3 describing the above domain. The corresponding signature Σ_3 contains the actions $start(V)$ and *stop*, continuous process *position*, and fluents *velocity*, $position(0)$, and $position(end)$. The $range(velocity)$ is the set of real numbers and the $range(position)$ is the set of non-negative real numbers. Let $\mathcal{G}_3$ contain the function

$$f_5(Y_0, V, T) = \begin{cases} Y_0 & \text{if } \ T = 0. \\ f_5(Y_0, V, T-1) + V & \text{if } \ T > 0. \end{cases}$$

where $Y_0 \in range(position)$, $V \in range(velocity)$ and T is a variable for *time*. The action description AD_3 contains the following statements. The effects of the action $start(V)$ where $V \in range(velocity)$ will given by the causal law:

$$start(V) \ \ causes \ \ velocity = V. \tag{5.1}$$

which says that performing the action $start(V)$ at time *end* in a state s causes *velocity* to be V in any successor state of s. The executability condition will have the form:

$$impossible \ \ start(V) \ \ if \ \ velocity \neq 0. \tag{5.2}$$

which says that in any state, s, it is not possible to perform $start(V)$ at time *end* if the *velocity* is not zero. In other words, the car cannot be started if it is already moving. The effects of the action *stop* will be given by the causal law:

$$stop \ \ causes \ \ velocity = 0. \tag{5.3}$$

which says that performing the action *stop* at time *end* in a state s causes the *velocity* of the car to be 0 in any successor state of s. The executability condition will have the form:

$$impossible \;\; stop \;\; if \;\; velocity = 0. \tag{5.4}$$

which says that in any state, s, it is not possible to *stop* the car at time *end* if the *velocity* is 0. In other words, the car cannot be stopped if it is not moving. *position* is defined by the state constraint

$$\begin{aligned} position \;\; = \;\; f_5(Y_0, V, T) \;\; if \;\; & position(0) = Y_0, \\ & velocity = V. \end{aligned} \tag{5.5}$$

which says that *position* is defined by Newtonian equations in any state.

We will now consider the program $\alpha(AD_3)$, obtained by translating the statements of AD_3. The actions and processes of the domain will be declared as

$$\begin{aligned} action(start(V), agent) :- \;\; & range(velocity, V), \\ & V > 0. \end{aligned}$$

$$action(stop, agent).$$

$$process(velocity, fluent).$$

$$process(position, continuous).$$

Statement (5.1) will be translated as

$$\begin{aligned} val(V, velocity, 0, I+1) :- \;\; & o(start(V), T, I), \\ & range(velocity, V). \end{aligned}$$

Statement (5.2) will be translated as

$$\begin{aligned} :-\ & o(start(V), T, I), \\ & val(V0, velocity, T, I), \\ & V0 \neq 0, \\ & range(velocity, V), \\ & range(velocity, V0). \end{aligned}$$

Statement (5.3) will be translated as

$$val(0, velocity, 0, I+1) : -o(stop, T, I).$$

Statement (5.4) will be translated as

$$\begin{aligned} :-\ & o(stop, T, I), \\ & val(0, velocity, T, I). \end{aligned}$$

Statement (5.5) will be translated as

$$function\ f5.$$

$$\begin{aligned} val(f5(Y0, V, T), position, T, I) :-\ & val(Y0, position, 0, I), \\ & val(V, velocity, T, I), \\ & in(T, I), \\ & range(position, Y0), \\ & range(velocity, V). \end{aligned}$$

Now consider the recorded history, Γ_1

$$obs(0, velocity, 0, 0).$$

$$obs(0, position, 0, 0).$$

These observations say that initially the car is not moving and positioned at 0. The program $\alpha(AD_3, \Gamma_1)$ is obtained by adding Γ_1 to $\alpha(AD_3)$.

Now suppose that the goal of an agent acting in this domain is to drive to a certain position on the X-axis and stop there. We will use the rule

$$\begin{aligned} goal(T, I) :- \ & val(8, position, T, I), \\ & val(0, velocity, T, I). \end{aligned} \tag{5.6}$$

to say that the goal is to reach position 8 and stop there. To achieve this goal, the values of I and T must satisfy the rule

$$success :- \ goal(T, I). \tag{5.7}$$

Failure is not an option. Therefore, we write

$$:- \ not\ success. \tag{5.8}$$

The rules (5.6), (5.7), and (5.8) will be added to the program $\alpha(AD_3, \Gamma_1)$. We know that I takes integer values from some interval $[0, n]$. This means that we can look for plans of no more than n consecutive steps. Candidate plans will be generated by the choice rule, PM_0:

$$\{o(A, T, I) : action(A, agent)\} :- I < n.$$

For any value of I ranging from 0 to $n-1$, if the goal is not satisfied, PM_0 will select a candidate action. PM_0 is also called the *planning module*.

Answer sets of the program $\alpha(AD_3, \Gamma_1) \cup PM_0$ will encode candidate plans, i.e. possible sequences of actions that satisfy *success*.

In our example, I and T take integer values from the intervals [0,2] and [0,6] respectively. A total of 21 different candidate plans were generated and the average run time was 4.6 seconds of which SMODELS took 2.75 secs and Lparse and SMODELS together took 4.4 seconds. One of the candidate plans

$$o(start(2), 4, 0).$$

$$o(stop, 4, 1).$$

is to start driving with a velocity of 2 at time point 4 of step 0 and then stop at time point 4 of step 1 at which point the car is positioned at 8.

5.2 Diagnosis

In this section we are interested in the agent's ability to diagnose. Diagnosis involves finding possible explanations for discrepancies between agent's predictions and system's actual behavior.

The algorithms used in [1; 3] to perform diagnostic tasks are based on encoding agent's knowledge in A-prolog and reducing the agent's task to computing answer sets of logic programs. We will use a similar approach to do diagnosis in hybrid domains.

The first step of an observe-think-act loop of an intelligent agent is to observe the world, explain observations, and update the knowledge base. We will try to provide the agent with the reasoning mechanism to explain observations.

We assume that the agent is capable of making correct observations, performing actions and remembering the domain history. We also assume that normally the agent is capable of observing all relevant exogenous actions occurring in its environment.

Now let us look at some terminology. Let D be a diagnostic domain.

Definition 5.1 By a diagnostic domain we mean the pair $\langle \Sigma, TD \rangle$ where Σ is a domain signature and TD is a transition diagram over Σ.

Consider an action description, AD of H that describes D. Let Γ_n be a recorded history of D up to moment n. Now, consider a mapping α, that maps the domain description $\mathcal{D} = \langle AD, \Gamma_n \rangle$ into rules of A-Prolog. The resulting logic program will be denoted by $\alpha(AD, \Gamma_n)$.

The recorded history, Γ_n contains a record of all actions, the agent performed or observed to have happened upto moment $n-1$. This knowledge enables the agent to make predictions about values of processes at the current moment, n.

To test these predictions, the agent observes the current value of some processes. We simply assume that there is a collection, P, of processes which are observable at n. The set

$$O^n = \{obs(v, p, t, n) : p \in P\}$$

contains the corresponding observations. At this point it is convenient to split the domain's history into two parts. The previously recorded history, Γ_n and the current observations O^n. A pair

$$\mathcal{C} = \langle \Gamma_n, O^n \rangle$$

will be often referred to as system's configuration. If new observations are consistent with the agent's view of the world, i.e. if $\mathcal{C}$ is consistent, then O^n simply becomes part of recorded history. In other words $\Gamma_{n+1} = \mathcal{C}$. Otherwise, the agent has to start seeking the explanation(s) of the mismatch.

Let

$$conf(\mathcal{C}) = \alpha(AD, \Gamma_n) \cup O^n.$$

$\mathcal{C}$ is consistent iff $conf(\mathcal{C})$ has an answer set.

Definition 5.2 The configuration $\mathcal{C}$ is a *symptom* of system's malfunctioning iff the program $conf(\mathcal{C})$ has no answer set.

An explanation of the symptom can be found by discovering some unobserved past occurrences of exogenous actions. Let $\mathcal{E}$ denote a set of elementary exogenous actions corresponding to D.

Definition 5.3 An *explanation*, E, of symptom $\mathcal{C}$ is a collection of statements

$$E = \{hpd(a, t, i) : 0 \leq i < n \text{ } and \text{ } a \in \mathcal{E}\}$$

such that $\Gamma_n \cup O^n \cup E$ is consistent.

Answer set programming provides a way of computing explanations for inconsistency of $\mathcal{C}$. Such a computation can be viewed as 'planning in the past'.

We construct a program called *diagnostic module*, DM, that computes explanations. The simplest diagnostic module, DM_0, is defined by the choice rule of SMODELS.

$$\{o(A,T,I) : action(A,ex)\} : -0 \leq I < n.$$

where *ex* stands for exogenous action.

Finding diagnoses of the symptom, $\mathcal{C}$, is reduced to finding answer sets of the program $conf(\mathcal{C}) \cup DM_0$. A set E of occurrences of exogenous actions is a possible explanation of inconsistency of $\mathcal{C}$ iff there is an answer set Q of $conf(\mathcal{C}) \cup DM_0$ such that

$$E = \{o(A,T,I) : action(A,ex) \wedge o(A,T,I) \in Q \wedge hpd(A,T,I) \notin Q\}.$$

Let us introduce some faults in Example 4.1 of the water tank and see how a diagnosis is done when the agent's predictions do not match with reality.

First we introduce the exogenous action 'break' which causes the faucet to malfunction. By malfunction, we mean that the faucet will be unable to output any water even when it is open. We will introduce the boolean fluent 'broken' to denote that the faucet is broken.

We extend the signature, Σ_2 in Example 4.1 to include the exogenous action *break* and fluent *broken.* A few additions and modifications will be made to the action description AD_2 of Example 4.1. The effects of the action *break* will be given by the causal law:

$$break \ \ causes \ \ broken. \tag{5.9}$$

which says that the exogenous action *break* causes the faucet to be *broken*. The executability condition is of the form

$$impossible \ \ break \ \ if \ \ broken. \tag{5.10}$$

which says that the faucet cannot break if it is already broken.

Statements (5.9) and (5.10) are now part of AD_2. The statements (4.9) and (4.10) will be modified to accomodate the effects of *break*.

$$\begin{aligned} inflow\_rate = 3 \ \ if \ \ & open, \\ & \neg broken. \end{aligned} \tag{5.11}$$

$$\begin{aligned} inflow\_rate = 0 \ \ if \ \ & open, \\ & broken. \end{aligned} \tag{5.12}$$

Let us now look at the translations of the above statements. First we must declare the exogenous action *break* and the fluent *broken*.

$$action(break, ex).$$

$$process(broken, fluent).$$

broken is a boolean fluent. Therefore, $range(broken)$ is defined as

$$range(broken, true).$$

$$range(broken, false).$$

Now, $\alpha(AD_2)$ from example (4.1) will contain the rules (4.13), (4.14), (4.15), (4.16), (4.18), (4.19), (4.20), and the following rules:

Statement (5.9) is translated as

$$val(true, broken, 0, I+1) : -o(break, T, I).$$

Statement (5.10) is translated as

$$\begin{aligned} :- \; & o(break, T, I), \\ & val(true, broken, T, I). \end{aligned}$$

Statement (5.11) is translated as

$$\begin{aligned} val(3, inflow\_rate, T, I) :- \; & val(true, open, T, I), \\ & val(false, broken, T, I). \end{aligned}$$

Statement (5.12) is translated as

$$\begin{aligned} val(0, inflow\_rate, T, I) :- \; & val(true, open, T, I), \\ & val(true, broken, T, I). \end{aligned}$$

Example 5.2 Consider the recorded history, Γ_1 containing the observations

$$hpd(turn(open), 0, 0).$$

$$obs(false, open, 0, 0).$$

$$obs(25, volume, 0, 0).$$

$$obs(0, inflow\_rate, 0, 0).$$

$$obs(false, broken, 0, 0).$$

Let $O^1 = \{obs(0, inflow\_rate, 0, 1)\}$. $obs(0, inflow\_rate, 0, 1)$ says that $inflow\_rate$ was observed to be 0 at time 0 of step 1. Obviously, $val(3, inflow\_rate, 0, 1)$ predicted by Γ_1 contradicts the observations.

This discrepancy can be explained by unobserved occurrence of action *break* at time 0 of step 0. We will now see how to compute the possible explanations.

The program $conf(\mathcal{C}) = \alpha(AD_2, \Gamma_1) \cup O^1$ is inconsistent. Therefore, we augment it with the diagnostic module, DM_0

$$\{o(A, T, I) : action(A, ex)\} : -0 \leq I < n.$$

The resulting program $conf(\mathcal{C}) \cup DM_0$ is consistent and has an answer set that contains $o(break, 0, 0)$ which is the possible explanation for the inconsistency of $conf(\mathcal{C})$.

The average run time for this program was 4.9 seconds of which SMODELS took 2.1 seconds and Lparse and SMODELS together took 4.7 seconds.

Example 5.3 Let us look at another scenario where answer set programming alone may not be enough to find explanations.

Consider the history, Γ_1

$$hpd(turn(close), 3, 0).$$

$$obs(true, open, 0, 0).$$

$$obs(25, volume, 0, 0).$$

$$obs(3, inflow\_rate, 0, 0).$$

$$obs(false, broken, 0, 0).$$

and the new observation, $O^1 = \{obs(17, volume, 0, 1)\}$. The program $conf(\mathcal{C}) = \alpha(AD_2, \Gamma_1) \cup O^1$ is found to be inconsistent. So it is augmented with the diagnostic module DM_0. The resulting program is still inconsistent.

The diagnostic module, DM_0 will compute only those explanations in which the unobserved exogenous actions occur in parallel with agent's action.

In Example 5.3, Γ_1 predicts $val(33, volume, 0, 1)$ which contradicts the observation $obs(17, volume, 0, 1)$. Therefore, we add DM_0 to find explanations. DM_0 will look for an exogenous action that occurs at the same time as the agent action $turn(close)$. But fails to find one.

In reality an unobserved exogenous event must have happened before the agent action was executed. A possible explanation for the unexpected observation $obs(17, volume, 0, 1)$ is the unobserved occurrence of *break* at time 1 of step 0.

Similarly, there may arise situations in which *break* could have happened at time point 0 or time point 2 of step 0. DM_0 will be unable to explain the unexpected observations in both situations.

Therefore, answer set programming alone is not enough to do diagnosis in such situations. We have some ideas on how to approach such problems. But before we talk about them, there are other issues such as grounding that must be addressed.

In most of our examples agent's knowledge is encoded in A-prolog and inference engines such as SMODELS are used to reason about such knowledge. Computing values of processes may involve trignometric functions, differential equations, complex formulas etc. Existing answer set solvers cannot carry out such computations. Also when numbers become large, the solvers run out of memory. Besides this, SMODELS grounds all variables in the program leading to poor efficiency.

In the following paragraphs, we propose an agent architecture for hybrid domains which will overcome computational problems and aid in efficient diagnosis.

A solution to the computation problem is to limit the answer set solver to reason about effects of actions and leave the computations to an external program. The external program will be called *monitor*. The agent and the monitor will interact with each other in the following manner.

The agent's task will be reduced to computing the answer sets of an A-prolog program. The answer sets will contain information about the values of processes at time 0 and the functions associated with these processes. The only change is that these functions are not evaluated yet.

The answer sets will be input to the monitor which then evaluates the functions asssociated with processes. Each time the agent performs an action the initial values and functions will be input to the monitor. The monitor will record all occurrences of agent actions.

The monitor will be capable of observing actions that occur naturally or triggered by other actions in the environment. It will report such observations to the agent. The agent will do the reasoning and send new input to the monitor.

An important function of the monitor is to help the agent diagnose. The computations done by the monitor will become the predictions of the monitor. The monitor will record observations with the help of sensors and then compare these observations with its predictions. If there is discrepancy it will inform the agent of all those observations that did not match, along with the time at which the discrepancy was found.

With this information the agent's task will become easier. Now the agent already knows when the exogenous action(s) occurred. It still needs to find out what exogenous action(s) took place . For this the agent will use answer set programming techniques. Once the correct explanation is found, the agent reasons about the effects of the exogenous action(s) and sends input to the monitor. The monitor will record the occurrence of exogenous action(s).

The Figure 5.1 summarizes the interaction between monitor and agent.

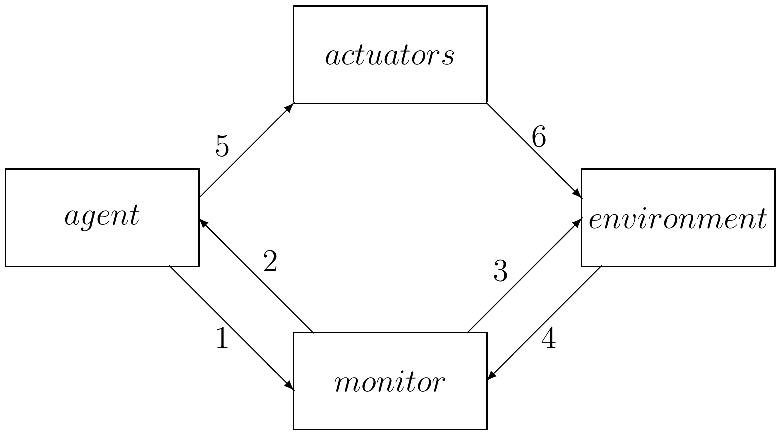


Figure 5.1: Architecture of an agent in a hybrid domain

The labeled arcs denote the following:

1 - The agent inputs initial values and functions associated with processes to the monitor.

2 - Monitor informs the agent about discrepancies, actions triggered in the environment, initial situation, etc.

3 - Monitor observes the environment.

4 - Monitor records observations.

5 - The agent sends messages to the actuators to perform actions.

6 - The actuators perform actions in the environment.

Finally we say that with the help of monitor we will be able to find an explanation for the inconsistency in Example 5.3, overcome computational problems, and improve efficiency considerably. There are still some pending issues such as grounding in SMODELS which can be overcome by delayed grounding.

Example 5.4 Now let us see an example in which the monitor detects inconsistency and reports it to the agent and the agent uses answer set programming techniques to find out an explanation for inconsistency. We will use the water tank example again. Consider the recorded history, Γ_1 consisting of

$$obs(3, inflow\_rate, 0, 0).$$

$$obs(25, volume, 0, 0).$$

$$obs(false, broken, 0, 0).$$

$$obs(true, open, 0, 0).$$

Suppose that the monitor observes that $inflow\_rate$ is 0 at time 3 of step 0. But the predicted value is 3. Since there is a discrepancy it sends a message to the agent that $inflow\_rate$ was observed to be 0 at time 3 of step 0. This will be represented by the fact:

$$error(0, inflow\_rate, 3, 0). \tag{5.13}$$

Now we write the general rule

$$obs(Y, P, 0, I + 1) :- \; error(Y, P, T, I). \tag{5.14}$$

which states that since an error was detected at time T of step I it must be true that an exogenous action must have occurred at this time, and therefore Y will be the observed value of process P at time 0 of the next step, I+1. We also need the rule

$$end(T, I) :- \; error(Y, P, T, I). \tag{5.15}$$

to make sure that the step ends at time T when the discrepancy was detected.

The maximum number of steps is incremented by one and the rules (5.13), (5.14), (5.15) are added to the program $\alpha(AD_2, \Gamma_1)$. The resulting program is inconsistent.

Therefore we augment it with the diagnostic module, DM_1 to restore consistency.

$$\begin{aligned} \{o(A,T,I) : action(A,ex)\} :- \ & error(Y,P,T,I), \\ & I < n. \end{aligned} \tag{5.16}$$

The answer set of the resulting program contains $o(break,3,0)$ which is indeed the correct explanation. The average run time for this program was 4.9 seconds of which SMODELS took 2.1 secs and Lparse and SMODELS together took 4.6 seconds.

CHAPTER 6

RELATED WORK

In this chapter we will compare language H with two other languages used for similar purposes. The first language is called *situation calculus* [13; 14] and the second one is called $\mathcal{ADC}$ [4] which stands for $\mathcal{A}$ctions with $\mathcal{D}$elayed and $\mathcal{C}$ontinuous effects.

6.1 Situation calculus

In this section we will compare language H with *situation calculus*. Situation calculus was introduced by John McCarthy in 1963 as a language for representing actions and their effects. But it was Reiter who enhanced situation calculus with features like *time*, concurrency, and natural actions to be able to model hybrid systems.

Situation calculus or sitcalc for short uses an approach based on first-order logic for modeling dynamical systems. The statements of the language are formulas of first-order logic. We on the other hand , use an action language/logic programming approach to modeling dynamical systems.

Situation calculus does not use transition function based semantics to characterize actions. By transition function based semantics we mean the approach in which the world is viewed as a dynamical system represented by a transition diagram whose nodes correspond to possible physical states of the world and whose arcs are labeled by actions.

Situation calculus uses the term *situation* to denote a possible world history. A *situation* is a finite sequence of actions. An initial situation denotes an empty sequence of actions.

In [14] Reiter points out the difference between the terms situation and state as - a state is a snapshot of the world while situation is a finite sequence of actions. A state is a collection of fluents that hold in a situation. Two states are the same if they assign the same truth values to all the fluents. Two situations are the same iff they result from the same sequence of actions applied to the initial situation. Two situations may be different yet assign the same truth values to the fluents. Situations do not repeat while states can repeat.

E.g. Consider the blocks world domain. Let $move(a, b),\ move(c, a)$ denote a situation, s, resulting from performing the action $move(a, b)$ followed by $move(c, a)$. A state, st, corresponding to the situation s will contain the fluents $on(a, b)$ and $on(c, a)$.

Let us talk about some situation calculus terminology. The symbol $do(a, s)$ denotes a successor situation to s, resulting from performing action a in situation s. Relations whose truth values vary from situation to situation are called *relational* fluents. For example, $on(x, y, s)$ denotes that block x is on y in situation s. Functions whose values vary from situation to situation are called *functional* fluents. For example, $height(s)$ denotes the height of an object in situation s. Since the language is based on first-order logic, the full set of quantifiers, connectives and logical symbols are used, making the language powerful and expressive.

But there are some limitations. Situation calculus does not encourage the use of state constraints because they are a source of deep theoretical and practical difficulties in modeling dynamical systems. Let us understand why.

We know that the statements of the language are formulas of first-order logic. A state constraint will be written as $A \supset B$ where A and B are first-order formulas. The contrapositive $\neg B \supset \neg A$ will also be true in this case.

For example, the state constraint

$$on(x, y, s) \wedge on(x, z, s) \supset y = z.$$

expresses a truth about the blocks world domain. But the contrapositive is not necessarily true.

$$y \neq z \supset \neg on(x, y, s) \vee \neg on(x, z, s).$$

This shows that using classical implication for knowledge representation does not give expected results. For this and other practical reasons situation calculus does not encourage the use of state constraints.

In situation calculus, the axioms for representing actions and their effects presuppose that actions are deterministic. Therefore, the action theories contain deterministic actions only. On the other hand, action theories of H contain both deterministic and non-deterministic actions.

Sitcalc requires that the initial situation be complete. We do not have such restrictions for language H. And the implementation in A-Prolog is capable of handling incomplete initial situations.

In [14] Reiter introduced the term *process* in order to overcome the problems associated with concurrent execution of actions with durations. Reiter conceives actions with duration as processes, represented by relational fluents, and introduces durationless actions that initiate and terminate these processes.

For example, instead of the action *walk(x,y)* we might have instantaneous actions *startWalk(x,y)* and *endWalk(x,y)* and the process of walking from x to y, represented by relational fluent *walking(x,y,s)*. *startWalk(x,y)* causes the fluent to become true, *endWalk(x,y)* causes it to become false. With this device of instantaneous *start* and *end* actions, arbitarily complex concurrency can be represented. For example,

$$\{startWalk(A,B), startChewGum\}, \{endChewGum, startSing\},$$

$$\{endWalk(A,B)\}$$

is the sequence of actions beginning with simultaneously starting to walk and starting to chew, followed by simultaneouly ending to chew and starting to sing, followed by ending to walk (at which time the singing process is still going on).

Since we assume that actions of H are instantaneous we adopt Reiter's approach to model actions with durations and delayed effects. The only difference is that the term *process* refers to both fluents and continuous processes.

Reiter adds explicit representation for *time* to situation calculus which allows to specify the exact times, or a range of times, at which actions in a world history must occur. The temporal argument is added to all instantaneous actions, denoting the actual *time* at which an action occurs. For example, bounce(ball,wall,7.3) is the instantaneous action of ball bouncing on wall at *time* 7.3. Here *time* refers to global time unlike our approach where local time is used.

New function symbols are introduced in the language to handle the temporal argument. A new function symbol *time : action* $\rightarrow$ *reals* is introduced. *time(a)* denotes the time of occurrence of action a. For example, *time(bounce(ball,wall,7.3))=7.3*.

Another function symbol *start : situation* $\rightarrow$ *reals* is introduced. *start(s)* denotes the start time of situation s. Therefore,

$$start(do(a, s)) = time(a).$$

The start time of the initial situation, s_0 is arbitrary and may or may not be defined depending on the application. In our approach the initial state starts at global time 0.

In sitcalc, every action takes *time* as one of its arguments and every fluent takes *situation* as one of its arguments. In our approach, the time and state parameters are not explicitly mentioned in the statements of H but it is implied that they are associated with every action and process.

The value of a fluent with the situation argument s is its value at the start time of s. For example, the functional fluent $height(s)$ denotes *height* at the start time of s and $height(do(a, s))$ denotes *height* at the start time of $do(a, s)$.

The language does not provide any features to compute the value of a fluent at a time other than the start time of a situation. If there is an action that occurs at every time point then it is possible to define fluent values at every time point.

In language H, we have a collection $\mathcal{G}$ of functions for defining continuous processes. For instance in example (2.1) from chapter 2 we have functions f_0 and f_1 for defining *height*.

Let us do an axiom by axiom comparision of situation calculus and H. For this we will model the Example 2.1 from chapter 2 in situation calculus. In that example we had actions *drop* and *catch* and processes *holding* and *height*. The continuous process *height* will be treated as a functional fluent. *holding* will be treated as a relational fluent.

To reason about the effects of actions situation calculus uses three kinds of axioms namely action precondition axioms, successor state axioms and unique names axiom for actions.

Action precondition axioms specify conditions that must be satisfied in order for an action to be executed in any situation. They can be considered as counterparts of executability conditions of language H. They are more powerful than executability conditions in the sense that they can be used to determine the occurrence times of natural actions. The following example demonstrates how to use action precondition axioms to predict the occurrence times of a natural action such as *bounce*.

Example 6.1

$$poss(bounce(t), s) \equiv isFalling(s) \wedge$$
$$\{height(s) + vel(s)[t - start(s)] -$$
$$\frac{1}{2}G[t - start(s)]^2 = 0\}.$$

Note that in situation calculus lower case letter are used to denote variables and upper case letters are used to denote constants. $height(s)$ and $vel(s)$ are the height and velocity of the ball at start of situation s. $poss(bounce(t), s)$ means that *bounce* is physically possible at time t during situation s.

The action precondition axioms for *drop* and *catch* from Example 2.2 will be

$$poss(drop(t), s) \equiv holding(s) \wedge height(s) \neq 0.$$

$$poss(catch(t), s) \equiv \neg holding(s).$$

Successor state axioms define the value of a relational fluent or functional fluent in the successor situation resulting from performing an action in the current situation. It has two parts. The first part defines what action causes the fluent to have a new value in the successor situation. The second part captures inertia. It says that the fluent will retain its value from the current situation if the action had no effect on the fluent. Let us define the fluents *holding* and *height* from Example 2.2 using successor state axioms of situation calculus.

$$\begin{aligned} holding(do(c, s)) \equiv \; & (\exists t)\ catch(t) \in c \vee \\ & holding(s) \wedge \neg(\exists t) drop(t) \in c. \end{aligned} \tag{6.1}$$

$$\neg holding(do(c,s)) \equiv (\exists t)\ drop(t) \in c \vee \tag{6.2}$$
$$\neg holding(s) \wedge \neg(\exists t) catch(t) \in c.$$

$$height(do(c,s)) = h \equiv holding(s) \wedge h = height(s) \vee \tag{6.3}$$
$$\neg holding(s) \wedge h = height(s) - \tfrac{1}{2}\ G\ time(c)^2.$$

The concurrent action c in the above statements is a collection of simple actions that occur at the same time. As you can see the successor state axiom for a fluent f provides the dual functionality of a dynamic causal law of H with f in the head and the inertia axiom for f in a single statement. Note that we use state constraints to define *height* in Example 2.1.

Consider the axiom (6.2). Suppose that in the initial situation S_0, *holding* is true. Now suppose that action *drop* occurs at global time **0** causing *holding* to be false in the resulting situation S_1. This means that *holding* is false at **0**. In other words, it is uniquely defined at **0**. This is possible because the start time of S_0 is not defined.

In our approach we use local time. Suppose that at local time 0 of the initial state , *holding* is true. Now suppose that the ball is dropped at this time. This causes *holding* to be false at local time 0 of the successor state. Since both these local times map to the same global time **0**, *holding* is not uniquely defined at **0**. Global time **0** is the point of transition for the value of *holding* from *true* to *false*.

Coming to implementation, axioms of situation calculus are translated into rules of Prolog. The resulting program can be verified for correctness by querying the prolog interpreter for fluents values and occurrence times of actions. The existing prolog systems support floating point numbers, thereby allowing fluent values and occurrence times of actions to be real numbers. Reiter's group uses the Eclipse Prolog system primarily because, as implied by its name - the ECLiPSe Constraint Logic Programming System - provides built in constraint solving libraries that are used for temporal reasoning.

Most of the existing prolog systems suffer from floundering, omit 'occurs check' and are not capable of generating multiple models. Eclipse Prolog overcomes some problems involving non-ground negative atoms but suffers from other drawbacks.

We translate statements of H into rules of A-Prolog. A-Prolog uses a reasoning algorithm that is completely different from the one Prolog uses and overcomes many of Prolog's shortcomings. A-Prolog programs can have multiple models which means that A-prolog can handle non-determinism. It is also capable of representing defaults.

The task of predicting the values of processes and the occurrence times of actions is reduced to computing answer sets of A-Prolog programs. We use existing answer set solvers like SMODELS to compute answer sets of our programs.

Reiter's main goal was to model natural actions- actions that occur in response to known laws of physics. For example, a ball *bouncing* on the floor after being dropped. The occurrence time of *bounce* is determined by Newtonian equations.

In order to represent such an action the laws of physics are embodied in action precondition axioms as in example 6.1.

In our language H there are no special axioms that will define when a natural action such as *bounce* will occur. In chapter 2 we mentioned that this task will be accomplished by writing *action triggers*. Let us look at an example where we write triggers.

Suppose that a book was dropped from a height h above the ground. The time at which it *hits the ground* is equal to $\sqrt{2 * h/g}$ where g is acceleration due to gravity. In order to determine the time of occurrence of *hit_ground* we need to know the initial value of *height* plus we need to know whether the book is falling or someone is holding it. For this we will use the fluent *holding*.

We will define the relation *act_poss* that defines the possible time, t, at which *hit_ground* can occur as follows:

$$function\ f.$$

$$\begin{aligned} act\_poss(hit\_ground, f(H), I) :- \ & val(H, height, 0, I), \\ & H > 0, \\ & val(false, holding, 0, I). \end{aligned}$$

f is a user defined function that is linked to lparse that returns the value $\sqrt{2 * H/g}$. The following rule says that the action *hit_ground* will occur at time t of step i if t

is the time at which it is supposed to occur and t is a time point during step i.

$$o(hit\_ground, T, I) :- \ act\_poss(hit\_ground, T, I),$$
$$in(T, I).$$

An other approach to modeling natural worlds is to get rid of natural actions and just have processes. The effects of actions will then be captured by process definitions. This may probably lead to complex process definitions but it gives us another approach to compare with.

Example 6.2 Let us revisit the Example 2.2 from chapter 2 . We will model this example again but with a different approach. We will get rid of the natural action *bounce*.

The action description AD_1 that we constructed previously will be modified. Let us call this modified version as $AD_1^{'}$. Please note that *velocity* will be treated as a continuous process now.

The corresponding signature $\Sigma_1^{'}$ contains the continuous processes *position* and *velocity* and fluents *position*(0), *position*(*end*), *velocity*(0), *velocity*(*end*).

$\mathcal{G}_1^{'}$ contains the functions

$$f_0(Y_0, V_0, T) = \begin{cases} Y_0 & \text{if } T = 0. \\ f_0(Y_0, V_0, T-1) + f_1(V_0, Y_0, T-1) & \text{if } T > 0. \end{cases}$$

$$f_1(V_0, Y_0, T) = \begin{cases} V_0 & \text{if } T = 0. \\ V_0 & \text{if } f_0(Y_0, V_0, T) \neq wp_1 \;\; \textit{and} \\ & f_0(Y_0, V_0, T) \neq wp_2 \;\; \textit{and} \\ & T > 0. \\ -V_0 & \text{otherwise.} \end{cases}$$

where $Y_0 \in \mathit{range}(\mathit{position})$, $V_0 \in \mathit{range}(\mathit{velocity})$ and wp_1, wp_2 are constants denoting the position of the walls w_1 and w_2 respectively. The process *position* will be defined by the state constraint

$$\mathit{position} = f_0(Y_0, V_0, T) \;\; \mathit{if} \;\; \mathit{position}(0) = Y_0, \\ \mathit{velocity}(0) = V_0.$$

which says that *position* will be defined by Newtonian equations in any state. *velocity* is defined by the state constraint

$$\mathit{velocity} = f_1(V_0, Y_0, T) \;\; \mathit{if} \;\; \mathit{velocity}(0) = V_0, \\ \mathit{position}(0) = Y_0.$$

which says that the direction of *velocity* will change only when the *position* of the ball is the same as the *position* of a wall. This is of course when *velocity* is not zero. The effects of the action *bounce* are captured by the function f_1.

When we look at Examples 6.2 and 2.2, and the way they are modeled in H there is not much difference. The advantage of Example 6.2 is that when we implement it using A-Prolog, the answer set solver need not compute occurrence times of *bounce*.

When we implement Example 2.2 we have to write extra rules that will compute the occurrence times of *bounce*. These computations cost us time. The approach suggested by Example 6.2 is just one way of improving efficiency.

Let us discuss some of the limitations of H. Language H is not capable of representing the effects of several concurrently executing actions on a process. We will use the features of SMODELS to reason about such effects. Let us look at the following example.

Example 6.3 Consider the actions *deposit* and *withdraw* that effect the *balance* of a bank account. Now suppose that multiple *deposits* and *withdrawals* were made from the same account at the same time. The resulting *balance* can be computed by adding all the deposits to the existing *balance* and subtracting all the withdrawals from it. We will now see how to do this using *weight* rules of SMODELS.

Let us construct an action description, AD_4 of H describing the above domain. The corresponding signature Σ_4 contains the actions $deposit(A, X)$ and $withdraw(A, X)$ which denote depositing X dollars into account A and withdrawing X dollars from account A respectively and the fluents $increase(A)$, $decrease(A)$ and $balance(A)$. $increase(A)$ and $decrease(A)$ are numerical fluents that denote the increase and decrease in account A respectively. The $range(balance(A))$, $range(increase(A))$ and $range(decrease(A))$ is the set of real numbers.

The effects of the action $deposit(A, X)$ will be given by the causal law:

$$deposit(A, X) \;\; causes \;\; increase(A) = X.$$

which says that depositing X dollars into account A at time *end* in a state s causes an *increase* of X dollars in account A in any successor state of s. It is translated as

$$\begin{aligned} val(X, increase(A), 0, I+1) :- \ & o(deposit(A, X), T, I), \\ & account(A), \\ & range(balance(A), X). \end{aligned}$$

The effects of the action *withdraw* will be given by the causal law:

$$withdraw(A, X) \ \ causes \ \ decrease(A) = X.$$

which is translated as

$$\begin{aligned} val(X, decrease(A), 0, I+1) :- \ & o(withdraw(A, X), T, I), \\ & account(A), \\ & range(balance(A), X). \end{aligned}$$

We cannot define *balance*(A) using our language but we can write the following rules in the language of SMODELS.

$$\#weight \ \ val(X, P, T, I) = X.$$

$$
\begin{gathered}
val(X, balance(A), 0, I+1) :-\ val(X_0, balance(A), 0, I), \\
X_1[val(X_3, increase(A), 0, I+1) : range(balance(A), X_3)]X_1, \\
X_2[val(X_4, decrease(A), 0, I+1) : range(balance(A), X_4)]X_2, \\
X = X_0 + X_1 - X_2, \\
range(balance(A), X_0), \\
range(balance(A), X_1), \\
range(balance(A), X_2), \\
range(balance(A), X), \\
account(A).
\end{gathered}
$$

The total *increase* and *decrease* in each account A, is computed by using the weight literals

$X_1[val(X_3, increase(A), 0, I+1) : range(balance(A), X_3)]X_1,$

$X_2[val(X_4, decrease(A), 0, I+1) : range(balance(A), X_4)]X_2.$

To be able to determine whether these weight literals are satisfied the weight declaration '$\#weight\ \ val(X, P, T, I) = X.$' is used to assign weights to each atom of the form $val(X, P, T, I)$. In this case the weight of the atom $val(X, P, T, I)$ is X.

For example, $X_1[val(X_3, increase(A), 0, I+1) : range(balance(A), X_3)]X_1$, is satisfied if the sum of weights of the satisfied literals of the form

$$val(X_3, increase(A), 0, I+1)$$

is equal to X_1 where $X_1 \in range(balance(A))$. Intuitively, it can be viewed as aggregating over elements of a set.

The new *balance* is obtained by adding the total *increase* to the existing *balance* and then subtracting the total *decrease* from it. The rule that we used to define $balance(A)$ is called a weight rule.

An other approach to implementing Example 6.3 is to use ASET-Prolog+ [16] which is an extension of A-Prolog by sets and aggregates. Functions such as *sum* of elements of a set and *cardinality* of a set are implemented in this language. The answer sets of ASET-Prolog+ programs are computed by an inference engine called the ASET-solver.

Like H, the language of situation calculus is not capable of representing the effects of several concurrently executing actions on a fluent. Instead sitcalc depends on the implementation to reason about such effects. Eclipse Prolog has built in aggregate functions such as $sum(L)$ which returns the sum of elements of list L. $sum(L)$ can be used to define $balance(A)$ from Example 6.3.

6.2 Language $\mathcal{ADC}$

The language $\mathcal{ADC}$ was introduced by Baral, Son, and Tuan in [4] as a language for specifying actions with durations, continuous effects, delayed effects, and dependency on non-sharable resources.

$\mathcal{ADC}$(Actions with Delayed and $\mathcal{C}$ontinuous effects) is the first language to use a transition function based approach to dealing with such actions. Language H also uses a transition function based approach for dealing with such actions.

In $\mathcal{ADC}$, actions can have delayed and continuous effects. This means that effects of actions might not happen immediately or can last over a period of time and therefore actions are associated with time intervals of the form $[t_1, t_2]$ where t_1, t_2 are real numbers such that $0 \leq t_1 \leq t_2$.

Example 6.4 The action of driving a car for 10 units of time with velocity v will be represented as $drive_{0,10}(v)$.

As mentioned earlier language H adopts Reiter's approach for dealing with actions with durations and delayed effects. For example, the action *drive* in example 6.4 will be represented by the fluent *driving* and instantaneous actions *start_drive*(v) and *end_drive*. *start_drive*(v) causes *driving* to be true and *end_drive* causes *driving* to be false.

Since the duration of *drive* is not captured in our representation, we have to write a trigger for *end_drive* which says that *end_drive* will occur after 10 units of time since *start_drive*(v) was executed. Similarly, sitcalc will need triggers to characterize actions with fixed durations.

When actions do not have fixed durations, $\mathcal{ADC}$ adopts Reiter's approach of using processes. The *start* and *end* actions are treated as instantaneous actions which initiate and terminate processes.

Let us look at how actions with delayed effects are represented in languages $\mathcal{ADC}$ and H.

Example 6.5 Consider a time bomb that explodes when the time left on its timer becomes zero. In language $\mathcal{ADC}$, we will represent the action of *setting* a *timer* to x units of time as *set_timer*(x). This action causes the bomb to explode which is represented by the fluent *explosion*. The following proposition of $\mathcal{ADC}$ represents this effect.

$$set\_timer(x)\ \ causes\ \ explosion\ \ from\ \ x\ to\ x.$$

In the above statement, the x in "*x to x*" denotes x time units relative to the time point when *set_timer* was executed. Therefore, the above causal law says that *set_timer*(x) causes *explosion* after exactly x time units since its execution. This action does not have continuous effects and the only effect is at the x^{th} time unit. This is suggested by the "to x" in the above statement. In $\mathcal{ADC}$, lower case letters are used to denote variables.

Let us construct an action description AD_6 of H describing this example. The corresponding signature Σ_6 contains the actions *set_timer*(X), *detonate*, continuous process $time\_left$ and fluents *explosion*, $time\_left(0)$ and $time\_left(end)$. *explosion* is a boolean fluent; range(time_left) is the set of non-negative real numbers. Let $\mathcal{G}_6$ contain the function

$$f_6(X,T) = \begin{cases} X & \text{if } T = 0. \\ f_6(X, T-1) - 1 & \text{if } T > 0. \end{cases}$$

where $X \in range(time\_left)$ and T is a variable for time. Setting the timer to X

time units initializes $time\_left$ to X. Therefore, we write

$$set\_timer(X) \;\; causes \;\; time\_left(0) = X.$$

$time\_left$ will be defined by the state constraints

$$time\_left = f_6(X, T) \;\; if \;\; time\_left(0) = X.$$

which says that $time\_left$ will decrement with every time unit. The action *detonate* will be triggered when $time\_left$ becomes 0 causing *explosion*. Therefore, we write

$$detonate \;\; causes \;\; explosion.$$

The trigger for *detonate* will not be part of AD_6.

When we compare both representations, the reader might observe that there are no time intervals associated with actions of H. Instead the time intervals are associated with the states of the transition diagram as in Figure 1.2.

$\mathcal{ADC}$ contains the following propositions to characterize the effects of actions:

$$executable \;\; a \; if \;\; c_1, \ldots, c_k. \tag{6.4}$$

$$a \;\; needs \;\; r_1, \ldots, r_m. \tag{6.5}$$

$$a \;\; causes \;\; f = valf(f, f_1, \ldots, f_n, t) \;\; from \;\; t_1 \;\; to \;\; t_2. \tag{6.6}$$

$$a \;\; contributes \;\; valf(f, f_1, \ldots, f_n, t) \;\; to \;\; f \; from \;\; t_1 \;\; to \;\; t_2. \tag{6.7}$$

$$a \;\; initiates \;\; p \;\; from \;\; t_s. \tag{6.8}$$

$$a \;\; terminates \;\; p \;\; at \;\; t_s. \tag{6.9}$$

$$p \;\; is\_associated\_with \;\; f = valf(f, f_1, \ldots, f_n, t). \tag{6.10}$$

$$p \;\; is\_associated\_with \;\; f \leftarrow valf(f, f_1, \ldots, f_n, t). \tag{6.11}$$

When we compare these propositions with the statements of H, we see that every proposition except for (6.7) has a counterpart in H. The executablity conditions of H are counterparts of the propositions (6.4) and (6.5). The dynamic causal laws of H, without the preconditions, are counterparts of the propositions (6.6), (6.8) and (6.9).

Propositions (6.10) and (6.11) contain functions for evaluating f. The state constraints of H with empty bodies can be viewed as counterparts of these propositions. For example, $height = f_0(50, T)$. The version of $\mathcal{ADC}$ presented in [4] does have conditional effects and state constraints.

The proposition (6.7) affects the value of f at t_1 by contributing an increase specified by $valf(f, f_1, \ldots, f_n, t)$ during the interval $[t_1, t_2]$. Let us look at an example from [4].

Example 6.6 Suppose that driving a car with velocity v for 10 units of time consumes $c(v)$ units of gasoline per unit time. This is expressed as:

$$drive_{0,10}(v) \;\; contributes \;\; -c(v) * t \;\; to \;\; gas\_in\_tank \; from \;\; 0 \;\; to \;\; 10.$$

And suppose that filling gas for 10 units of time will increase the amount of gasoline by $2 * t$. This is expressed as:

$$fill\_gas_{0,10} \;\; contributes \;\; 2*t \;\; to \;\; gas\_in\_tank \;\; from\, 0 \;\; to \;\; 10.$$

Let us assume that while driving we can refill and that initially the gas_in_tank was 20. Suppose that action $drive_{1,10}(3)$ starts its execution at time 0 and $fill\_gas_{0,10}$ starts it execution at time 0. Let $c(3) = 1.5$. The net value of gas_in_tank at time 1 will be $= 20 - 1.5 * 1 + 2 * 1$ which is 20.5. Similarly at time 2, $gas\_in\_tank = 21$ and so on.

As mentioned earlier, language H and *sitcalc* depend on the implementation to represent the effects of several concurrently executing actions. Note that properties of objects that change continuously with time are still referred to as fluents in the language of $\mathcal{ADC}$.

We will now discuss the semantics for action theories in $\mathcal{ADC}$. In the presence of time and delayed effects, the effects of an action might not happen immediately or might last over a time interval. Therefore, a state encodes not only the fluent values but also obligations due to delayed effects of recent actions and actions under execution.

A state is a pair $\langle I, Q\rangle$ where I is an interpretation and Q is a set of future effects. An interpretation I maps a fluent f to a value $v \in dom(f)$. Given that $a_1, \ldots, a_n$ are executable in a state s, a function $\Phi(s, \{(a_1, t_1), \ldots, (a_n, t_n)\}, t)$, where $t_1 \leq t_2 \ldots \leq t_n \leq t$, expresses the state of the world after t units of time from the time t_0 corresponding to s assuming that the actions $a_1, \ldots, a_n$ were started at relative (to

t_0) times $t_1, \ldots, t_n$ respectively. Note that t is a number ≥ 0. If the effects of the actions $a_1, \ldots, a_n$ terminate at a time, t' such that $t' < t$ then the values of fluents during $(t', t]$ will be defined by inertia.

In $\mathcal{ADC}$, all notions of time are relative. In sitcalc, time is the actual occurrence time of an action. In our approach, time is local. But it is possible to predict global values of processes by mapping local time into global time.

The version of $\mathcal{ADC}$ presented in [4] does not allow conditional effects and temporal preconditions. Examples such as 2.2 in which the effects of actions depend on the value of some fluents temporally and conditionally, cannot be modeled using this version of $\mathcal{ADC}$. It does not contain state constraints and therefore will be unable to model examples such as 4.1.

The authors of [4] are planning to enhance $\mathcal{ADC}$ by adding new features such as those for expressing temporal preconditions, conditional effects, or state constraints. They use a Java planner to implement their action theories. Results are satisfactory.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis we introduced the action language H for modeling hybrid domains. We looked at many examples and showed how they are modeled in this language. The syntax and semantics of the language were defined. We implemented action theories of H using A-Prolog. Some examples involved planning and diagnosis.

We were able to show that language H overcomes many of the limitations of *sitcalc* and $\mathcal{ADC}$. It provides almost all the functionalities of *sitcalc* and $\mathcal{ADC}$ and can model domains that cannot be modeled by the other two.

Using local time not only improves efficiency but also allows us to represent different kinds of time units. For instance, given a transition diagram, TD, the local time of a state, $s \in TD$, could represent time in the order of seconds, while the local time of an other state could represent minutes or hours.

For example, consider the action $set\_timer(X)$ that sets the timer on a microwave to X time units. The local time of a state resulting from performing this action represents either minutes or seconds, depending on whether X is minutes or seconds.

Now let us look at some future work prospects. One area for future research is delayed grounding. By delayed grounding we mean that functional values and times are grounded if and only when needed. This could reduce the program size and improve efficiency.

Another area for future research is developing mechanisms for efficient planning and diagnosis. In chapter 5, there were some scenarios where a diagnosis could not be found. We had some ideas on how to do a diagnosis in such circumstances. One of the them was to introduce an external program called *monitor* that would interact with the A-Prolog programs, do most of the computations, and help in diagnosis. We would like to pursue this idea in the future.

Hybrid systems often involve control. An agent acting in such a system should be reactive and deliberative. A reactive agent is one that acts with respect to its sensor data. A deliberative agent is one that acts with respect to its goals.

Example of a system that combines reactive and deliberative reasoning is the Mars rover. If the Mars rover finds a rock that is interesting, it would come up with a plan to reach the rock, and execute it. Hence it is deliberative in this case. On its way to the rock it may come across obstacles. The system converts its sensory data to motion vectors to avoid obstacles. Hence it is reactive in this case. Sometimes it replans depending on the sensor data in which case it combines reactive and deliberative reasoning.

In this thesis we have not addressed how to combine reactive and deliberative reasoning. We would like to address this issue in the future.

REFERENCES

1. 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. M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In *AAAI Spring 2003 Symposium*, 2003.

3. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In Minker, J,. ed., *Logic-Based AI*, Kluwer Academic publishers, (2000), 257-279.

4. 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. 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. M. Gelfond and V. Lifschitz. Action Languages. In *Electronic Transactions on Artificial Intelligence*, 3(6), 1998.

7. 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. V. Lifschitz, Two components of an action language, In *Annals of Mathematics and Artificial Intelligence*, Vol. 21, 1997, pp. 305-320.

9. V. Lifschitz. Action languages, Answer Sets and planning. In *The Logic Programming Paradigm:a 25 year perspective.*357-373, Springer Verlag, 1999.

10. Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. In *proceedings of IJCAI-95*, pages 1978-1984, 1995.

11. Norman McCain and Hudson Turner. Causal theories of action and change. In *proceedings of AAAI-97*, pages 460-465, 1997.

12. I.Niemela and P.Simons. Smodels - an implementation of the stable model and well founded semantics for normal logic programs. In *Proceedings of LP-NMR'97*, pages 420-429, 1997.

13. R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceed-*

ings of the Fifth International Conference (KR'96), pages 2-13, Cambridge, Massachusetts, U.S.A., November 1996.

14. R. Reiter. Time, concurrency and processes. In *Knowledge in action: Logical Foundations for specifying and implementing dynamical systems*, pages 149-183, ISBN 0-262-18218-1, MIT, 2001.

15. Richard Watson and Sandeep Chintabathina. Modeling hybrid systems in action languages. In *the proceedings of the 2nd intl ASP'03 workshop*, pages 356-370, Messina, Sicily, Italy, September 2003.

16. Mary Heidt. Developing an inference engine for ASET-Prolog. Master's thesis, University of Texas at El Paso, Dec 2001.

APPENDIX A

IMPLEMENTATION OF WATER TANK EXAMPLE

A-Prolog code for implementing the water tank example 4.1 from chapter 4.

```
% Declarations

% actions

action(turn(open),agent).

action(turn(close),agent).

% processes

process(open,fluent).

process(inflow_rate,fluent).

process(outflow_rate,continuous).

process(volume,continuous).

% time

const m=6.

time(0..m).

#domain time(T;T1;T2).

% step

const n=2.

step(0..n).

#domain step(I).

% Range of processes
```

```
range(open,true).

range(open,false).

range(inflow_rate,0).

range(inflow_rate,3).

values(0..60).

% Suppose that the maximum volume of the tank

% whose length is 4 m , breadth is 3 m and height is

% 5 m is 60 cubic meter.

range(volume,Y) :- values(Y).

% The maximum value of outflow_rate is 10.

range(outflow_rate,Y) :-  values(Y),
                          Y <= 10.

% Domain independent axioms

% A process cannot have two values at the same time.

−val(Y2,PN,T,I) :-  val(Y1,PN,T,I),
                    neq(Y2,Y1),
                    process(PN,PT),
                    range(PN,Y1),
                    range(PN,Y2).

% Axioms for defining ‘end’ of a state.
```

```
end(T, I) :-  o(AN, T, I),
              action(AN, AT).

% If no action occurs during a step then the step will
% end at the last possible time unit.
{end(m, I)}1.
% No state can have more than one end.
:-  end(T1, I),
    end(T2, I),
    neq(T1, T2).

% Every state must end.
ends(I) :- end(T, I).
:- not ends(I).


% Axioms that define 'in' and 'out'.
out(T, I) :-  end(T1, I),
              T > T1.
in(T, I) :- not out(T, I).


% The value of fluent does not change with time.
```

```
val(Y,PN,T,I) :-  val(Y,PN,0,I),
                  in(T,I),
                  process(PN,fluent),
                  range(PN,Y).
```

% Inertia axiom: "Things normally stay as they are."

```
val(Y,PN,0,I+1) :-  val(Y,PN,T,I),
                    end(T,I),
                    not -val(Y,PN,0,I+1),
                    process(PN,PT),
                    range(PN,Y).
```

% Reality Check (obs(Val,Process,T,I))

```
:-  obs(Y,PN,T,I),
    not val(Y,PN,T,I),
    process(PN,PT),
    range(PN,Y).

o(AN,T,I) :-  hpd(AN,T,I),
              action(AN,AT).
```

% Axiom for defining initial state.

val(Y,PN,0,0) :- $obs(Y, PN, 0, 0)$,

$process(PN, PT)$,

$range(PN, Y)$.

% Process definitions

% 1. open

% Dynamic causal law describing effects of 'turn(open)'

$val(true, open, 0, I + 1)$:- $o(turn(open), T, I)$.

% Executability conditions for 'turn(open)'

:- $o(turn(open), T, I)$,

$val(true, open, T, I)$.

% Dynamic causal law describing effects of 'turn(close)'

$val(false, open, 0, I + 1)$:- $o(turn(close), T, I)$.

% Executability conditions for 'turn(close)'

:- $o(turn(close), T, I)$,

$val(false, open, T, I)$.

% 2. inflow_rate

% State constraints defining 'inflow_rate'

% If the faucet is open , inflow is 3.

$val(3, inflow\_rate, T, I)$:- $val(true, open, T, I)$.

% If the faucet is closed, inflow is 0.

$val(0, inflow\_rate, T, I)$:- $val(false, open, T, I)$.

% 3. outflow_rate

% State constraints defining ‘outflow_rate’

% outflow_rate is a function of the water level of the tank

$function\ f4'$.

$val(f4'(Y), outflow\_rate, T, I)$:- $val(Y, volume, T, I)$,
$range(volume, Y)$.

% 4. volume

% State constraints defining ‘volume’

$function\ f3'$.

$val(f3'(Y0, N), volume, T+1, I)$:- $val(Y0, volume, T, I)$,
$val(N, inflow\_rate, T, I)$,
$in(T+1, I)$,
$range(volume, Y0)$,
$range(inflow\_rate, N)$.

% History

$obs(25, volume, 0, 0).$

$obs(false, open, 0, 0).$

$hpd(turn(open), 0, 0).$

$hpd(turn(close), 3, 1).$

% Defining some relations to get better looking output.

```
volume(Y, T, I) :-  val(Y, volume, T, I),
                    range(volume, Y).

outflow_rate(Y, T, I) :-  val(Y, outflow_rate, T, I),
                          range(outflow_rate, Y).

inflow_rate(Y, T, I) :-  val(Y, inflow_rate, T, I),
                         range(inflow_rate, Y).
```

% SMODELS directives to hide and show relations.

```
hide.

show  inflow_rate(Y, T, I),
      volume(Y, T, I),
      outflow_rate(Y, T, I).
```