TOWARDS ANSWER SET PROGRAMMING BASED ARCHITECTURES FOR INTELLIGENT AGENTS

by

SANDEEP CHINTABATHINA, M.S.

A Ph.D. DISSERTATION

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Richard Watson
Committee Chairman

Dr. Michael Gelfond

Dr. Yuanlin Zhang

Dr. Marcello Balduccini

Ralph Ferguson
Dean of the Graduate School

December, 2010

©2010, Sandeep Chintabathina

To my parents.

ACKNOWLEDGEMENTS

I would like to thank my adviser Richard Watson for his guidance and support during my days as a graduate student. I am proud to be his first doctoral student. We wrote several papers together which taught me several things about writing technical papers. His problem solving capabilities are amazing and I benefited from that during our discussions.

Michael Gelfond laid the foundation for this work when I started graduate school. He is one of the best computer scientists I have ever known. His strong mathematical background, ability to concentrate and perseverance have inspired me to become what I am today. Working with him I have learned how to organize my research and tackle difficult problems without getting discouraged. By attending his talks and classes I have learned a lot about giving talks and teaching classes. I am very thankful for all the contributions he is making in my life.

I am thankful to Marcello Balduccini for being on my committee and giving valuable comments on my dissertation. He is truly an inspiring and active researcher. I am also thankful to Yuanlin Zhang for being on my committee. I enjoyed my discussions with him during our joint work on another project.

I would like to thank all my fellow KR lab members. In some way or another they have made a contribution towards this dissertation. We discussed several topics and attended several seminars together which has helped me to understand a wide variety of topics. I would especially like to thank Ricardo for being there for me since the day I started graduate school. We had several interesting discussions that sometimes ended up in a trip to the nearest restaurant which was a lot of fun. Besides being lab mates we are very good friends.

I would like to thank all my family members. Especially I would like to thank my parents, my sister and her family for their patience and support through all these years. Next, I would like to thank my wife, Tanecia, for her constant support during the last few years. She pushed me and inspired me to finish this dissertation. Without

her moral and emotional support it would have been difficult to finish this work. I am very glad to have her beside me.

I would like to thank the faculty and staff of the Computer Science department for their academic and financial support over the years.

I would like to thank all my friends who have been there for me during good times as well as difficult times. They made my stay in Lubbock very exciting. I wish them all a bright and wonderful future. I would like to thank the staff at the international cultural center for organizing various festivals and events that brought me and my friends together. I would also like to thank the faculty and staff of the Spanish department for giving me an opportunity to learn Spanish. Texas Tech and Lubbock will surely be missed!

CONTENTS

ABSTRACT

The design of intelligent agents is an important research area in the field of Artificial Intelligence. Research in this area has led to the development of agent architectures that support various tasks such as reasoning, planning, diagnosis etc. One such architecture is based on the agent repeatedly executing the *observe-think-act* loop. In this architecture a dynamic system is viewed as a transition diagram whose nodes represent possible physical states of the system and whose arcs are labeled by actions. One of the approaches to describing these diagrams is a theory based on *action languages*, which are high-level languages for reasoning about actions and their effects. One such action language is $\mathcal{AL}$. A theory in $\mathcal{AL}$ (also called an *action description*) describes a transition diagram that contains all possible trajectories of a given dynamic system. However, it was not designed to reason about properties of a domain that change continuously with time.

In this dissertation we present action language H which extends $\mathcal{AL}$ with the ability to reason about continuous change. We design this language by extending the signature of $\mathcal{AL}$ with a collection of numbers for representing continuous time and a collection of functions defined over time (processes). Like $\mathcal{AL}$, H is based on transition diagram based semantics. We model a variety of examples in H to demonstrate that H is very useful for knowledge representation. We compared H with other approaches and discovered that action descriptions of H are simpler, concise and elaboration tolerant. We studied timed automata and discovered that H expands timed automata.

An action description of $\mathcal{AL}$ is implemented by translating it into a program of answer set programming (ASP) and computing answer sets of the resulting program. Thus, various tasks of the agent can be reduced to asking questions about answer sets of programs. In this dissertation, we came up with an encoding of action descriptions of H into a variant of ASP called $\mathcal{AC}$. Using this encoding, several agent tasks can be reduced to asking questions about answer sets of $\mathcal{AC}$ programs. We proved that this encoding is correct. We are able to run our encodings using existing systems and

confirm that the resulting answer sets are the ones we expected.

LIST OF FIGURES

CHAPTER I
INTRODUCTION

This dissertation is a contribution towards the design of intelligent agents acting in a changing environment. An intelligent agent is a software entity capable of reasoning, planning and acting on its own. The design of such agents is an important research area in the field of artificial intelligence. Research in this area has led to development of agent architectures that support various tasks such as planning, diagnosis, learning etc. One such architecture is based on the agent repeatedly executing the *observe-think-act-loop* [6, 23]. This architecture is applicable if the dynamic system which includes the agent and its environment is viewed as a transition diagram whose states correspond to possible physical states of the system and whose arcs are labeled by actions. A transition, $\langle \sigma, a, \sigma' \rangle$, of a diagram denotes that action a is possible in state σ and that after the execution of a the system may move to state σ'. The diagram consists of all possible trajectories of the system. One of the approaches to describing these diagrams is a theory based on *action languages* - high-level languages for reasoning about actions and their effects [18]. A theory in an action language (often called an *action description*) describes a transition diagram that contains all possible trajectories of a given dynamic system.

Currently there are several action languages that are used to study different features of dynamic domains. For instance action language $\mathcal{AL}$ [6, 42] is simply an extension of action language $\mathcal{A}$ [17] by state constraints which express causal relations between *fluents* [1]. The semantics of $\mathcal{AL}$ formalize McCarthy's *Principle of inertia* which says that "*Things tend to stay the same unless they are changed by actions*" [33]. Let us look at an example domain and see how $\mathcal{AL}$ can be used to represent knowledge in that domain.

Consider an agent holding a brick at a certain height above the ground. The agent is capable of dropping the brick and catching it. The effects of *drop* and *catch* on

[1] functions whose values depend on a state and may change as a result of actions

boolean fluent *holding* are captured using statements of $\mathcal{AL}$

drop causes $\neg$*holding*

catch causes *holding*

The first statement says that if *drop* is executed in some state then *holding* will be false in the resulting state. The second statement says that if *catch* is executed in some state then *holding* will be true in the resulting state. We can specify conditions under which *drop* and *catch* are impossible using statements

impossible *drop* if $\neg$*holding*

impossible *catch* if *holding*

The transition diagram described by this action description is

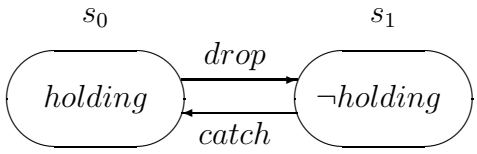

Figure 1.1: Transitions caused by *drop* and *catch*

Now let us consider the following scenario. Suppose the agent is holding the brick at 500 meters above the ground and wants to catch it at 300 meters above the ground. To achieve this the agent can drop the brick and then catch it at exactly 300 meters above the ground. In one approach the agent can compute the height of the brick as it falls to the ground and perform catch when the desired height is reached. In another approach the agent can determine the time it takes to descend 200 meters and schedule the catch action. In the first approach, as the brick falls under the influence of gravity its *height* changes continuously with time. *Height* is defined by a function of time also called a *process*. However, once the brick is caught its *height* becomes a constant. The *height* of the brick can be considered as a special kind of fluent that changes not only as a result of actions but also continuously with time. We refer to

fluents whose values may change continuously with time as *process fluents*. Domains consisting of both process and non-process fluents are called *hybrid domains*.

From the example, it is clear that $\mathcal{AL}$ can capture non-process fluents such as *holding*. However, it was not designed to capture process fluents. There are logic-based formalisms such as Situation Calculus and Event Calculus that are capable of reasoning about process fluents. Situation calculus, developed by John Mccarthy in 1962, is based on classical logic and uses logical entailment for reasoning about actions. Reiter and fellow researchers extended the language of situation calculus (sitcalc) [39] to incorporate time, concurrency and other features. Event calculus, developed by Kowalski and Sergot, was an alternative to situation calculus for reasoning about actions. Shanahan [40] extended event calculus to be able to reason about process fluents. Both approaches demonstrate via examples how their approach can be used for reasoning about process fluents. However, it is difficult to express causal relations between fluents in both approaches. There is an action language based approach for reasoning about process fluents. The language is called $\mathcal{ADC}$ and was developed by Baral, Son and Tuan Le. This language is based on action language $\mathcal{A}$ and does not support state constraints.

In this dissertation we are interested in agents acting in hybrid domains. We present a new action language capable of representing knowledge in such domains. The language is called H which is an abbreviation for *Hybrid*. We developed this language by extending the signature of $\mathcal{AL}$ with

- a collection of numbers for representing time
- a collection of functions defined over time (*processes*)
- fluents with non-boolean values including fluents whose values are functions of time (process fluents).

Time can be either discrete or continuous depending upon the type of domain. The extended signature allows us to add process fluents and processes to the statements of H. So we end up extending $\mathcal{AL}$ with the ability to reason about process fluents.

The semantics of $\mathcal{AL}$ is based on the McCain-Turner equation [31]. We modify this equation slightly to define the semantics of H. In this way both languages are based on the same underlying intuition.

We presented an earlier version of H in [9]. However, in this dissertation we refined the syntax of the language and enhanced the language with triggers which allows us to specify conditions under which actions are triggered.

In this dissertation we model a variety of examples in H to demonstrate that H is very useful for knowledge representation. We compared action descriptions of H with logical theories of situation calculus and discovered that action descriptions of H are simpler, concise and elaboration tolerant. We establish relationship between H and *timed automata* [1] which was developed to reason about functions that change continuously with time. However, only a particular type of function (clock) is allowed in this formalism. We provide more details in chapter 6.

An action description of $\mathcal{AL}$ can be viewed as the specification of a domain. It is implemented by translating it into a program of answer set programming (ASP) and computing answer sets [15, 16] of the resulting program. ASP is a *declarative* knowledge representation language that is well suited for difficult, primarily NP-hard, search problems. [28]. It can be used to represent and reason with recursive definitions, defaults and their exceptions, causal relations, beliefs and various forms of incomplete information.

A program of ASP contains statements that represent information relevant to the problem being solved. The answer set semantics assign a collection of answer sets to such a program. An answer set is a possible set of beliefs which can be built by a rational reasoner on the basis of the statements of the program and the *rationality principle* which states that *one shall not believe anything one is not forced to believe.* Currently, there are several inference engines (solvers) for computing answer sets of programs. Answer sets encode solutions to the given problem. When an action description is translated into a program of ASP, various tasks of the agent are reduced to asking questions about answer sets of the program.

In this dissertation we came up with an encoding of action descriptions of H into a variant of ASP called $\mathcal{AC}$ which integrates ASP and constraint logic programming (CLP). The combination of non-monotonic logic with numerical constraint solving is quite suitable for H. Using this encoding, agent tasks such as prediction and planning are reduced to computing answer sets of $\mathcal{AC}$ programs. We proved that this encoding is correct. With the help of new solvers such as EZCSP [2] and LUNA [37] we were able to run our encodings and confirm that the resulting answer sets are the ones we expected. In the past we used traditional ASP solvers to compute answers sets of programs involving numerical computations. These solvers are very inefficient when dealing with numbers. But thanks to the new solvers and our encoding, we are able to compute answer sets in a reasonable amount of time.

We summarize our approach as follows. We are interested in building intelligent agents acting in hybrid domains. To do this we represent knowledge of the agent in some language, then encode this knowledge as a logic program and compute models of this program. Thus, reducing various tasks of the agent to asking questions about models of logic programs. Our choice for an action language is H and our choice for a logic programming language is $\mathcal{AC}$. Using our approach we are able to model domains that could not be modeled properly in the past.

The dissertation is organized as follows. In chapter 2 we define the syntax and semantics of language H. In chapter 3 we demonstrate that H can model various types of domains and is therefore a good knowledge representation language. In chapter 4 we show how to encode action descriptions of H into $\mathcal{AC}$ programs. In chapter 5 we talk about the solvers we use for computing answer sets of our encodings. In chapter 6 we compare H with existing formalisms such as timed automata and situation calculus. In chapter 7 we present proofs of our theorems. Finally, chapter 8 contains conclusions and future work.

[2] http://marcy.cjb.net/ezcsp/index.html

CHAPTER II
SYNTAX AND SEMANTICS OF H

In this chapter we present the syntax and semantics of language *H*. The first section covers the syntax and next section covers the semantics. Towards the end of this chapter we will give an example to illustrate the syntax and semantics.

2.1 Syntax

By *sort* we mean a non-empty countable collection of strings in some fixed alphabet. A *sorted signature* Σ is a collection of sorts and function symbols.

A *process signature* is a sorted signature with special sorts *time*, *action*, and *process*. Sort *time* is normally identified with one of the standard numerical sorts with the exception that it contains an ordinal ω such that for any $x \in time \setminus \{\omega\}$, $\omega > x$. No operations are defined over ω. If time is discrete, elements of $time \setminus \{\omega\}$ may be viewed as non-negative integers, otherwise they can be interpreted as either rational numbers, constructive real numbers, etc.

Sort *process* contains strings of the form $\lambda T.p$ where T is a variable ranging over *time* and p is a mathematical expression defining function of T. A string $\lambda T.p$ represents a function defined over *time*. The λ is said to bind T in p. If p does not contain T then $\lambda T.p$ is a constant function. For simplicity we assume that all functions from *process* have the same range denoted by the sort $range(process)$. An example of a function from sort *process* is $\lambda T.10 - 4.9 * (T - 5)^2$ which defines the height of a freely falling object that was dropped from a height of 10 meters at time 5.

Sort *action* is divided into subsorts *agent* and *exogenous*. Elements of *agent* are actions performed by an agent and elements of *exogenous* are actions that are not performed by an agent. Both agent and exogenous actions will be referred to as actions.

The collection of function symbols includes standard numerical functions and fluents.

Intuitively, *fluents* are properties that may change as a result of actions. For example, the *height* of a brick held at a certain height above the ground could change when it is dropped. Every process signature contains reserved fluents *start* and *end* of sort *time*.

A *term of sort* s is either

1. a string $y \in s$ or
2. a fluent of sort s or
3. a standard arithmetic term of sort s

Notice that if $f(\bar{x})$ is a term of sort *process* and t is a term of sort *time* then $f(\bar{x})(t)$ is a term of sort $range(process)$. For example, to represent the height of brick b we can introduce a fluent $height(b)$ of sort *process*. By $height(b)(5)$ we denote the height of b at time 5.

An *atom* of Σ is an expression of the form $t = y$ where t is a term of some sort s and $y \in s$. Examples of atoms are $end = 10$, $4 < 5$, $height(b)(5) = 20$ etc. An atom in which t is a fluent is called a *fluent atom*. An example of a fluent atom is $height(b) = \lambda T.10 - 4.9 * (T - 5)^2$.

A *literal* of Σ is an atom or its negation. Negation of $=$ will be often written as $\neq$. If t is a term of boolean sort then $t = true$ ($t \neq false$) is often written as t and $t = false$ ($t \neq true$) is often written as $\neg t$.

Language, H, is parameterized by a process signature Σ with standard interpretations of numerical functions and relations (such as $+, <, \leq, \neq$, etc) and the sort *process*.

Definition 2.1.1. An *action description* of $H(\Sigma)$ is a collection of statements of the form:

$$l_0 \text{ if } l_1, \ldots, l_n. \quad (1)$$

$$e \text{ causes } l_0 \text{ if } l_1, \ldots, l_n. \quad (2)$$

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n. \quad (3)$$

$$l_1, \ldots, l_n \text{ triggers } e. \quad (4)$$

where e's are elements of *action*, l_0's are fluent atoms and $l_1, \ldots, l_n$ are literals of the signature of H. l_0 is referred to as the *head* of a statement and $l_1, \ldots, l_n$ are referred to as the *body* of a statement.

A statement of the form (1) is called a *state constraint*. It guarantees that any state satisfying $l_1, \ldots, l_n$ also satisfies l_0. A statement of the form (2) is called a *dynamic causal law* and it states that if action, e, were executed in a state satisfying literals $l_1, \ldots, l_n$ then any successor state would satisfy l_0. A statement of the form (3) is called an *executability condition* and it states that actions $e_1, \ldots, e_m$ cannot be executed in a state satisfying $l_1, \ldots, l_n$. If $n = 0$ then *if* is dropped from statements (1), (2) and (3). A statement of the form (4) is called a *trigger* and it states that action e is triggered in any state satisfying $l_1, \ldots, l_n$.

Note that statements of an action description of H contain no variables other than the variable for time T. However, we will allow other variables to occur in a statement as long as that statement is considered as a shorthand for the collection of statements obtained by replacing each occurrence of a variable other than T by its corresponding ground instances.

As we can see statements of H are very similar to statements of $\mathcal{AL}$. In fact, all statements except for statements of the form (4) are similar to statements of $\mathcal{AL}$. The difference is that we allow literals such as $f = 5$ and $p = \lambda T.T^2$ to appear in the statements of H where f and p are fluents. The following proposition specifies conditions under which an action description of H is syntactically equivalent to an action description of $\mathcal{AL}$.

Proposition 2.1.1. An action description of H that does not contain triggers and whose statements contain literals involving only boolean fluents is an action description of $\mathcal{AL}$.

2.2 Semantics

The semantics of language H is based on a slightly modified McCain-Turner equation [31]. An action description, AD, of H(Σ) describes a transition diagram, $TD(AD)$, whose nodes correspond to possible physical states of a system and whose arcs are labeled by actions. A transition $\langle s, a, s' \rangle$ of the diagram denotes that action a is possible in s and as a result of execution of a the system will move to state s'. In this section we will give a formal definition for a state and a transition of $TD(AD)$. We begin with interpreting symbols of Σ.

Definition 2.2.1. Given an action description AD of $H(\Sigma)$, an *interpretation* I of Σ is a mapping defined as follows.

- for every non-process sort, s, and every string $y \in s$, I maps y into itself i.e. $y^I = y$.
- standard interpretation is used for the sort *process* and other standard numerical functions and relations.
- I maps every fluent into a properly typed function.

Often an interpretation I of Σ is identified with a collection, $s(I)$, of atoms of the form $t = y$ such that $t^I = y$ where t and y are terms of some sort. In other words, $s(I) = \{t = y \mid t^I = y\}$.

Before we give the definition of a state of $TD(AD)$ let us consider the following definitions. First, let us define what it means for a literal to be true w.r.t a set of atoms of Σ.

Definition 2.2.2. Given a consistent set, L, of atoms of Σ

- An atom $t = y$ is *true in* L (symbolically $L \models t = y$) iff $t = y \in L$.
- A literal $t \neq y$ is *true in* L ($L \models t \neq y$) iff $L \models t = y_0$ and $y \neq y_0$.

We will now define what it means for a set of atoms to be closed under state constraints of AD.

Definition 2.2.3. A *set L of atoms is closed under the state constraints* of AD if for every state constraint

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

of AD if $L \models l_i$ for every i, $1 \leq i \leq n$ then $L \models l_0$.

Next, we define what it means for a set of atoms to satisfy a trigger of AD.

Definition 2.2.4. A *set L of atoms of H satisfies a trigger*

$$l_1, \ldots, l_n \text{ triggers } e$$

of AD iff $L \models l_i$ for every i such that $1 \leq i \leq n$.

Intuitively, if a set of atoms satisfies a trigger it means that the corresponding action will take place at some time point.The next definition characterizes sets of atoms that define the earliest possible occurrence times of triggered actions.

Definition 2.2.5. A *set L of atoms of H is closed under triggers* of AD iff $\neg\exists$ L' such that L' satisfies at least one trigger of AD and $L \setminus L' = \{end = t_2\}$ and $L' \setminus L = \{end = t_1\}$ and $t_1 < t_2$.

Now we are ready to give the definition of a state of $TD(AD)$.

Definition 2.2.6. Given an interpretation I of Σ, $s(I)$ is a *state* of $TD(AD)$ if each of the following holds.

- $s(I)$ is a collection of atoms of the form $t = y$ such that $t^I = y$ where t and y are terms of the same sort.
- $s(I)$ is closed under the state constraints of AD.
- if $s(I) \models start = t_1$ and $s(I) \models end = t_2$ then $t_1 \leq t_2 \wedge t_1 < \omega$.

- $s(I)$ is closed under the triggers of AD.
- if $s(I) \models p = \lambda T.f$ where p is a fluent of sort *process* then $\lambda T.f$ is defined over the domain $\{t \mid start^I \leq t \leq end^I \wedge t < \omega\}$.
- If p is a fluent of sort *process* and t is a term of sort *time* then $s(I) \models p(t) = x$ iff $s(I) \models p = \lambda T.f$ and $\lambda T.f(t^I) = x$.

By definition of interpretation every symbol is mapped uniquely. Therefore, states of $TD(AD)$ are complete and consistent. Whenever convenient the parameter I will be dropped from $s(I)$.

Intuitively, a state can be viewed as a collection of functions of time defined over an interval. The endpoints of the interval are implicitly defined by the reserved fluents *start* and *end*. The domain of each function is the set $\{t \mid start \leq t \leq end \wedge t < \omega\}$. We say that a state is defined over an interval of the form $[start, end]$ iff $end \neq \omega$. There is at least one arc labeled by an action leading out of such a state. We say that a state is defined over an interval of the form $[start, end)$ iff $end = \omega$. There is no arc leading out of such a state. States that begin at time 0 are called *initial states*. They define the initial conditions of a domain.

Now that we have defined what a state is we will define what is means for an action to be possible in a state.

Definition 2.2.7. Action a is *possible* in state, s, if for every non-empty subset a_0 of a, there is no executability condition

$$\text{impossible } a_0 \text{ if } l_1, \ldots, l_n.$$

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

Given a state s and action e let us define what are the direct effects of executing e in s.

Definition 2.2.8. Let e be an elementary action that is possible in state s. By $E_s(e)$ we denote the set of all *direct effects of* e w.r.t s.

$$E_s(e) = \{l_0 \mid e \text{ causes } l_0 \text{ if } l_1, \ldots, l_n \in AD \wedge s \models l_i \text{ for every } i, 1 \leq i \leq n\}$$

If a is a compound action then $E_s(a) = \bigcup_{e \in a} E_s(e)$.

The following definition allows us to identify set of literals with adjacent intervals.

Definition 2.2.9. Let x, y, and z be elements of sort *time* such that $x \leq y \leq z \wedge y < \omega$ and s and s' be sets of literals of H. We say that s' *follows* s iff $s \models \{start = x, end = y\}$ and $s' \models \{start = y, end = z\}$.

Given two sets of literals s and s' we will use the notation $T_s(s')$ to project the interval of s'. Therefore,

$$T_s(s') = \begin{cases} \{start = t_1, end = t_2\} \text{ if } s' \text{ follows } s \wedge \{start = t_1, end = t_2\} \subseteq s'. \\ \emptyset \text{ otherwise.} \end{cases}$$

The consequences of a set of atoms w.r.t a set of state constraints is defined as follows.

Definition 2.2.10. Given a set S of atoms and a set Z of state constraints of AD the set, $Cn_Z(S)$, of *consequences of* S *under* Z is the smallest set of atoms (w.r.t set theoretic inclusion) containing S and closed under Z.

Definition 2.2.11. *Action* a *is complete w.r.t a set of literals* s if for every trigger r of the form

$$l_1, \ldots, l_n \text{ triggers } e$$

$e \in a$ iff s satisfies r.

We know that a state contains arbitrary atoms of Σ. However, for the next definition we will focus our attention on fluent atoms and atoms formed from *start* and *end*. Other atoms belonging to a state will be ignored because they are either universally true or could be derived from fluent atoms.

Definition 2.2.12. A transition diagram $TD(AD)$ is a tuple $\langle \phi, \psi \rangle$ where

- ϕ is the set of states.
- ψ is the set of all transitions $\langle s, a, s' \rangle$ such that each of the following holds.
  - a is complete w.r.t s
  - a is possible in s
  - s' follows s
  - s' is closed under the triggers of AD
  - $$s' = Cn_Z(E_s(a) \cup (s \cap s') \cup T_s(s')) \tag{2.1}$$

    where Z is the set of state constraints of AD.

The set, $E_s(a)$, consists of direct effects of a while the set, $s \cap s'$, consists of facts preserved by inertia. $T_s(s')$ projects the *start* and *end* of s'. The application of Cn_Z to the union of these sets adds the indirect effects.

We will revisit the brick drop example from chapter 1 to illustrate the syntax and semantics of H. The example demonstrates how H can be used for modeling continuous change.

Example 2.2.1 (Continuous change)**.** Consider an agent acting in a domain consisting of a brick. The brick is held above the ground by the agent. The actions available to the agent are *drop* and *catch*. Dropping the brick causes the *height* of the brick to change continuously with time as defined by Newton's laws of motion.

Suppose that the agent prefers to catch the brick only if it is more than 100 units above the ground. In such a situation the agent needs to determine the height of the falling brick at various time points and decide whether or not to catch the brick. The agent can use a formal language like H to model continuous changes in the presence of actions.

Let $\mathcal{A}_0$ be an action description of H. Signature $\Sigma(\mathcal{A}_0)$ consists of boolean fluent *holding*, process fluent *height*, and actions *drop* and *catch*. $\Sigma(\mathcal{A}_0)$ contains auxillary fluent *time_changed* to denote the latest time point at which the brick was either dropped or caught. It also contains fluent *ht_changed* to denote height of the brick at the time indicated by *time_changed*. All fluents are inertial. Let sort *process* contain functions ranging over $\mathcal{R}$.

$$process = \{\lambda T.20 - 4.9 * (T-1)^2, \ldots, \ldots, \lambda T.10 - 4.9 * (T-5)^2, \ldots, \ldots\}$$

As mentioned earlier, *height* ranges over *process*. The corresponding causal laws are as follows.

drop causes $\neg holding$ (1)

catch causes *holding* (2)

impossible *drop* if $\neg holding$ (3)

impossible *catch* if *holding* (4)

impossible *drop* if $height(end) = 0$ (5)

impossible *catch* if $height(end) = 0$ (6)

drop causes $ht\_changed = X$ if $height(end) = X$ (7)

drop causes $time\_changed = T_0$ if $end = T_0$ (8)

catch causes $ht\_changed = X$ if $height(end) = X$ (9)

catch causes $time\_changed = T_0$ if $end = T_0$ (10)

$height = \lambda T.X - 4.9 * (T - T_0)^2$ if $ht\_changed = X$, $\neg holding$, $time\_changed = T_0$ (11)

$height = \lambda T.X$ if $ht\_changed = X$, $holding$ (12)

The effects of *drop* and *catch* on *holding* are captured by causal laws (1) and (2) respectively. Executability condition (3) states that *drop* cannot be executed if the agent is not *holding* the brick. Similarly, (4) states that *catch* cannot be executed if the agent is already *holding* the brick. Executability conditions (5) and (6) state

that *drop* and *catch* cannot be executed if the height of the brick at the end of a state is zero. Since every action is executed only at the end of a state, it is sufficient to specify conditions on the height of the brick at the end of a state. Causal laws (7), (8), (9) and (10) state the effects of *drop* and *catch* on *ht_changed* and *time_changed*. Both actions reset the values of these auxillary fluents which are necessary to keep the domain markovian. State constraint (11) states that if the brick is not held then *height* is defined by a function of time which is obtained from Newton's laws of motion. Finally, state constraint (12) states that if the brick is held then *height* remains constant.

The transition diagram, $TD(\mathcal{A}_0)$, contains an infinite number of states and transitions. We will look at a particular trajectory of this diagram with initial state s_0 defined as follows.

$$\begin{aligned} s_0 = \ & \{height = \lambda T.500, holding, ht\_changed = 500, \\ & time\_changed = 0, start = 0, end = 5\} \end{aligned}$$

Action *drop* is possible in s_0. There are several successor states of s_0 w.r.t *drop* all of which differ in the value of *end*. Let us consider the following candidate for a successor state.

$$\begin{aligned} s_1 = \ & \{height = \lambda T.500 - 4.9 * (T-5)^2, \neg holding, ht\_changed = 500, \\ & time\_changed = 5, start = 5, end = 8\} \end{aligned}$$

Let us check whether s_1 satisfies the modified McCain-Turner equation. The direct effects of *drop* are encoded by the set $E_{s_0}(drop)$.

$$E_{s_0}(drop) = \{\neg holding, ht\_changed = 500, time\_changed = 5\}$$

As we can see nothing is carried over by inertia from state s_0 to s_1. We also have

$$T_{s_0}(s_1) = \{start = 5, end = 8\}$$

The consequences of the set $E_{s_0}(drop) \cup T_{s_0}(s_1)$ w.r.t state constraints (11) and (12)

gives the set

$$\{height = \lambda T.500 - 4.9 * (T-5)^2, \neg holding, ht\_changed = 500,$$
$$time\_changed = 5, start = 5, end = 8\}$$

which is the same as s_1. So s_1 satisfies the modified McCain-Turner equation. Hence, we conclude $\langle s_0, drop, s_1 \rangle$ is a transition of $TD(\mathcal{A}_0)$.

Now consider the state s_1. Action *catch* is possible in s_1 and a successor of s_1 w.r.t *catch* is determined using the approach described above. The trajectory $\langle s_0, drop, s_1, catch, s_2 \rangle$ of $TD(\mathcal{A}_0)$ is depicted in figure 2.1. In the diagram, we ignore auxillary fluents and use h as an abbreviation for *height*.

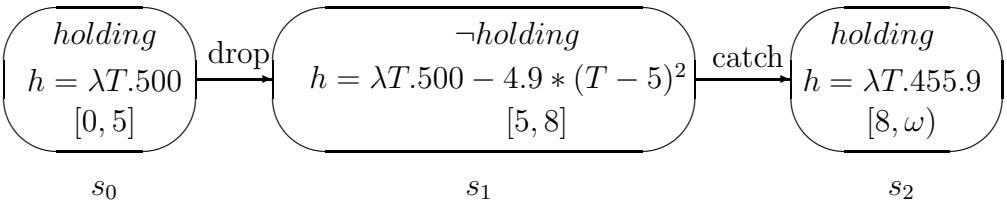

Figure 2.1: Transitions caused by *drop* and *catch*

As we can see, each state has an interval associated with it and fluents are mapped into functions defined over this interval. This is a major difference between transition diagrams described by H and $\mathcal{AL}$.

The interval $[8, \omega)$ associated with state s_2 implies that no actions take place in s_2 and the domain remains in s_2 for a very long time. We also see that the end of one state coincides with the start of the next state. For this reason when an action changes the value of a fluent it is possible that the fluent has one value at the end of current state and an other value at the start of the successor state. This implies that the fluent is not uniquely defined at the shared time point. We consider these time points as the transition points for a fluent. In figure 2.1 it is possible to see that time points 5 and 8 are transition points for *holding*.

CHAPTER III
KNOWLEDGE REPRESENTATION IN H

In this chapter we demonstrate how to represent knowledge in H. We present some examples to demonstrate that H can be used for modeling various types of domains involving actions with delayed effects, actions with duration, resources and so on. Later on, we will develop a methodology for writing action descriptions of H. Our first example involves reasoning about resources in a continuous domain.

3.1 Examples

Example 3.1.1 (Reasoning about resources)**.** Consider a domain consisting of 2 faucets and a sink. The faucets can be opened and closed. When a faucet is open it discharges fluid at a rate of 3 gallons a minute into the sink. If the faucet is closed no fluid is discharged. We would like to determine the volume of fluid in the sink as the faucets are being opened and closed.

In order to model this domain let us introduce an action description $\mathcal{A}_1$ of H. Let $faucet$ be a sort containing names f_1 and f_2 for faucet 1 and faucet 2 respectively. Signature $\Sigma(\mathcal{A}_1)$ consists of the sort $faucet$, process fluent $volume$, boolean fluents $opened(f_1)$ and $opened(f_2)$, numeric fluents $outflow\_rate(f_1)$ and $outflow\_rate(f_2)$ and actions $open(f1)$, $open(f2)$, $close(f1)$, and $close(f2)$. We also have auxillary fluents $time\_changed$, which denotes the last time point at which a faucet position was changed, and $v\_changed$ which denotes the volume of the sink at the time denoted by $time\_changed$. For simplicity we will assume that there is no upper limit on the

volume of the sink. The corresponding causal laws are as follows.

$$open(F) \text{ causes } opened(F) \tag{1}$$

$$\text{impossible } open(F) \text{ if } opened(F) \tag{2}$$

$$close(F) \text{ causes } \neg opened(F) \tag{3}$$

$$\text{impossible } close(F) \text{ if } \neg opened(F) \tag{4}$$

$$open(F) \text{ causes } v\_changed = X \text{ if } volume(end) = X \tag{5}$$

$$close(F) \text{ causes } v\_changed = X \text{ if } volume(end) = X \tag{6}$$

$$open(F) \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{7}$$

$$close(F) \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{8}$$

$$outflow\_rate(F) = 3 \text{ if } opened(F) \tag{9}$$

$$outflow\_rate(F) = 0 \text{ if } \neg opened(F) \tag{10}$$

$$volume = \lambda T.X + \int_{T_0}^{T}(Y + Z)dT \text{ if } \begin{array}{l} outflow\_rate(f_1) = Y, \\ outflow\_rate(f_2) = Z, \\ v\_changed = X \\ time\_changed = T_0. \end{array} \tag{11}$$

Causal laws (1) and (3) state the effects of *open(F)* and *close(F)* on *opened(F)*. We use F as a shorthand for faucets names f_1 and f_2. Executability condition (2) states that a faucet which is already open cannot be opened. Similarly, (4) states that a faucet which is already closed cannot be closed. Causal laws (5) thru (8) state the effects of opening and closing a faucet on fluents *v_changed* and *time_changed*. No other actions will alter the values of these auxillary fluents. State constraints (9) and (10) define $outflow\_rate$ of a faucet when the faucet is open and closed respectively. Finally, state constraint (11) defines *volume* by adding up the contributions made by each faucet over time to the existing volume.

As we can see, in example (2.2.1) and example (3.1.1) we use state constraints to define process fluents. This approach leads to elaboration tolerant action descriptions. For example, if a third faucet is added to example (3.1.1) then it will require only a minor change to the state constraint. It must be noted that if situation calculus is

used to model the domain from example (2.2.1) then *height* will be defined using a successor state axiom which is the counterpart of a dynamic causal law. In fact, every process fluent is defined using a successor state axiom. The use of state constraints is not encouraged because it may lead to unintuitive results. The problem with using successor state axioms for everything is that the axioms can become very long and complicated. The resulting theory may be less elaboration tolerant. For example, imagine specifying *volume*, from example (3.1.1), as a direct effect of opening and closing faucets f_1 and f_2. It can get very complicated. Our second example domain contains actions with delayed effects.

Example 3.1.2 (Actions with delayed effects)**.** Consider an agent acting in a domain consisting of a timer. The agent is capable of setting up and turning off the timer. The effect of setting up the timer to x units of time is that an alarm will sound after x units of time have elapsed. The action has a delayed effect instead of an immediate effect.

Let us see how we can use language H to model such delayed effects. We can determine whether an alarm will sound depending upon how much time is left on the timer. Let $\mathcal{A}_2$ be an action description of H. The corresponding signature $\Sigma(\mathcal{A}_2)$ consists of process fluent *alarm*, boolean fluent *active*, action $set\_timer(X)$ which denotes the action of setting the alarm to some time X and action $turn\_off$ to deactivate the timer. We have a fluent *timer* to denote the time to which the timer was set and fluent *time_changed* to denote the time at which the timer was set. The fluent *alarm* denotes whether or not the alarm is sounding. Fluent *active* denotes whether or not the alarm is active. Let sort *process* contain functions

$$process = \{f(0,0,T), f(0,1,T), \ldots, \ldots, \ldots\}$$

Given that the timer is set to x units at time t_0, function $f(x, t_0, T)$ returns true if there is no time left on the timer at time $T \geq t_0$ and false otherwise. Here is the

definition.

$$f(x, t_0, T) = \begin{cases} true \text{ if } x - T + t_0 \leq 0 \\ false \text{ otherwise} \end{cases}$$

The corresponding causal laws are as follows.

$$set\_timer(X) \text{ causes } active \tag{1}$$

$$set\_timer(X) \text{ causes } timer = X \tag{2}$$

$$set\_timer(X) \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{3}$$

$$turn\_off \text{ causes } \neg active \tag{4}$$

$$alarm = \lambda T.f(X, T_0, T) \text{ if } \begin{array}{l} timer = X, \\ time\_changed = T_0, \\ active. \end{array} \tag{5}$$

$$alarm = \lambda T.false \text{ if } \neg active. \tag{6}$$

Dynamic laws (1) thru (3) capture the effects of $set\_timer(X)$ on fluents *active*, *timer* and *time_changed* respectively. Dynamic law (4) captures the effect of $turn\_off$ on *active*. State constraint (5) states that if the alarm is *active* then it may or may not sound depending upon the amount of time left on the timer. State constraint (6) states that if the alarm is not *active* then it will not sound.

Our next example domain contains actions with duration.

Example 3.1.3 (Actions with durations)**.** Consider a domain consisting of an agent capable of driving a car. The action of driving a car is an action with duration. As Reiter suggested in [39] an action with duration can be considered as a process that is started and terminated by two instantaneous actions. In this example, instantaneous actions *start_car* and $turn\_off$ will initiate and terminate the fluent *started*. The effects of these actions can be easily modeled using action languages capable of handling discrete properties. However, if we are interested in determining how much gas is left in the car's tank we need language H. This is because as the car is running, gas is being consumed continuously.

Let $\mathcal{A}_3$ be an action description of H. The corresponding signature $\Sigma(\mathcal{A}_3)$ consists of actions *start_car* and $turn\_off$, boolean fluent *started*, numeric fluent *consume_rate* and process fluent *gas_volume*. Fluent *consume_rate* denotes the rate at which gas is being consumed. We have auxillary fluent *time_changed* to denote the last time at which the car was started or turned off. We also have *v_changed* to denote the volume of gas in the car at the time denoted by *time_changed*. The corresponding causal laws are as follows.

start_car causes *started* (1)

impossible *start_car* if *started* (2)

impossible *start_car* if $gas\_volume(end) = 0$ (3)

$turn\_off$ causes $\neg started$ (4)

impossible $turn\_off$ if $\neg started$ (5)

start_car causes $v\_changed = X$ if $gas\_volume(end) = X$ (6)

start_car causes $time\_changed = T_0$ if $end = T_0$ (7)

$turn\_off$ causes $v\_changed = X$ if $gas\_volume(end) = X$ (8)

$turn\_off$ causes $time\_changed = T_0$ if $end = T_0$ (9)

$consume\_rate = 5$ if *started* (10)

$consume\_rate = 0$ if $\neg started$ (11)

$$gas\_volume = \lambda T.max(0, X - \int_{T_0}^{T} Y\ dT) \quad \text{if } consume\_rate = Y, \quad (12)$$
$$v\_changed = X,$$
$$time\_changed = T_0.$$

Dynamic laws (1) and (4) capture the effects of *start_car* and $turn\_off$ on *started*. Executability conditions (2) and (3) state that *start_car* cannot be executed if the car is already started and there is no gas in the tank respectively. Executability condition (5) states that it is impossible to $turn\_off$ if the car is already turned off. Dynamic laws (6) thru (9) capture the effects of *start_car* and $turn\_off$ on auxillary fluents *v_changed* and *time_changed*. State constraints (10) and (11) define *consume_rate* when the car is running and not running respectively. Finally, state constraint (12) defines *gas_volume* by deducting the volume consumed over

the interval $[T_0, T]$ from the existing volume. We can extend this action description to determine whether the car is physically moving as follows. First, we extend the signature to include the new process fluent *moving*. Let *process* sort contain additional functions $f(0,0,T), f(1,1,T), \ldots\ldots$ Given that the volume of gas at time t_0 is x, function $f(x, t_0, T)$ returns true if there is enough gas at time $T \geq t_0$ and false otherwise. We assume that the consumption rate is 5. Here is the definition.

$$f(x, t_0, T) = \begin{cases} true \text{ if } x - 5 * (T - t_0) > 0 \\ false \text{ otherwise} \end{cases}$$

We then add the following state constraints to $\mathcal{A}_3$.

$$moving = \lambda T.f(X, T_0, T) \text{ if } started, \quad v\_changed = X, \quad time\_changed = T_0. \tag{13}$$

$$moving = \lambda T.false \text{ if } \neg started \tag{14}$$

(12) states that if the car has been started then it may or may not move depending upon the volume of gas in the tank. (13) states that the car is not moving if it has not been started.

In the next example we talk about domains that exhibit resilient behavior.

Example 3.1.4 (Default or resilient behavior)**.** Consider an agent acting in a domain consisting of a spring door. The agent is capable of opening and releasing the door. The *default behavior* of the door is to remain closed until it is opened by the agent. Once open, the agent can release it which causes the door to revert back to its default position. As we can see, the agent does not explicitly close the door. Instead, it is the spring action of door that triggers the closing of the door. If we use an action language such as $\mathcal{AL}$ to model this domain the corresponding theory will fail to capture the resilient behavior of this domain. Language H on the other hand allows triggers which can be used to capture resilient and natural behavior. We can use a trigger to specify conditions under which the door closes. To be more realistic we can assume that once

the door is released it closes automatically after 10 seconds. *Releasing* the door can be viewed as an action with delayed effects. Similar to example 3.1.2, releasing the door starts a timer that counts down the *time left* for the door to close. Once the timer reaches zero a *close* action is triggered causing the door to close.

Let $\mathcal{A}_4$ be an action description of H. The corresponding signature $\Sigma(\mathcal{A}_4)$ consists of multi-valued fluent *door*, process fluent $time\_left$, agent actions *open*, *release* and event *auto_close*. Fluent *door* denotes the status of the spring door and ranges over $\{closed, open, closing\}$. Fluent $time\_left$ denotes the time left for the door to close. For simplicity we will assume that it takes 10 seconds for the door to close once it is released. We also have numeric fluent *time_changed* to denote the last time point at which the door was released. The corresponding causal laws are as follows.

$$open \text{ causes } door = open \tag{1}$$

$$\text{impossible } open \text{ if } door = open \tag{2}$$

$$release \text{ causes } door = closing \tag{3}$$

$$\text{impossible } release \text{ if } door = closing \tag{4}$$

$$release \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{5}$$

$$time\_left = \lambda T.max(0, 10 - T + T_0) \quad \text{if } time\_changed = T_0, \; door = closing. \tag{6}$$

$$time\_left = \lambda T.0 \text{ if } door = open \tag{7}$$

$$time\_left = \lambda T.0 \text{ if } door = closed \tag{8}$$

$$time\_left(end) = 0, door = closing \text{ triggers } auto\_close \tag{9}$$

$$auto\_close \text{ causes } door = closed \tag{10}$$

$$\text{impossible } release \text{ if } door = closed \tag{11}$$

Dynamic laws (1) and (3) capture the effects of *open* and *release* on fluent *door* respectively. (2) and (4) are executability conditions for actions *open* and *release* respectively. Dynamic law (5) captures the effect of *release* on *time_changed*. State constraint (6) defines $time\_left$ as a function of time as the door is *closing*. State constraints (7) and (8) state that $time\_left$ is zero when the door is open and closed

respectively. (9) is a trigger which states that any state in which the door is *closing* if $time\_left$ becomes zero then *auto_close* is triggered. (10) captures the effect of *auto_close* on fluent *door*. Finally, (11) is an executability condition for *release*.

A simple extension of this domain is that a latch on the door automatically locks the door once it closes. To model this behavior we introduce a new event called *latch_lock* which is triggered when the door closes. We extend the domain of *door* to include a new value called *locked*. Finally, we extend $\mathcal{A}_4$ by adding causal laws involving *latch_lock*. Therefore, we write

$$door = closed \text{ triggers } latch\_lock \quad (13)$$

$$latch\_lock \text{ causes } door = locked \quad (14)$$

$$\text{impossible } release \text{ if } door = locked \quad (15)$$

(13) is a trigger which says that *latch_lock* is triggered when the door is closed. Dynamic law (14) captures the effect of *latch_lock* on *door*. Finally, (15) is an executability condition for *release*.

Our final example demonstrates that H can model the natural behavior of a bouncing ball.

Example 3.1.5 (Natural behavior)**.** Consider an agent acting in a domain consisting of a ball. The ball is held above the ground by the agent. The actions available to the agent are *drop* and *catch*. Dropping the ball causes the *height* of the ball to change continuously with time as defined by Newton's laws of motion. As the ball accelerates towards the ground it gains velocity. If the ball is not caught before it reaches the ground it hits the ground with velocity v and bounces up into the air with velocity $r * v$ where r is the rebound coefficient. The bouncing ball reaches a certain height and falls back towards the ground due to gravity. Therefore, we have two natural actions *bounce* and $fall$. Let us see how we can use language H to determine the height of the ball as various actions take place.

Let $\mathcal{A}_5$ be an action description of H. The corresponding signature $\Sigma(\mathcal{A}_5)$ consists of fluent *status*, process fluents *height* and *velocity*, agent actions *drop* and *catch* and

natural actions *bounce* and $fall$. Fluent *status* denotes the status of the ball and ranges over $\{descending, ascending, stationary\}$. Σ contains fluent *time_changed* which denotes the last time point at which the status of the ball was changed. We also have fluents *ht_changed* and *v_changed* which denote the height and velocity of the ball at the time denoted by *time_changed* respectively. The corresponding causal laws are as follows.

$$drop \text{ causes } status = descending \tag{1}$$

$$\text{impossible } drop \text{ if } status = descending \tag{2}$$

$$\text{impossible } drop \text{ if } status = ascending \tag{3}$$

$$catch \text{ causes } status = stationary \tag{4}$$

$$\text{impossible } catch \text{ if } status = stationary \tag{5}$$

$$\text{impossible } catch \text{ if } height(end) = 0 \tag{6}$$

$$\text{impossible } drop \text{ if } height(end) = 0 \tag{7}$$

$$drop \text{ causes } ht\_changed = X \text{ if } height(end) = X \tag{8}$$

$$catch \text{ causes } ht\_changed = X \text{ if } height(end) = X \tag{9}$$

$$drop \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{10}$$

$$catch \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \tag{11}$$

$$velocity = \lambda T.9.8 * (T - T_0) \text{ if } \begin{array}{l} status = descending, \\ time\_changed = T_0 \end{array} \tag{12}$$

$$velocity = \lambda T.0 \text{ if } status = stationary \tag{13}$$

$$velocity = \lambda T.max(0, X - 9.8 * (T - T_0)) \text{ if } \begin{array}{l} status = ascending, \\ v\_changed = X, \\ time\_changed = T_0 \end{array} \tag{14}$$

$$status = descending,\ height(end) = 0,\ velocity(end) > 0 \text{ triggers } bounce \quad (15)$$

$$bounce \text{ causes } v\_changed = X * 0.8 \text{ if } velocity(end) = X \quad (16)$$

$$bounce \text{ causes } status = ascending \quad (17)$$

$$bounce \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \quad (18)$$

$$status = ascending,\ velocity(end) = 0,\ height(end) > 0 \text{ triggers } fall \quad (19)$$

$$fall \text{ causes } status = descending \quad (20)$$

$$fall \text{ causes } ht\_changed = X \text{ if } height(end) = X \quad (21)$$

$$fall \text{ causes } time\_changed = T_0 \text{ if } end = T_0 \quad (22)$$

$$height = \lambda T.max(0, X - 4.9 * (T - T_0)^2) \text{ if } ht\_changed = X,\ time\_changed = T_0,\ status = descending \quad (23)$$

$$height = \lambda T.X * (T - T_0) - 4.9 * (T - T_0)^2 \text{ if } v\_changed = X,\ time\_changed = T_0,\ status = ascending \quad (24)$$

$$height = \lambda T.X \text{ if } ht\_changed = X,\ status = stationary \quad (25)$$

Dynamic laws (1),(4),(17) and (20) capture the effects of actions *drop*, *catch*, *bounce* and $fall$ on *status* respectively. Dynamic laws (8),(9) and (21) capture the effects of *drop*, *catch* and $fall$ on *ht_changed* respectively. Dynamic law (16) states that if *bounce* occurs then *v_changed* is 80% of the balls original *velocity*. Dynamic laws (10), (11), (18), and (22) capture the effect of *drop*, *catch*, *bounce* and $fall$ on *time_changed* respectively. Triggers (15) and (19) state the conditions under which actions *bounce* and $fall$ are triggered. State constraints (12), (13) and (14) define *velocity* when the ball is *descending*, *stationary* and *ascending* respectively. Similarly, state constraints (23), (24) and (25) define *height*.

3.2 Methodology

After working with a number of examples we came up with methodology for writing action descriptions of H. Here are some guidelines to writing decent action

descriptions of H.

- Define every process fluent using a state constraint. This will allow us to write simpler and elaboration tolerant action descriptions.
- Do not allow reserved fluents such as *start* and *end* to appear in the heads of dynamic causal laws.
- Currently, there are no restrictions on the type of actions allowed in the triggers. However, an action classified as an agent action should not be part of a trigger.
- Avoid writing action descriptions that contain both a trigger as well as an impossibility condition involving the same action. This is because if the bodies of both laws are satisfied then it implies that the action is triggered and impossible at the same time. We can avoid such situations by combining the impossibility condition with the trigger to obtain a collection of triggers. In this way we end up with only one type of statements and avoid conflicts with other statements. For example, if A is an action description of H containing only the statements

  $$l_1, \ldots, l_n \text{ triggers } e$$

  $$\text{impossible } e \text{ if } k_1, \ldots, k_m$$

  then A' is an equivalent action description of H obtained by combining these laws into a collection of triggers

  $$l_1, \ldots, l_n, \neg k_1 \text{ triggers } e$$
  $$\ldots\ldots\ldots$$
  $$l_1, \ldots, l_n, \neg k_m \text{ triggers } e$$

In our examples we see that triggers are very useful when dealing with natural actions. Are triggers absolutely necessary? The answer to this question is no. For example, it is possible to write an action description that does not contain any triggers to

describe the behavior of a bouncing ball. However, it is quite cumbersome to write such a description. In the absence of triggers the functions associated with the fluents must carry the burden of defining the fluents correctly. We may have to use meta functions that take into account the time intervals during which the ball is falling or ascending and assign the appropriate function. This can get quite complicated. Therefore, for the sake of convenience and simplicity we prefer to use triggers.

CHAPTER IV
IMPLEMENTING H

There are two primary goals of this chapter. First, to understand how to reason about properties of a domain using language *H*. Second, to understand how to automate this process of reasoning using ASP.

We know that an action description of H describes a transition diagram that contains an infinite number of paths. However, given an initial state and a history of events that took place there are only a finite number of paths that are valid. In order to reason about properties of a domain it is enough to identify these valid paths. In the first section of this chapter we will see how to specify history in *H* and identify valid paths.

An action description of H can be viewed as the specification of a domain. We implement it by translating statements of H into ASP rules. Since both languages are declarative the translation is simple and direct. Various tasks of the agent are thus reduced to computing answer sets of the resulting ASP program. Another advantage of using ASP is that it is not difficult to prove the correctness of the translation.

4.1 Specifying history

In addition to the action description, the agents 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 action description is *deterministic* (i.e. for any state-action pair there is atmost one successor state [6]) then there is only one such path. Here is a formal definition.

Definition 4.1.1. Given an action description AD of domain $\mathcal{D}$ and an integer $n > 0$, the *recorded history*, Γ_n, of $\mathcal{D}$ upto moment n is a pair $\langle \mathcal{O}, \mathcal{H} \rangle$ where $\mathcal{O}$ is a collection of statements of the form

$$obs(p, i, t, y)$$

where p is a fluent, $y \in range(p)$, $t \in time$, and integer $i \in [0, n]$ and $\mathcal{H}$ is a collection of statements of the form

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

where a is an elementary action, $t \in time$ and integer $i \in [0, n)$.

Integer i, often referred to as a *step*, denotes the order in which states and actions appear in a trajectory. Intuitively, the statement $hpd(a, i, t)$ means that action a was observed to have happened at time t in step i. The statement $obs(p, i, t, y)$ means that fluent p was observed to have the value y at time t in step i. Observations of the form $obs(p, 0, 0, y)$ define the initial values of fluents.

Definition 4.1.2. Given an action description AD and a recorded history Γ_n of domain $\mathcal{D}$, the pair $\langle AD, \Gamma_n \rangle$ is called a *domain description* of $\mathcal{D}$.

We know that the transition diagram described by an action description consists of a number of paths. However, given a set of observations some of these paths are no longer valid with respect to the observations. The following definition identifies all those paths that are compatible with agent's observations.

Definition 4.1.3. Given a domain description $\langle AD, \Gamma_n \rangle$ and the transition diagram, $TD(AD)$, described by AD, a path

$$\langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n \rangle \in TD(AD)$$

is a *model of* Γ_n if each of the following holds.

1. for every i, $0 \leq i < n$, $a_i = \{a \mid hpd(a, i, t) \in \Gamma_n\}$
2. for every i, $0 \leq i < n$, if $hpd(a, i, t) \in \Gamma_n$ then $s_i \models end = t$

3. for every i, $0 \leq i \leq n$, if $obs(p, i, t, y) \in \Gamma_n$ and

   - p is a process fluent then $\exists \lambda T.f(T) \in process$ such that

     $$s_i \models p = \lambda T.f(T) \wedge \lambda T.f(T)(t) = y$$

   - p is a non-process fluent then

     $$s_i \models p = y$$

In order to understand the above definition let us look at example 2.2.1 about an agent acting in a domain consisting of a brick. Let Γ_1 be a recorded history of this domain consisting of statements

$$obs(holding, 0, 0, false).$$
$$obs(height, 0, 0, 500).$$
$$hpd(catch, 0, 5).$$
$$obs(height, 1, 6, 377.5).$$

According to these statements the agent observes that *holding* is false and *height* is 500 units at *time* 0 in the initial state. At $time = 5$ seconds in the initial state the agent catches the ball. At $time = 6$ seconds in the resulting state the agent observes that the height is 377.5 units.

Now consider the following trajectory $P \in TD(\mathcal{A}_0)$.

$$\langle \{\neg holding, height = \lambda T.500 - 4.9 * T^2, ht\_changed = 500,$$
$$time\_changed = 0, start = 0, end = 5\},$$
$$catch,$$
$$\{holding, height = \lambda T.377.5, ht\_changed = 377.5,$$
$$time\_changed = 5, start = 5, end = 10\}\rangle$$

Upon careful observation it is possible to see that P is indeed a model of Γ_1.

4.2 Translation into a logic program

A domain description written in language H can be viewed as a specification of a dynamically changing system. Given such a specification a user can implement it in several ways. One way to implement a domain description is to translate it into an equivalent logic program such that there is a one-to-one correspondence between models of the logic program and models of the specification. In this way various tasks of an agent are reduced to computing models of logic programs.

In this section we will describe how to translate a domain description written in H into an equivalent logic program. Later on, we claim that this translation is correct. For our purposes we chose the logic programming language $\mathcal{AC}$ which combines answer set programming (ASP) and constraint logic programming (CLP). The combination of non-monotonic logic with numerical constraint solving is quite suitable for H. Here is a brief introduction to $\mathcal{AC}$.

4.2.1 Syntax of $\mathcal{AC}$

$\mathcal{AC}$ is a typed language. Its programs are defined over a *sorted* signature Σ, consisting of *sorts*, and properly typed *predicate symbols*, *function symbols* and *variables*.

By a *sort*, we mean a non-empty countable collection of strings over some fixed alphabet. Strings of a sort S are referred as *object constants* of S. Each variable takes on values of a unique sort. A *term* of Σ is either a constant, a variable, or an expression $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol, and $t_1, \ldots, t_n$ are terms of proper sorts. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of proper sorts. A *literal* is either an atom or its negation.

Sorts of $\mathcal{AC}$ can be partitioned as *regular* and *constraint*. Intitively, a sort is declared to be a constraint sort if it is a large (often numerical) set with primitive constraint relations, e.g., $\leq$.

A function $f : S_1 \times \ldots S_n \rightarrow S$, where $S_1, \ldots, S_n$ are regular sorts and S is a contraint

sort, is called a *bridge function*. We introduce a special predicate symbol *val* where $val(f(t), y)$ holds if y is the value of function symbol f for argument t. For simplicity, we write $f(t) = y$ instead of $val(f(t), y)$. The domain and range of a bridge function f are denoted as $domain(f)$ and $range(f)$ respectively.

The partitioning of sorts induces a natural partition of predicates and literals of AC. *Regular predicates* denote relations among objects of regular sorts; *constraint predicates* denote primitive numerical relations on constraint sorts; predicate *val* is called the *bridge predicate*; all the other predicates are called *defined predicates*.

4.2.1.1 Declarations

$\mathcal{AC}$ uses *declarations* to describe the signature. A regular/constraint sort declaration consists of the keyword `#rsort`/`#csort` followed by a list of sort names, like:

$$\#rsort\ sort\_name_1, \ldots, sort\_name_n$$

or

$$\#csort\ sort\_name_1, \ldots, sort\_name_n$$

For example,

$$\#rsort\ boolean, step.$$

$$\#csort\ time, meters.$$

A predicate delaration is an expression of the form:

$$\#pred\_name(sort\_name_1, \ldots, sort\_name_n)$$

where $pred\_name$ is an n-ary predicate symbol and $sort\_name_1, \ldots, sort\_name_n$ is a list of sort names corresponding to the types of the arguments of $pred\_name$. For example,

$$\#holding(step, boolean).$$

A bridge function declaration is an expression of the form:

$$\#func\_name(sort\_name_1, \ldots, sort\_name_n) : sort\_name$$

where $func\_name$ is an n-ary bridge function symbol, $sort\_name_1, \ldots, sort\_name_n$ is a list of sort names corresponding to the types of the arguments of $func\_name$, and $sort\_name$ is the sort of $func\_name$. For example,

$$\#height(step) : meters.$$

For simplicity, we allow multiple predicate and bridge function declarations in the same line proceeded by a single # symbol.

A rule in $\mathcal{AC}$ is an expression of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$$

where l's are literals of Σ.

A program Π of $\mathcal{AC}$ is a collection of rules and declarations over a signature Σ. Every predicate and sort used in a rule of Π **must** appear in a declaration statement. Rules that contain only regular literals are called *regular rules*. Rules whose head is a defined literal are called *defined rules* and the remaining are called *middle rules*.

4.2.2 Semantics of $\mathcal{AC}$

Terms, literals, and rules are called *ground* if they contain no variables and no symbols for arithmetic functions.

A *bridge assignment* is a mapping from every bridge function symbol f and element $x \in domain(f)$ to a value $y \in range(f)$. Such an assignment will be represented by the set of atoms $f(x) = y$. A set S of ground literals is an *answer set* of an AC program Π if there is a bridge assignment V such that S is an answer set of the ASP program $\Pi \cup V$.

4.2.3 Translation into $\mathcal{AC}$

Let $\langle AD, \Gamma_n \rangle$ be a domain description of H. First, let us see how to translate action description AD into an equivalent $\mathcal{AC}$ program. The corresponding signature $\Sigma(AD)$

contains various sorts including the sort *process* which is a collection of functions of time. An $\mathcal{AC}$ program classifies sorts as regular and constraint. But *process* sort does not fall into either category. However, it is possible to treat the sort $range(process)$, which is the range of all functions from *process*, as a constraint sort. Therefore, in our translation we will be dealing with the sort $range(process)$ instead of *process*.

Clearly, this affects the way we translate atoms such as $p = \lambda T.g(T)$. The corresponding $\mathcal{AC}$ translation would be the defined predicate $p(i, t, y)$ where i is a step, $t \in time$ and $y \in range(process)$ such that $\lambda T.g(T)(t) = y$. We elaborate more on this later. However, this leads to another issue. How do we write the inertia axiom for a process fluent? Intuitively, a process fluent will continue to stay as it is unless it is forced to change. But when it comes to translation we are talking about value of a process fluent at some time t instead of the function associated with it. As a result, it is not possible to say that if a process fluent is defined by a function then it will be defined by the same function unless it is forced to change.

We suggest a solution here. Before translating any action description, A, we will transform it into an equivalent action description A' such that every dynamic causal law of A containing a process fluent in the head is replaced by a dynamic causal law-state constraint pair. All other laws will be left untouched. For example, every dynamic causal law of the form

$$a \text{ causes } p = \lambda T.g(T) \text{ if } body.$$

will be replaced by the pair

$$a \text{ causes } f \text{ if } body.$$
$$p = \lambda T.g(T) \text{ if } f.$$

where $f \in \Sigma(A') \setminus \Sigma(A)$ is an inertial fluent. The result of this transformation is that every process fluent is defined exclusively in terms of other fluents. In other words, process fluents become defined fluents. As a result, there is no need to write inertia axioms for process fluents in our translation.

The transformation we described is pretty simple, straight forward and applicable to any arbitrary action description of H. For the rest this chapter, whenever we say action descriptions of H we mean action descriptions that do not contain process fluents in the heads of dynamic causal laws. We proceed with describing the translation.

Let n be a positive integer and let $sig^n(AD)$ be a set of $\mathcal{AC}$ statements such that

- for every sort $s \in \Sigma(AD)$, $sig^n(AD)$ contains a declaration that classifies it as either regular or constraint.

  $$\#rsort\ s_0.$$

  $$\#csort\ s_1.$$

  Sort s_0 is defined as a regular sort while s_1 is defined as a constraint sort. For example, sort *action* is a regular sort whereas *time* is a constraint sort. We therefore write

  $$\#rsort\ action.$$

  $$\#csort\ time.$$

  The classification of sorts allows the $\mathcal{AC}$ compiler to ground only regular terms. All other terms will be left ungrounded until a constraint solver processes them.

- $sig^n(AD)$ contains definitions for regular sorts. This is accomplished by listing all members of a sort as facts. For example, $sig^n(AD)$ defines members of sort *action* using atoms of the form

  $$action(a).$$

  where $a \in action$. We know that AD describes a transition diagram that contains a number of trajectories. A trajectory is a sequence of transitions of the form $\langle s_i, a_i, s_{i+1} \rangle$. The indices, denoted by i, of a trajectory captures the order in which states and actions appear in the trajectory. The indices (also called steps) of a trajectory are denoted by positive integers. $sig^n(AD)$ captures

these steps in an $\mathcal{AC}$ program by introducing a regular sort *step* and defining it as follows.

$$\#rsort\ step.$$
$$step(0).$$
$$step(1).$$
$$\ldots$$
$$step(n).$$

The superscript n of $sig^n(AD)$ denotes the maximum value of a step. Another advantage of using steps is that it will allow us to talk about values of fluents at various steps.

- $sig^n(AD)$ contains declarations for predicates. $sig^n(AD)$ encodes fluents of $\Sigma(AD)$ as predicates of an $\mathcal{AC}$ program. For every fluent $p : s_1 \times \cdots \times s_m \rightarrow s \in \Sigma(AD)$
  - if s is a regular sort then $sig^n(AD)$ contains the declaration

    $$\#p(s_1, \ldots, s_m, step, s).$$

    where p is a regular predicate symbol.
  - if s is a constraint sort then $sig^n(AD)$ contains the declaration

    $$\#p(s_1, \ldots, s_m, step) : s.$$

    where p is a bridge function symbol.
  - if s is the sort *process* then $sig^n(AD)$ contains the declaration

    $$\#p(s_1, \ldots, s_m, step, time, range(process)).$$

    where p is a defined predicate symbol.

We will ignore the arguments $s_1, \ldots, s_m$ to make our presentation easier.

- $sig^n(AD)$ contains the declaration

$$\#occurs(action, step).$$

  where *occurs* is a regular predicate symbol. This predicate is used to denote actions that appear in the causal laws of AD.

- $sig^n(AD)$ contains the declaration

$$\#start(step) : time, end(step) : time$$

  for bridge functions *start* and *end*. They denote the interval associated with each step.

Since AC allows variables we will use (possibly indexed) I as a variable for step and T as a variable for time.

4.2.4 Domain dependent axioms

In this section we show how causal laws of AD are translated into equivalent $\mathcal{AC}$ rules. To do this we introduce some notation.

Let l be a literal of H and I be a variable ranging over step. By $\alpha(l, I)$ we denote a collection of literals of $\mathcal{AC}$ defined as follows.

- If l is a fluent atom of the form $p = y$
  - and p ranges over a regular sort then $\alpha(p = y, I)$ denotes the $\mathcal{AC}$ atom $p(I, y)$.
  - and p ranges over a constraint sort then $\alpha(p = y, I)$ denotes the $\mathcal{AC}$ atom $p(I) = y$.
  - and p ranges over *process* sort then $\alpha(p = \lambda T.g(T), I)$ denotes the collec-

tion of $\mathcal{AC}$ atoms

$$\begin{aligned} & p(I,T,Y), \\ & start(I) = T_1, \\ & end(I) = T_2, \\ & T_1 \leq T \leq T_2, \\ & T < \omega, \\ & Y = g(T) \end{aligned} \tag{4.1}$$

where Y ranges over $range(process)$.

- If l is $end = y$ then $\alpha(end = y, I)$ denotes the atom $end(I) = y$. Similarly for $start$.
- If l is of the form $p(end) = y$ where p is a $process$ fluent and $y \in range(process)$ then $\alpha(p(end) = y, I)$ denotes the collection of atoms $p(I, t, y), end(I) = t$. Similarly for $p(start) = y$. Also, for any $t \in time$, $\alpha(p(t) = y, I)$ denotes the atom $p(I, t, y)$.
- If l is an arithmetic atom of H then $\alpha(l, I) = l$
- If l is a literal of the form $p \neq y$ then $\alpha(p \neq y, I) = \neg\alpha(p = y, I)$

Notice that if l is an atom involving a process fluent then the cardinality of $\alpha(l, I)$ is normally greater than one. In such cases $\neg\alpha(l, I)$ is obtained by negating the corresponding *defined* predicate. For example, in (4.1) replace $p(I, T, Y)$ by $\neg p(I, T, Y)$ to obtain $\neg\alpha(p = \lambda T.g(T), I)$.

We introduce a function, $\tau(r)$, which transforms every causal law $r \in AD$ into a rule of $\mathcal{AC}$. We proceed with the definition of $\tau(r)$. For every causal law $r \in AD$

- if r is of the form

$$e \text{ causes } p = y \text{ if } l_1, \ldots, l_n$$

then $\tau(r)$ is the rule

$$\begin{aligned} \alpha(p = y, I + 1) :- \ & occurs(e, I), \\ & \alpha(l_1, I), \\ & \ldots, \\ & \alpha(l_n, I). \end{aligned} \tag{4.2}$$

- if r is of the form

$$p = y \text{ if } l_1, \ldots, l_n$$

where p is a non-process fluent then $\tau(r)$ is the rule

$$\alpha(p = y, I) : -\alpha(l_1, I), \ldots, \alpha(l_n, I). \tag{4.3}$$

- if r is of the form

$$p = \lambda T.g(T) \text{ if } l_1, \ldots, l_n$$

where p is a process fluent then $\tau(r)$ is the rule

$$\begin{aligned} p(I, T, Y) :- \ & \alpha(l_1, I), \\ & \ldots\ldots \\ & \alpha(l_n, I), \\ & start(I) = T_1, \\ & end(I) = T_2, \\ & T_1 \leq T \leq T_2, \\ & T < \omega, \\ & Y = g(T). \end{aligned} \tag{4.4}$$

- if r is of the form

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n$$

then $\tau(r)$ is the rule

$$\begin{aligned} :- \ & occurs(e_1, I), \\ & \ldots\ldots, \\ & occurs(e_m, I), \\ & \alpha(l_1, I), \\ & \ldots\ldots, \\ & \alpha(l_n, I). \end{aligned} \tag{4.5}$$

- if r is of the form

$$l_1, \ldots, l_m \text{ triggers } e$$

then $\tau(r)$ is the rule

$$\begin{aligned} triggered(e, I) :- \ & \alpha(l_1, I), \\ & \ldots\ldots, \\ & \alpha(l_m, I), \\ & I < n. \end{aligned} \tag{4.6}$$

where n is the maximum value for steps.

In the above rules every $\alpha(l, I)$ will be replaced by the collection of $\mathcal{AC}$ literals it denotes.

4.2.5 Domain independent axioms

Domain independent axioms are part of every $\mathcal{AC}$ program resulting from translating an action description of H. They define properties shared by every domain. For instance, inertia is a very common property. Let us see how inertia axioms are written in $\mathcal{AC}$.

In language H there is no restriction on the type of fluents allowed. Fluents that obey the principle of inertia are *inertial* fluents. Fluents that are defined explicitly in terms of other fluents and are not direct consequences of actions are *defined* fluents. This implies that defined fluents do not appear in the heads of dynamic causal laws.

Let $\beta(AD)$ be a set of $\mathcal{AC}$ rules such that

- for every inertial (non-process) fluent $p \in \Sigma(AD)$, $\beta(AD)$ contains the rule

$$\begin{aligned} \alpha(p = Y, I + 1) :- \ & \alpha(p = Y, I), \\ & not\ \neg\alpha(p = Y, I + 1). \end{aligned}$$

where variable Y ranges over $range(p)$. If p is a process fluent that is inertial then $\beta(AD)$ contains no inertia axiom for it. This is because all process fluents are defined explicitly using state constraints.

- for every non-process fluent p , $\beta(AD)$ contains the following uniqueness axiom.

$$\begin{aligned} \neg\alpha(p = Y_1, I) :- \ & \alpha(p = Y_2, I), \\ & Y_1 \neq Y_2. \end{aligned}$$

where variables Y_1 and Y_2 range over $range(p)$. Similarly, for every process fluent p $\beta(AD)$ contains

$$\begin{aligned} \neg p(I, T, Y_1) :- \ & p(I, T, Y_2), \\ & Y_1 \neq Y_2. \end{aligned}$$

where Y_1 and Y_2 range over $range(process)$.

- $\beta(AD)$ contains domain independent axioms related to *start* and *end*. The first axiom states that the first step of the trajectory always starts at time 0.

$$start(0) = 0.$$

The second axiom defines the start time for all other steps. According to this axiom if step I ends at time T then the next step will start at time T.

$$start(I + 1) = T : -end(I) = T, I < n.$$

Here n is the maximum value for steps. This axiom captures the fact *if s is a state that ends at time t, then any state s′ following s will start at t*. The next

two axioms ensure that for any step I if T_1 is the start time and T_2 is the end time then the constraint $T_1 \leq T_2 \wedge T_1 < \omega$ is not violated.

$$\begin{aligned} :- \ & start(I) = T_1, \\ & end(I) = T_2, \\ & T_1 > T_2. \end{aligned}$$

$$\begin{aligned} :- \ & start(I) = T_1, \\ & T_1 \geq \omega. \end{aligned}$$

- for every action e that appears in a trigger of AD, $\beta(AD)$ contains

$$\neg triggered(e, I) :- \text{not } triggered(e, I), I < n.$$

$$\begin{aligned} :- \ & triggered(e, I), \\ & \text{not } occurs(e, I), \\ & I < n. \end{aligned}$$

$$\begin{aligned} :- \ & \neg triggered(e, I), \\ & occurs(e, I), \\ & I < n. \end{aligned}$$

where n is the maximum value for steps. The above axioms ensure compliance between triggering conditions and execution of actions. The first axiom is a closed world assumption for *triggered(e,I)*. The second axiom states that it is impossible that e is not executed when it is triggered. The third axiom states that it is impossible that e is executed when it is not triggered.

4.2.6 Translating history

In the previous section we understood how to translate statements of an action description of H into equivalent $\mathcal{AC}$ rules. This translation alone is not enough. A domain description also includes a recorded history that must be translated into equivalent $\mathcal{AC}$ rules.

Let Γ_n be a recorded history upto moment n. Statements of Γ_n are encoded as facts of an $\mathcal{AC}$ program. A statement of the form $obs(p, i, t, y)$ is encoded as the fact

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

And a statement of the form $hpd(a, i, t)$ is encoded as the fact

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

Given a set of observations we need to ensure that the agent's predictions match with his observations. The *reality check* axiom guarantees this. Let $R(AD)$ be a set of rules such that

- for every fluent $p \in \Sigma(AD)$
  - if p is a non-process fluent then $R(AD)$ contains

$$\begin{aligned} :- \ & obs(p, I, T, Y), \\ & \neg\alpha(p = Y, I). \end{aligned}$$

  - if p is a process fluent then $R(AD)$ contains

$$\begin{aligned} :- \ & obs(p, I, T, Y), \\ & \neg p(I, T, Y). \end{aligned}$$

- $R(AD)$ contains the rules

$$\begin{aligned} & occurs(A, I) :- hpd(A, I, T), I < n. \\ & end(I) = T :- hpd(A, I, T), I < n. \end{aligned} \tag{4.7}$$

  where n is the maximum value for steps. The rules state that if action A was observed to have happened at time T of step I then A must have occurred in step I and I must have ended at time T respectively.

- for every fluent $p \in \Sigma(AD)$,

– if p is a non-process fluent then $R(AD)$ contains the rule

$$\alpha(p = Y, 0) : -obs(p, 0, 0, Y).$$

which defines the initial value of p.

– if p is a process fluent then $R(AD)$ contains the rule

$$p(0, 0, Y) : -obs(p, 0, 0, Y).$$

Given an action description AD and recorded history Γ_n, by $\Delta_{AD}(\Gamma_n)$ we denote the $\mathcal{AC}$ program

$$\Delta_{AD}(\Gamma_n) = \Gamma_n \cup R(AD)$$

.

By $\Pi^n(AD)$ we denote an $\mathcal{AC}$ program containing a description of the signature of AD, translations of statements of AD and other domain independent axioms. The n denotes the maximum value for steps. Therefore,

$$\Pi^n(AD) = sig^n(AD) \cup \bigcup_{r \in AD} \tau(r) \cup \beta(AD) \tag{4.8}$$

where $sig^n(AD)$, $\tau(r)$ and $\beta(AD)$ are as defined in the previous sections.

Using the above definitions we can define the translation of both AD and Γ_n into $\mathcal{AC}$. Given a domain description $\langle AD, \Gamma_n \rangle$, the translation of $\langle AD, \Gamma_n \rangle$ into $\mathcal{AC}$, denoted by $\Pi^n(AD, \Gamma_n)$, is a collection of $\mathcal{AC}$ rules such that

$$\Pi^n(AD, \Gamma_n) = \Pi^n(AD) \cup \Delta_{AD}(\Gamma_n) \cup \{end(n) = t_n\} \tag{4.9}$$

where $t_n \geq start(n)$.

4.2.7 Correctness

Given a domain description $\langle AD, \Gamma_n \rangle$ how do we know that the corresponding translation $\Pi^n(AD, \Gamma_n)$ is a correct translation? One way to prove correctness is to

establish a one-to-one relation between answer sets of $\Pi^n(AD, \Gamma_n)$ and models of Γ_n. Here are some definitions that will be useful to establish this relationship.

Given a program Π of $\mathcal{AC}$, by $lit(\Pi)$ we denote the set of all ground literals of Π.

Definition 4.2.1. Let $\langle AD, \Gamma_n \rangle$ be a domain description of H and $A \subseteq lit(\Pi^n(AD, \Gamma_n))$. We say that A *defines the sequence* $\langle s_0, a_0, s_1, \ldots, a_{n1}, s_n \rangle$ if each of the following holds.

- for every i, $0 \leq i \leq n$, $s_i = \{l \mid l \text{ is literal of H } \wedge \alpha(l, i) \subseteq A\}$
- for every i, $0 \leq i < n$, $a_i = \{e \mid occurs(e, i) \in A\}$.

Γ_n is *complete* if and only if

- for every fluent $p \in \Sigma(AD)$, Γ_n contains $obs(p, 0, 0, y)$ and
- for every i, $0 \leq i < n$, Γ_n contains $hpd(a, i, t)$ where $a \in action$ and $t \in time$.

Here is a theorem that establishes relationship between a domain description of H and its corresponding translation into $\mathcal{AC}$.

Theorem 4.1. Given a domain description $\langle AD, \Gamma_n \rangle$; if Γ_n is complete then M is a model of Γ_n iff M is defined by some answer set of $\Pi^n(AD, \Gamma_n)$.

A proof of this theorem is available in chapter 7.

CHAPTER V
EXISTING SYSTEMS

In this chapter we will look at existing systems that can be used for implementing action descriptions of H. As mentioned before we implement a theory of H by translating it into a logic program. The corresponding logic program should have two important qualities - ability to represent knowledge with ease and ability to perform numerical computations. In the past few years some researchers [2, 34] have focused on integrating answer set programming(ASP) and Constraint logic programming(CLP) to bring these two qualities together. They came up with new systems that combine ASP and CLP reasoning techniques and were able to achieve significant improvement in performance over existing ASP solvers. In this thesis we use two such systems for implementing action descriptions of H namely *EZCSP* [1] and *ACsolver*. In the next few sections we will understand the advantages and limitations of each system. A third system called *Clingcon* [2] is also available but the underlying constraint solver only deals with finite domains which is a limitation when dealing with continuous functions.

5.1 EZCSP

In [2] the author describes an approach for integrating ASP and constraint programming in which ASP is viewed as a specification language for constraint satisfaction problems. The ASP programs are written in such a way that their answer sets encode the desired constraint satisfaction problems (CSPs). The solution to the CSPs are then computed using constraint satisfaction techniques. The *extended answer set* of such a program, Π, is a pair $\langle A, \alpha \rangle$ such that A is an answer set of Π and α is a solution to the CSP defined by A.

EZCSP is an inference engine for computing extending answer sets of ASP programs

[1] http://marcy.cjb.net/ezcsp/index.html
[2] http://www.cs.uni-potsdam.de/clingcon/

as defined in [2]. Unlike other approaches for integrating ASP and CLP, this approach is a loose coupling of ASP and CLP. For this reason there is no need to modify the underlying ASP and CLP solvers so that they can work together. There is no commitment to a particular type of solver and this allows programmers to select a solver that best fits their needs. The current version of EZCSP uses gringo+clasp [3] by default as ASP solver and SICSTUS Prolog as constraint solver. It also allows the use of ASP solvers such as lparse+smodels [4].

CSPs are encoded in ASP using the following three types of statements.

- A constraint domain declaration is a statement of the form:

  $$cspdomain(\mathcal{D})$$

  where $\mathcal{D}$ is a constraint domain such as *fd*, *r* or *q*. By *fd* we mean finite domain, *r* implies real numbers and *q* implies rational numbers. This statement specifies the constraint domain of a CSP.

- A constraint variable declaration is a statement of the form:

  $$cspvar(x, l, u)$$

  where x is a ground term denoting a CSP variable and l and u are numbers from the constraint domain. The statement says that the domain of x is $[l, u]$.

- A constraint statement is a statement of the form:

  $$required(\gamma)$$

  where γ is a constraint involving variables specified by the *cspvar* statements. Intuitively, the statement says that γ must be satisfied by any solution to the CSP.

[3] http://potassco.sourceforge.net/

[4] http://www.tcs.hut.fi/Software/smodels/

When encoding action descriptions of H we need to realize that an action description of H describes a transition diagram that contains a number of trajectories. A trajectory is a sequence of transitions of the form $\langle s_i, a_i, s_{i+1} \rangle$. The indices (also called steps) of a trajectory are denoted by positive integers. An EZCSP encoding must treat these steps as regular ASP predicates.

Now we will see how fluents and atoms of H are encoded into EZCSP. A fluent ranging over a large domain is encoded as a CSP variable with *step* as one of its argument. For example, the fluent *ht_changed* from example 2.2.1 will be encoded as the CSP variable $ht\_changed(s)$ where s is a step. The corresponding atom $ht\_changed = 20$ will be encoded as $required(ht\_changed(s) == 20)$. Atoms containing fluents that range over small domains will be encoded as regular ASP predicates. For example, if *color* is a fluent that ranges over $\{yellow, green\}$ then the atom $color = yellow$ will be encoded as $color(s, yellow)$.

Coming to process fluents, however, we encounter the same issues we encountered with language $\mathcal{AC}$. The sort *process* is neither regular nor constraint. For this reason, instead of dealing with this sort, we work with $range(process)$ which is the range of all functions from *process*. In $\mathcal{AC}$ we translated atoms involving process fluents as defined predicates. In EZCSP, however, there is no counter part for defined fluents. For this reason we chose to map a process fluent into a numerical fluent such that the value of this numerical fluent in step s is the value of the process fluent at the end of s. We will encode this numerical fluent as a CSP variable with step as one of its argument. For example, the process fluent *height* from example 2.2.1 is mapped into the numerical fluent h which will be encoded as the CSP variable $h(s)$.

One of the limitations of EZCSP is that it is not possible to use the solutions of CSPs to make new inferences on the ASP side. Unlike *ACsolver*, an inference engine for $\mathcal{AC}$ programs [34], the ASP solver does not receive any feedback from the constraint solver. For this reason, statements of H which contain constraints in the body cannot be encoded directly into EZCSP. In some situations we were able to come up with

alternative ways to encode such statements and in some situations we were unable to do so. For example, the executability condition from example 2.2.1

$$\text{impossible } \mathit{catch} \text{ if } \mathit{height}(\mathit{end}) = 0$$

can be encoded into the EZCSP rule

$$\mathit{required}(\mathit{height}(I) > 0) : -\mathit{occurs}(\mathit{catch}, I).$$

which says that if action *catch* takes places in step I then the *height* at the end of I should be greater than zero. The rule captures the intuition of the executability condition. For all other situations in which it not possible to come up with alternative ways we introduce more rules and constructs. For example, to encode the statement

$$p \text{ if } x < y$$

in EZCSP the author of EZCSP suggests that we write

$$\begin{aligned} 1\{\mathit{req}(\mathit{lt}), \mathit{req}(\mathit{geq})\}1. \\ \mathit{required}(x < y) : -\mathit{req}(\mathit{lt}). \\ \mathit{required}(x >= y) : -\mathit{req}(\mathit{geq}). \\ p : -\mathit{required}(x < y). \end{aligned}$$

The author also states that from a computational perspective, having more than a couple of these translations in a program may lead to a poor performance because each combination of *required* atoms results in a separate call to the constraint solver.

We know that constraint variables appear only in *required* predicates. So when we write an inertia axiom for a constraint variable we end up adding a required predicate to the body of a rule. However, the current version of the solver does not support *required* predicates in the bodies of rules. To overcome this problem, we replace required predicates in the bodies with auxillary ASP predicates. Later we add rules that

define these auxillary predicates. For example, the inertia axiom for the constraint variable $ht\_changed(I)$ will be written as follows.

$$required(ht\_changed(I+1) == ht\_changed(I)) : -not\ ab(I), step(I).$$
$$ab(I) : -occurs(drop, I), step(I).$$
$$ab(I) : -occurs(catch, I), step(I).$$

As we can see, every action that changes the value of $ht\_changed(I)$ must be taken into account to define the inertia axiom. In spite of this inconvenience we are able to find a solution to the frame problem.

The performance of the system is pretty good. The system is quite reliable and has an easy syntax. There is a clear separation between constraint variables and regular predicates because constraint variables can appear only in *required* predicates. For a program containing approximately 600 rules and 50 variables, the system returns an answer set in less than 1.5 seconds and returns all answer sets within 2.5 seconds.

A sample EZCSP code that encodes the bouncing ball example is available in appendix A. We gave an initial height for the ball and a sequence of drop-catch actions along with their times of occurrence. The answer set of the program consists of atoms denoting the height of the ball at various time points and the times at which the ball bounces and falls back to the ground.

Solving a planning problem in H is similar to solving a planning problem in $\mathcal{AL}$ with the exception that in addition to generating plans (sequences of actions), we also schedule these actions in order to satisfy the goal of an agent. For example, a sample planning problem and a corresponding plan is shown below.

Initial state : Agent holds ball at 100m
Goal state: Agent holds ball at 50m
plan: drop, catch
Schedule : drop at 5, catch at 8.194

We used EZCSP to solve such planning problems. However, further investigation is needed before we can comment on the performance of EZCSP w.r.t planning problems.

5.2 Luna

Luna is an inference engine for computing answer sets of $\mathcal{AC}$ programs [37]. Unlike EZCSP, this system is a tight coupling of an ASP solver and a constraint solver. Current implementation uses the constraint solver *clp(r)* and the ASP solver *Surya* which is similar to Smodels. The implementation requires modifying both solvers so that they can work together.

Luna was built over the prototype solver $\mathcal{AC}solver$ [34]. The main differences between *Luna* and $\mathcal{AC}solver$ are the organization of the main search function and some incrementality implemented by Luna. During the computation of answer sets of an $\mathcal{AC}$ program, *Luna* stores some intermediate results that are reused if necessary, while $\mathcal{AC}solver$ repeats the computation of these results. These differences result in improved efficiency of *Luna*.

We introduced language $\mathcal{AC}$ in chapter 4. However, the current implementation of *Luna* is based on an earlier version [36] of $\mathcal{AC}$. Due to several implementation issues there are some restrictions on the language of the solver. For this reason the language is a strict subset of $\mathcal{AC}$. In spite of this the solver has been able to solve problems that traditional ASP solvers are unable to solve.

There are some similarities between Luna and EZCSP when it comes to encoding atoms of H. Atoms containing fluents ranging over small domains are encoded as regular ASP predicates. Atoms containing fluents ranging over large domains are encoded as bridge predicates (previously called mixed). As explained in chapter 4, atoms containing process fluents can be encoded as defined predicates. However, due to limitations of the current version of *Luna* we are unable to do so. One limitation is that bridge predicates are not allowed in the bodies of defined rules. This is vital to defining process fluents. Alternatively, we can encode process fluents as bridge functions of $\mathcal{AC}$ such that the value of the bridge function in step s is the value of the process fluent at the end of s. This is very similar to what we do in EZCSP. In fact, in [2] the author establishes a relationship between a subclass of $\mathcal{AC}$ programs

and EZCSP programs.

As mentioned earlier *Luna* allows feedback from the constraint solver to make new inferences on the ASP side. Statements of H that contain constraints in the body can be encoded as middle rules of $\mathcal{AC}$. For example, the statement from example 2.2.1

$$\text{impossible } catch \text{ if } height(end) = 0$$

can be encoded as the $\mathcal{AC}$ rule

$$\begin{aligned} :- \ & occurs(catch, I), \\ & height(I, X), \\ & X == 0. \end{aligned}$$

where $height(I, X)$ is a bridge predicate which says that the height at the end of step I is X. In general, any statement of the form

$$p \text{ if } x < y$$

can be encoded as the $\mathcal{AC}$ rule

$$\begin{aligned} p :- \ & value\_of(x, V_1), \\ & value\_of(y, V_2), \\ & V_1 < V_2. \\ & \\ & :- \text{not } p. \end{aligned}$$

where $value\_of$ is a bridge function that determines the values of variables x and y.

In the current version of *Luna* the *not* operator cannot be applied to bridge predicates and defined predicates. For this reason when we write defaults, such as inertia, involving these predicates we introduce auxillary predicates. For example, atoms containing the fluent *ht_changed* from example 2.2.1 will be encoded as the bridge predicate $ht\_changed(I, X)$ where I is a variable for step and X ranges over meters.

The corresponding inertia axiom is written as follows.

$$
\begin{aligned}
:- \; & ht\_changed(I, X), \\
& next(I, I1), \\
& ht\_changed(I1, X1), \\
& \neg ab(I), \\
& X! = X1.
\end{aligned}
$$

$$
\begin{aligned}
& ab(I) :- occurs(drop, I). \\
& ab(I) :- occurs(catch, I). \\
& \neg ab(I) :- not\ ab(I).
\end{aligned}
$$

This is very similar to what we do in EZCSP for solving the frame problem.

A *Luna* encoding of the brick drop example from chapter 2 is available in appendix B. The encoding is provided with an initial situation and a sequence of actions and the answer set returned by *Luna* contains atoms denoting the predicted values of fluents. We were also able to encode small size planning problems and compute answer sets in a reasonable amount of time. We hope that further testing will help improve the performance of the solver.

Current implementation of *Luna* suffers from runtime errors which arise due to coupling issues between solvers. We compared the performance of *Luna* against EZCSP and found that EZCSP performs better in many cases. We also compared the performance of *Luna* against $\mathcal{AC}solver$ and found that *Luna* outperforms $\mathcal{AC}solver$ [37].

CHAPTER VI
RELATED WORK

In this section we will compare language H with other formalisms that are used for modeling real-time systems. We begin with *timed automata* [1]- a formalism that came up in the early 1990s to model the behavior of real-time systems over time. Later on we will compare H with situation calculus.

6.1 Timed Automata

During the 1980s a number of studies were dedicated towards using automata theory for specification and verification of systems. These studies ignore time and focus on sequences of events, also called *event traces*. A set of event sequences characterizes the behavior of a system. Since a set of sequences is a formal language, it is possible to use automata for the specification and verification of systems [1]. When the systems are finite-state we can use finite automata which accepts strings of finite length. However, if we want to model systems that could possibly run forever we need ω-automata which is a finite automata with the acceptance condition modified suitably to accept strings of infinite length.

When dealing with real-time systems, however, we cannot abstract away from continuous time. Verification of such systems involves reasoning about functions that change continuously with time. The behavior of such a system is characterized not only by sequences of events but also by sequences of times which denote the times at which events take place. As a result, we now have timed traces which are obtained by pairing event traces with sequences of times. If we want to define a language of timed traces using finite automata it is difficult because it is not obvious how to transform a timed traced into an ordinary formal language. For this reason Alur and Dill developed a theory of *timed languages* and *timed automata* [1] to specify and verify real-time systems. In the next few paragraphs we will give a formal introduction to timed automata. The definitions in the next few paragraphs have been obtained from

[1].

A *time sequence* $\tau = \tau_1\tau_2...$ is an infinite sequence of time values $\tau_i \in R$ with $\tau_i > 0$, such that τ increases strictly monotonically i.e. $\tau_i < \tau_{i+1}$ for all $i \geq 1$ and for every $t \in R$, there is some $i \geq 1$ such that $\tau_i > t$. For any time sequence τ, $\tau_0 = 0$.

A *timed word* over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1\sigma_2 \ldots$ is an infinite word over Σ and τ is a time sequence. A *timed language* over Σ is a set of timed words over Σ.

A *timed transition table* is capable of defining a timed language. In this table, a transition depends upon the input symbol as well as the time of the input symbol relative to the times of previously read symbols. For this reason, a finite set of (real valued) clocks are associated with each table. The set of clocks can be viewed as set of stop-watches that can be reset and checked independently of one another, but all of them refer to the same clock. A clock constraint is associated with each transition and only when the current clock values satisfy this constraint will a transition be taken.

Given a set X of clock variables, $\Phi(X)$ denotes the set of all clock constraints δ such that

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where $x \in X$ and $c \in \mathcal{R}$. Constraints such as $true$, $x = c$, $x \in [2, 5)$ are considered abbreviations.

A *clock interpretation* v for a set X of clocks is a mapping from X to R. Clock interpretation v for X *satisfies* a clock constraint δ iff δ evaluates to true using the value given by v.

To proceed with the defintion of timed transition table we introduce some notation. For $t \in R$, $v + t$ denotes the clock interpretation which maps every clock x to the value $v(x)+t$. For any $Y \subseteq X$, $[Y \rightarrow t]v$ denotes the clock interpretation for X which assigns t to each $x \in Y$, and agrees with v over the rest of the clocks.

A timed transition table $\mathcal{T}$ is a $\langle \Sigma, S, S_0, C, E \rangle$, where

- Σ is a finite alphabet
- S is a finite set of states
- $S_0 \subseteq S$ is a set of start states
- C is a finite set of clocks
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ gives the set of transitions.

An edge $\langle s, s', a, \lambda, \delta \rangle$ represents transition from state s to s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset and δ is a clock constraint over C.

Given a timed word the behavior of the transition table is captured by defining runs. A *run* records the state and the values of all the clocks at the transition points.

A run r, denoted by $\langle \bar{s}, \bar{v} \rangle$, of a timed transition table $\langle \Sigma, S, S_0, C, E \rangle$ over a timed word (σ, τ) is an infinite sequence of the form

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \ldots$$

with $s_i \in S$ and $v_i \in [C \rightarrow R]$, for all $i \geq 0$, satisfying the requirements

- *Initiation*: $s_0 \in S_0$, and $v_0(x) = 0$ for all $x \in C$.
- *Consecution*: for all $i \geq 1$, there is an edge in E of the form $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle$ such that $v_{i-1} + \tau_i - \tau_{i-1}$ satisfies δ_i and v_i equals $[\lambda_i \rightarrow 0](v_{i-1} + \tau_i - \tau_{i-1})$.

Consider the following timed transition table from [1] with two clocks x and y over alphabet $\{a, b, c, d\}$.

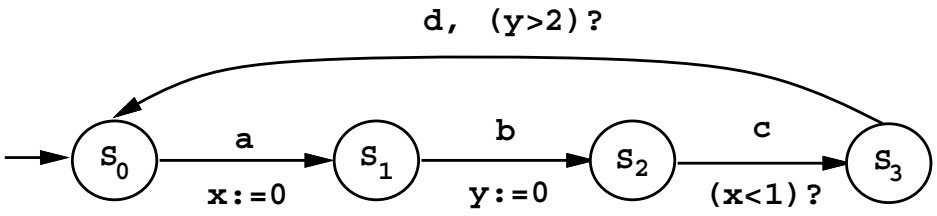

Figure 6.1: Timed transition table with 2 clocks

The timed transition table of figure 6.1 accepts the language

$$L = \{((abcd)^{\omega}, \tau) \mid \forall j.((\tau_{4j+3} < \tau_{4j+1} + 1) \wedge (\tau_{4j+4} > \tau_{4j+2} + 2))\}.$$

Now consider the following timed word.

$$(a, 2), (b, 2.7), (c, 2.8), (d, 5), \ldots, \ldots,$$

An initial segment of the run over this word is as follows.

$$\langle s_0, [0, 0]\rangle \xrightarrow[2]{a} \langle s_1, [0, 2]\rangle \xrightarrow[2.7]{b} \langle s_2, [0.7, 0]\rangle \xrightarrow[2.8]{c} \langle s_3, [0.8, 0.1]\rangle \xrightarrow[5]{d} \langle s_0, [3, 2.3]\rangle$$

Notice that the clock interpretation is represented by the pair $[x, y]$.

A *timed Buchi automaton* (TBA) is a tuple $\langle \Sigma, S, S_0, C, E, F\rangle$, where $\langle \Sigma, S, S_0, C, E\rangle$ is a timed transition table and $F \subseteq S$ is set of accepting states. A run $r = (\bar{s}, \bar{v})$ of a TBA over timed word (σ, τ) is called an accepting run iff $inf(r) \cap F \neq \emptyset$. The language $L(A)$ of timed words accepted by A is the set

$$\{(\sigma, \tau) \mid \text{A has an accepting run over } (\sigma, \tau)\}$$

The class of timed languages accepted by TBA are called *timed regular languages*.

A timed transition table $\langle \Sigma, S, S_0, C, E\rangle$ is called *deterministic* iff

1. it has only one start state, $\mid S_0 \mid = 1$,and

2. for all $s \in S$, for all $a \in \Sigma$, for every pair of edges of the form $\langle s, -, a, -, \delta_1 \rangle$ and $\langle s, -, a, -, \delta_2 \rangle$, the clock constraints δ_1 and δ_2 are mutually exclusive (i.e. $\delta_1 \wedge \delta_2$ is unsatisfiable).

A timed automaton is deterministic iff its timed transition table is deterministic.

6.1.1 Relationship with H

In this section we define the relationship between timed automata and language H. As we can see both formalisms are used for modeling real-time systems. Timed automata captures timing delays between actions and periodic behavior by introducing clock variables. Language H is capable of modeling such behavior by introducing process fluents that keep ticking with time just like a clock. However, unlike clock variables, process fluents can be arbitrary functions of time and hence more general.

A state of a transition diagram described by an action description of H is a collection of atoms of H. However, a state of a timed automata is just a symbol. In language H, we have executability conditions and triggers which allow us to express conditions under which an action is impossible and triggered respectively. This is very useful because properties of a domain can change over time. Timed automata, on the other hand, describes what event sequences are valid but does not contain any constructs that express when an action is impossible or triggered. For example, in the bouncing ball example 3.1.5 from chapter 3 we used triggers to specify conditions under which bounce takes place. We cannot model such behavior using timed automata for two reasons. First, we do have such triggers and second, there is restriction on the type of variables and constraints allowed. Timed automata allows only clock variables and supports only a few types of constraints on these variables. Because of these limitations it is not possible to determine the bouncing times which form a decreasing geometric sequence [43]. If we assume that the ball keeps bouncing forever at equal intervals of time then timed automata can be used. But this does not happen in reality.

Language H also contains state constraints that capture indirect effects of actions. There are no constructs for that purpose in timed automata. Also, only elementary actions are allowed in the transitions of a timed automata. There are no such restrictions in language H. Hence we conclude that language H is more general and expressive than timed automata.

Our claim is that every deterministic timed transition table can be mapped into an equivalent action description of H. In the next few paragraphs we will see how this can be accomplished.

Let $\mathcal{T}\langle\Sigma, S, S_0, C, E\rangle$ be a deterministic timed transition table. By $\mathcal{M}(\mathcal{T})$ we denote an action description of H whose signature, $\psi(\mathcal{M}(\mathcal{T}))$, is described below.

- $\psi(\mathcal{M}(\mathcal{T}))$ contains a sort *action* defined as follows.

$$action = \{e \mid e \in \Sigma\}$$

- $\psi(\mathcal{M}(\mathcal{T}))$ contains fluent *state* ranging over S
- For every $x \in C$
  - $\psi(\mathcal{M}(\mathcal{T}))$ contains process fluent x
  - $\psi(\mathcal{M}(\mathcal{T}))$ contains fluent $cur(x)$ denoting the reset value of x
  - $\psi(\mathcal{M}(\mathcal{T}))$ contains fluent $time\_reset(x)$ denoting the time at which x was last reset.

Next we will translate transitions of T into equivalent causal laws of H. In order to do this we introduce some notation. Given a clock constraint δ over C, by $\mathcal{M}(\delta)$ we denote the constraint obtained by replacing every occurrence of x in δ by $x(end)$. If δ is of the form $\delta_1 \wedge \delta_2$ then $\mathcal{M}(\delta)$ is the collection of atoms $\mathcal{M}(\delta_1), \mathcal{M}(\delta_2)$.

We assume that every transition of the form $\langle s, s', a, \lambda, \delta_1 \vee \delta_2\rangle \in E$ is reduced to two transitions $\langle s, s', a, \lambda, \delta_1\rangle$ and $\langle s, s', a, \lambda, \delta_2\rangle$ respectively.

For every transition $\langle s, s', a, \lambda, \delta\rangle \in E$

- $\mathcal{M}(\mathcal{T})$ contains dynamic law

$$a \text{ causes } state = s' \text{ if } \begin{array}{l} state = s, \\ \mathcal{M}(\delta). \end{array}$$

- For every $x = c \in \lambda$, $\mathcal{M}(\mathcal{T})$ contains dynamic laws

$$a \text{ causes } cur(x) = c \text{ if } \begin{array}{l} state = s, \\ \mathcal{M}(\delta). \end{array}$$

$$a \text{ causes } time\_reset(x) = t \text{ if } \begin{array}{l} end = t, \\ state = s, \\ \mathcal{M}(\delta). \end{array}$$

- For every $x \in C$, $\mathcal{M}(\mathcal{T})$ contains state constraints

$$x = \lambda T.(T - T_0) + Z \text{ if } \begin{array}{l} cur(x) = Z, \\ time\_reset(x) = T_0. \end{array}$$

Given a timed transition table, $\mathcal{T}$, we would like to establish conditions under which the corresponding translation, $\mathcal{M}(\mathcal{T})$, is correct. In order to do this we introduce some definitions.

Let $\mathcal{T}\langle\Sigma, S, S_0, C, E\rangle$ be a deterministic timed transition table and (σ, τ) be a timed word of the form $(\sigma_1, \tau_1), (\sigma_2, \tau_2), \ldots, \ldots$ and so on.

Given a state $s_0 \in S_0$ and clock interpretation v_0 such that for every $x \in C$, $v_0(x) = 0$ we say that a set s_0' of atoms of H is *compatible with the pair* $\langle s_0, v_0\rangle$ iff

1. for every $x \in C$, $s_0' \models \{cur(x) = 0, time\_reset(x) = 0\}$
2. $s_0' \models \{state = s_0, start = 0\}$ and $s_0' \models end = t$ such that $t > 0$
3. for every $x \in C$, $s_0' \models x = \lambda T.(T - t_0) + c$ such that $s_0' \models \{cur(x) = c, time\_reset(x) = t_0\}$

4. for every $x \in C$, $s'_0 \models x(start) = c$ iff $v_0(x) = c$

Given two sets s'_{i-1} and s'_i of atoms of H ($i \geq 1$), we say that the tuple $\langle s'_{i-1}, \sigma_i, s'_i \rangle$ is *compatible with the segment*

$$\langle s_{i-1}, v_{i-1} \rangle \xrightarrow[\tau_i]{\sigma_i} \langle s_i, v_i \rangle$$

w.r.t transition $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle \in E$ iff

1. $s'_{i-1} \models state = s_{i-1}$ and $s'_{i-1} \models \{start = \tau_{i-1}, end = \tau_i\}$
2. $s'_{i-1} \models \mathcal{M}(delta_i)$
3. for every $x \in C$, $s'_{i-1} \models x = \lambda T.(T - t_0) + c$ such that $s'_{i-1} \models \{cur(x) = c, time\_reset(x) = t_0\}$
4. for every $x \in C$, $s'_{i-1} \models x(start) = c$ iff $v_{i-1}(x) = c$
5. for every $x = c \in \lambda_i$, $s'_i \models \{cur(x) = c, time\_reset(x) = \tau_i\}$
6. $s'_i \models \{state = s_i, start = \tau_i\}$ and $s'_i \models end = t$ such that $t > \tau_i$.
7. for every $x \in C$, $s'_i \models x = \lambda T.(T - t_0) + c$ such that $s'_i \models \{cur(x) = c, time\_reset(x) = t_0\}$
8. for every $x \in C$, $s'_i \models x(start) = c$ iff $v_i(x) = c$.

A sequence $p : \langle s'_0, \sigma_1, s'_1, \ldots, \ldots \rangle$ is *compatible with*

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \ldots$$

iff

- s'_0 is compatible with the pair $\langle s_0, v_0 \rangle$
- for $i \geq 1$, every tuple $\langle s'_{i-1}, \sigma_i, s'_i \rangle \in p$ is compatible with a segment of r

  $$\langle s_{i-1}, v_{i-1} \rangle \xrightarrow[\tau_i]{\sigma_i} \langle s_i, v_i \rangle$$

  w.r.t transition $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle$.

Theorem 6.1. Given a deterministic timed transition table $\mathcal{T}\langle\Sigma, S, S_0, C, E\rangle$,

$$r : \langle s_0, v_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1\rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2\rangle \xrightarrow[\tau_3]{\sigma_3} \dots$$

is a run of $\mathcal{T}$ over timed word (σ, τ) iff $\langle s'_0, \sigma_1, s'_1, \dots, \dots\rangle$ is a path of $TD(\mathcal{M}(\mathcal{T}))$ such that $\langle s'_0, \sigma_1, s'_1, \dots, \dots\rangle$ is compatible with r.

A proof of this theorem is available in chapter 7.

6.2 Situation Calculus

Situation calculus was introduced by John McCarthy in 1963 as a way of logically specifying dynamic systems in artificial intelligence. It is a second-order language designed for representing actions and their effects. The tasks of simulation, control and analysis of a dynamic system are reduced to logical entailment.

When it was first introduced it was used for investigating the frame problem. Later on, Ray Reiter [39] and his research group extended the language to incorporate features such as time, concurrency, probability etc. In the next few paragraphs we will see how this approach is different from our approach based on language H.

There are several fundamental differences between the situation calculus (sitcalc) and language H. A specification of a system in sitcalc consists of a collection of logical formulas. These formulas along with the foundational axioms logically entail what is true in the system. Our approach is based on action languages in which a dynamic system is represented by a transition diagram whose nodes represent possible states of the world and whose arcs are labeled by actions. A specification written in H describes a transition diagram. As explained in chapter 2, a state is often identified with the collection of literals it entails.

Sitcalc uses the term *situation* to capture a finite sequence of actions. The initial situation denotes an empty sequence of actions. There is a binary function symbol *do* such that $do(a, s)$ denotes the successor situation to s resulting from performing the action a. There is a major difference between a situation and a state. A state of

H is a snapshot of the world over an interval. Two states are the same if they map all symbols to the same value. 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.

The language of sitcalc allows quantifiers, connectives and logical symbols, thereby, making it powerful and expressive. However, classical logic has its own limitations when it comes to representing knowledge. First of all it is monotonic in nature which is the inability to withdraw conclusions in the presence of new information. Second there are theoretical and practical difficulties when dealing with state constraints in sitcalc. This is because in sitcalc state constraints are of the form $a \supset b$. We know that this classical formula is equivalent to its contrapositive $\neg b \supset \neg a$. But in the theory of action and change [32], a state constraint of the form $a \rightarrow b$ expresses the causal dependency of b on a and $\neg b \rightarrow \neg a$ is not necessarily true. The arrow, $\rightarrow$, captures a form of directionality which can not be expressed in classical logic in a straight forward way [29]. So when we use state constraints in sitcalc it is possible to get unintuitive results. Hence, researchers in this area do not encourage the use of state constraints. On the other hand, there are no problems with using state constraints in H.

In both sitcalc and language H, properties of a domain are represented by fluents. However, there are some differences. Language H contains a process sort which is a collection of functions of time. Process fluents are assigned functions from this sort. In sitcalc all fluents are either boolean or real. For properties that change continuously with time, the value of the corresponding fluent is its value at the start of a situation. For example, in sitcalc the height of a falling ball is denoted by the fluent $height(s)$ where s is a situation. By $height(s) = 300$ we mean that height is 300m at the start of s. In H, $height$ is a function of time.

In both approaches actions are instantaneous. However, in language H because of instantaneous effects it is possible that fluents are not uniquely defined at time points

shared between states. In figure 2.1 from chapter 2 it is possible to see that when the brick is dropped at time 5 in state s_0, holding is *true* at time 5 but then it becomes *false* at time 5 in the resulting state s_1. We have a similar situation when the brick is caught at time 8. This does not happen in sitcalc.

In [39] Reiter introduced natural actions in sitcalc. These actions obey natural laws and will occur at their predicted times, provided no earlier actions prevent them from occurring. In order to determine the occurrence times of such actions Reiter uses action precondition axioms which are equivalent to triggers of H. However, Reiter restricts his worlds to natural worlds where every action is a natural action. Language H does not have such restrictions.

A logical theory of sitcalc is implemented by translating the corresponding axioms into rules of Prolog. Whenever the resulting Prolog program succeeds on a sentence, then that sentence is logically entailed by the theory and whenever it fails on a sentence, then the negation of that sentence is entailed by the theory. However, Prolog in general has several limitations that make it inadequate for reasoning about dynamic systems. Prolog cannot generate multiple models and as a result cannot deal with uncertainty or incomplete information which is very common in dynamic systems. Negation as failure (*not*) operator was introduced in Prolog as a way to achieve non-monotonicity. But the procedural semantics of *not* gives incorrect results when reasoning about defaults [4]. Non monotonic logics such as answer set programming(ASP) are capable of generating multiple models and reasoning about defaults. We implement theories of H by translating them into ASP programs and reap the benefits of this approach.

CHAPTER VII
PROOFS OF THEOREMS

In this chapter we present proofs for theorem 4.1 and theorem 6.1. We begin with theorem 4.1.

7.1 Proof of Theorem 4.1

This theorem is similar to theorem 1 from [3]. Both theorems establish relationship between action theories and logic programming. However, the underlying action languages and logic programming languages are different. Therefore, in some situations we use ideas from the proof of theorem 1 to prove some of our own results while in other situations we came up with our own approach. Here is the statement of our theorem.

Given a domain description $\langle AD, \Gamma_n \rangle$; if Γ_n is complete then M is a model of Γ_n iff M is defined by some answer set of $\Pi^n(AD, \Gamma_n)$.

Proof

Given a recorded history, Γ_l, upto moment l we will refer to l as the length of Γ_l. We will give a proof by induction on the length of a recorded history, denoted by l.

From now onwards, whenever we refer to an $\mathcal{AC}$ program we refer to the ground version of it. This is because answer sets are defined only for ground programs. Therefore, by $\Pi^n(AD, \Gamma_n)$ we mean a ground $\mathcal{AC}$ program that contains all possible instantiations for time. If time is continuous then this program is infinitely large.

Base case: for $l = 1$.
If Γ_1 is complete then $M = \langle s_0, a_0, s_1 \rangle$ is a model of Γ_1 iff M is defined by some answer set of $\Pi^1(AD, \Gamma_1)$.

left to right:
We must show that if $M = \langle s_0, a_0, s_1 \rangle$ is a model of Γ_1 then M is defined by some answer set of $\Pi^1(AD, \Gamma_1)$.

Since M is a model of Γ_1, M is a path of $TD(AD)$ such that

$$a_0 = \{a \mid hpd(a, 0, t_0) \in \Gamma_1\}$$

and $s_0 \models \{start = 0, end = t_0\}$.

Since s_1 follows s_0, we have $s_1 \models \{start = t_0, end = t_1\}$ where $t_0 \leq t_1$.

By $\Pi^1(AD, \Gamma_1)$ we denote the $\mathcal{AC}$ program

$$\Pi^1(AD, \Gamma_1) = sig^1(AD) \cup \bigcup_{r \in AD} \tau(r) \cup \beta(AD) \cup \Gamma_1 \cup R(AD) \cup \{end(1) = t_1\}$$

where t_1 is the end time of s_1.

Let

$$occurs(a_0, 0) = \{occurs(a, 0) \mid a \in a_0\}$$

and

$$\alpha(s_0, 0) = \bigcup_{s_0 \models l} \alpha(l, 0)$$
$$\alpha(s_1, 1) = \bigcup_{s_1 \models l} \alpha(l, 1)$$

These sets represent the $\mathcal{AC}$ encoding of a_0, s_0 and s_1 respectively.

Let $triggered(s_0)$ be the set of all atoms of the form $triggered(e, 0)$ such that s_0 satisfies the trigger for e. Therefore,

$$triggered(s_0) = \{triggered(e, 0) \mid \ s_0 \text{ satisfies } l_1, \ldots l_n \text{ triggers } e \in AD\}$$

We also have $\neg triggered(s_0)$ which is defined as follows.

$$\neg triggered(s_0) = \{\neg triggered(e, 0) \mid \ triggered(e, 0) \notin triggered(s_0) \wedge$$
$$e \text{ appears in a trigger}\}$$

Let $def(AD) \subseteq sig^1(AD)$ be the collection of statements of $sig^1(AD)$ that define sorts of $\Pi^1(AD, \Gamma_1)$.

Let A be the set

$$A = \begin{array}{l} def(AD) \cup \alpha(s_0, 0) \cup \alpha(s_1, 1) \cup occurs(a_0, 0) \cup \\ \Gamma_1 \cup triggered(s_0) \cup \neg triggered(s_0) \end{array} \quad (7.1)$$

By construction of A it is clear that A defines $M = \langle s_0, a_0, s_1 \rangle$. We will show that A is an answer set of $\Pi^1(AD, \Gamma_1)$.

We will begin by showing that A is *closed under rules of* $\Pi^1(AD, \Gamma_1)$. Let us denote this program by P. The corresponding reduct P^A contains

- $sig^1(AD)$ which contains declarations and definitions for various sorts of the program. The declarations are directives to the complier and therefore can be safely ignored. However, by construction, A is closed under $def(AD)$ which contains definitions for various sorts.

- Γ_1 which is the recorded history. By construction A is closed under Γ_1.

- the fact $end(1) = t_1$. Since $s_1 \models end = t_1$, from(7.1) it is true that $\alpha(end = t_1, 1) \in A$. In other words, $end(1) = t_1 \in A$. Hence A is closed under this rule.

- rules of the form

  $$occurs(a, 0) :- hpd(a, 0, t).$$

  which are part of $R(AD)$. If $hpd(a, 0, t) \in A$ then by(7.1) it is true that $hpd(a, 0, t) \in \Gamma_1$. Since M is a model of Γ_1, $a_0 = \{a \mid hpd(a, 0, t) \in \Gamma_1\}$. Consequently, $occurs(a, 0) \in occurs(a_0, 0)$. By construction of A, $occurs(a, 0) \in A$ and therefore A is closed under such rules.

- rules of the form

  $$end(0) = t :- hpd(a, 0, t).$$

  which are part of $R(AD)$. If $hpd(a, 0, t) \in A$ then by 7.1 it is true that $hpd(a, 0, t) \in \Gamma_1$. Since M is a model of Γ_1, $s_0 \models end = t$. Consequently, $\alpha(end = t, 0) \in \alpha(s_0, 0)$. By construction $end(0) = t \in A$. Hence A is closed under such rules.

- rules of the form

$$\alpha(p = y, 0) :- obs(p, 0, 0, y).$$

where p is a non-process fluent and rules of the form

$$p(0, 0, y) :- obs(p, 0, 0, y).$$

where p is a process fluent. Since Γ_1 is complete the body of these rules is contained in A. Since M is a model of Γ_1, for every non-process fluent p, if $obs(p, 0, 0, y) \in \Gamma_1$ then $s_0 \models p = y$. By construction of A, $\alpha(p = y, 0) \in A$. Similarly, for every process fluent p, if $obs(p, 0, 0, y) \in \Gamma_1$ then $s_0 \models p(0) = y$. By construction of A, $\alpha(p(0) = y, 0) \in A$. In other words, $p(0, 0, y) \in A$. Hence A is closed under both forms of rules.

- rules of the form

$$\begin{aligned} :- \quad & obs(p, i, t, y), \\ & \neg\alpha(p = y, i). \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} :- \quad & obs(p, i, t, y), \\ & \neg p(i, t, y). \end{aligned}$$

where p is a process fluent and i is an integer from $[0, 1]$. If $obs(p, i, t, y) \notin \Gamma_1$ then A is closed under both rules. Since M is a model of Γ_1, for every non-process fluent p, if $obs(p, i, t, y) \in \Gamma_1$ then $s_i \models p = y$. By construction of A, $\alpha(p = y, i) \in A$. Since states are consistent sets of literals, $s_i \not\models \neg p = y$. By construction of A, $\neg\alpha(p = y, i) \notin A$. Similarly, for every process fluent p, if $obs(p, i, t, y) \in \Gamma_1$ then $s_i \models p(t) = y$. By construction of A, $\alpha(p(t) = y, i) \in A$. Using the same argument $\neg\alpha(p(t) = y, i) \notin A$. In other words, $\neg p(i, t, y) \notin A$. Hence A is closed under both rules.

- rules of the form

$$\begin{aligned} \neg\alpha(p = y_1, i) :- \quad & \alpha(p = y_2, i), \\ & y_1 \neq y_2. \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\neg p(i,t,y_1) :- \; p(i,t,y_2),$$
$$y_1 \neq y_2.$$

where p is a process fluent and i is an integer from $[0,1]$. If $\alpha(p = y_2, i) \notin A$ then A is closed under the first rule. Similarly, if $p(i,t,y_2) \notin A$ then A is closed under the second rule. On the other hand if $\alpha(p = y_2, i) \in A$ then from (7.1) it is true that $s_i \models p = y_2$. Since states are consistent sets of literals, for any $y_1 \in range(p)$ such that $y_1 \neq y_2$, $s_i \models p \neq y_1$. By construction of A, $\neg\alpha(p = y_1, i) \in A$. We use a similar argument to deduce that $\neg p(i,t,y_1) \in A$. Hence A is closed under both rules.

- rules of the form

$$\alpha(p = y, 1) : -\alpha(p = y, 0).$$

for every $\alpha(p = y, 1) \in A$. These rules are the *reduced* inertia axioms from $\beta(AD)$. Our reasoning is that by construction of A, for every non-process fluent p, $\exists y \in range(p)$ such that $\alpha(p = y, 1) \in A$. And for any $y_1 \in range(p)$ such that $y_1 \neq y$, $\neg\alpha(p = y_1, 1) \in A$. Hence A is closed under such rules.

- the fact

$$start(0) = 0.$$

Since $s_0 \models start = 0$ it is true that $start(0) = 0 \in A$. Hence A is closed under this rule.

- rules of the form

$$start(1) = t :- \; end(0) = t.$$

which are part of $\beta(AD)$. If $end(0) = t \notin A$ then A is closed under such rules. If $end(0) = t \in A$ then from (7.1) it is true that $s_0 \models end = t$. Since M is a path, it is true that s_1 follows s_0 and $s_1 \models start = t$. By construction of A, $start(1) = t \in A$. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} :- \ & start(i) = t_1, \\ & end(i) = t_2, \\ & t_1 > t_2. \end{aligned}$$

$$\begin{aligned} :- \ & start(i) = t_1, \\ & t_1 \geq \omega. \end{aligned}$$

  where i is an integer from $[0, 1]$. Since s_i is a state it is true that if $s_i \models start = t_1$ and $s_i \models end = t_2$ then $t_1 \leq t_2 \wedge t_1 < \omega$. By construction of A, the body of these rules is never satisfied. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} \alpha(p = y, 1) :- \ & occurs(a, 0), \\ & \alpha(l_1, 0), \\ & \ldots, \\ & \alpha(l_n, 0). \end{aligned}$$

  that are obtained by translating dynamic causal laws of AD. If the body is not contained in A then A is closed under such rules. If the body is contained in A then from (7.1) it is true that $a \in a_0$ and $s_0 \models l_i$ for every i such that $1 \leq i \leq n$. As we can see s_0 satisfies the preconditions of the dynamic causal law and we are given that s_1 is the successor state w.r.t s_0 and a_0. From equation (2.1) we conclude that $s_1 \models p = y$. By construction of A, $\alpha(p = y, 1) \in A$. Hence A is closed under such rules.

- rules of the form

$$\alpha(p = y, i) : -\alpha(l_1, i), \ldots, \alpha(l_n, i). \tag{7.2}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} p(i,t,y) :- \ & \alpha(l_1,i), \\ & \ldots\ldots \\ & \alpha(l_n,i), \\ & start(i) = t_1, \\ & end(i) = t_2, \\ & t_1 \leq t \leq t_2, \\ & t < \omega, \\ & y = g(t). \end{aligned} \tag{7.3}$$

where p is a process fluent and i is an integer from $[0,1]$. If the bodies of these rules are not contained in A then A is closed under such rules. We know that rules of the form (7.2) encode state constraints of the form

$$p = y \text{ if } l_1, \ldots, l_n$$

If the body of (7.2) is contained in A then from (7.1) it is true that $s_i \models l_k$ for every k such that $1 \leq k \leq n$. Since s_0 and s_1 are states, they are closed under state constraints of AD. From equation (2.1) we conclude that $s_i \models p = y$. By construction of A, $\alpha(p = y, i) \in A$. Hence A is closed under (7.2). Now suppose that the body of (7.3) is contained in A. From (7.1) it is true that $s_i \models l_k$ for every k such that $1 \leq k \leq n$ and $s_i \models \{start = t_1, end = t_2\}$. We know that rules of the form (7.3) encode state constraints of the form

$$p = \lambda T.g(T) \text{ if } l_1, \ldots, l_n$$

Since s_i is a state, it is closed under state constraints of AD. From equation (2.1) we conclude that $s_i \models p = \lambda T.g(T)$. By construction of A, $\alpha(p = \lambda T.g(T), i) \subseteq A$. In other words, $p(i,t,y) \in A$. Hence A is closed under (7.3).

- rules of the form

$$\begin{aligned} :- \ & occurs(e_1, 0), \\ & \ldots\ldots, \\ & occurs(e_m, 0), \\ & \alpha(l_1, 0), \\ & \ldots\ldots, \\ & \alpha(l_n, 0). \end{aligned} \tag{7.4}$$

  which encode executability conditions of the form

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n$$

  Since $\langle s_0, a_0, s_1 \rangle \in TD(AD)$, a_0 is possible in s_0. This implies that either $\{e_1, \ldots, e_m\} \not\subseteq a_0$ or $s_0 \not\models l_k$ for some k in $[1, n]$. By construction of A, the body of (7.4) is not satisfied. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} :- \ & occurs(e_1, 1), \\ & \ldots\ldots, \\ & occurs(e_m, 1), \\ & \alpha(l_1, 1), \\ & \ldots\ldots, \\ & \alpha(l_n, 1). \end{aligned}$$

  Since A does not contain any atoms of the form $occurs(a, 1)$, the body of these rules is never satisfied. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} triggered(e, 0) :- \ & \alpha(l_1, 0), \\ & \ldots\ldots, \\ & \alpha(l_m, 0). \end{aligned}$$

  which encode triggers of the form

$$l_1, \ldots, l_m \text{ triggers } e$$

If the bodies of these rules are not contained in A then A is closed under such rules. If the bodies are contained in A then by (7.1) it is true that $s_0 \models l_k$ for every k such that $1 \leq k \leq m$. This implies that s_0 satisfies the trigger. From (7.1) we conclude that $triggered(e, 0) \in A$. Hence A is closed under such rules.

- rules of the form

$$\neg triggered(e, 0).$$

for every $\neg triggered(e, 0) \in A$. It is obvious that A is closed under such rules.

- rules of the form

$$: -triggered(e, 0).$$

for every $occurs(e, 0) \notin A$ such that e appears in a trigger. This implies that $e \notin a_0$ and by definition of completeness of actions, s_0 does not satisfy any trigger for e. From (7.1) it is true that $triggered(e, 0) \notin A$ and $\neg triggered(e, 0) \in A$. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} : - \ & \neg triggered(e, 0), \\ & occurs(e, 0). \end{aligned}$$

where e appears in a trigger. We will show that A does not satisfy the body of these rules. In the first case, if $occurs(e, 0) \in A$ then from (7.1) it is true that $e \in a_0$ and by definition of completeness of actions, s_0 satisfies a trigger for e. From (7.1) it is clear that $triggered(e, 0) \in A$ and $\neg triggered(e, 0) \notin A$. Hence A is closed under such rules. If $\neg triggered(e, 0) \in A$ then from (7.1) it is true that s_0 does not satisfy any trigger for e. By definition of completeness of action, $e \notin a_0$. By construction of A, $occurs(e, 0) \notin A$. Hence A is closed under such rules. Finally, if neither $\neg triggered(e, 0)$ nor $occurs(e, 0)$ belong to A then it is obvious that A is closed under such rules.

Our next step is to prove that *A is the minimal set closed under rules of* P^A. We will prove this by assuming that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A which later leads to a contradiction.

Proof by contradiction:
Assume that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A. Let σ be a set of literals of H such that $\sigma \subset s_1$. By $\alpha(\sigma, 1)$ we denote the set

$$\alpha(\sigma, 1) = \bigcup_{\sigma \models l} \alpha(l, 1)$$

Let

$$\begin{aligned} B = \; & def(AD) \cup \alpha(s_0, 0) \cup \alpha(\sigma, 1) \cup occurs(a_0, 0) \cup \Gamma_1 \cup \\ & triggered(s_0) \cup \neg triggered(s_0) \end{aligned} \tag{7.5}$$

such that B is closed under rules of P^A. As we can see $B \subset A$.

We know that s_1 satisfies the modified McCain-Turner equation

$$s_1 = Cn_Z(E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1))$$

Since $\sigma \subset s_1$, we have

$$\sigma \subset Cn_Z(E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1))$$

Let us see how each component of this equation is related to σ.

Let $D \subseteq P^A$ be the set of all dynamic causal laws of the form

$$a \text{ causes } p = y \text{ if } l_1, \ldots, l_m$$

such that $s_0 \models l_i$ for every i, $1 \leq i \leq m$, and $a \in a_0$. From (7.5) it is true that $\bigcup_i \alpha(l_i, 0) \subseteq B$ and $occurs(a, 0) \in B$. For every $d \in D$, we know that B is closed under the rule $\tau(d)$

$$\begin{aligned} \alpha(p = y, 1) :- \; & occurs(a, 0), \\ & \alpha(l_1, 0), \\ & \ldots\ldots, \\ & \alpha(l_n, 0). \end{aligned}$$

Since the body is contained in B we conclude that $\alpha(p = y, 1) \in B$. From (7.5) it is true that $\sigma \models p = y$. Therefore, $E_{s_0}(a_0) \subseteq \sigma$.

We know that P^A contains the reduced inertia axiom

$$\alpha(p = y, 1) : -\alpha(p = y, 0).$$

for every $\alpha(p = y, 1) \in A$. From 7.1 it is true that $p = y \in s_1$. If $p = y \in s_0 \cap s_1$ then by (7.5) it is true that $\alpha(p = y, 0) \in B$. Since B is closed under rules of P^A we conclude that $\alpha(p = y, 1) \in B$. From (7.5) it is true that $\sigma \models p = y$. Therefore, $s_0 \cap s_1 \subseteq \sigma$.

Since $\langle s_0, a_0, s_1 \rangle \in TD(AD)$ it is true that s_1 follows s_0. Therefore, if $s_0 \models end = t_0$ then $s_1 \models start = t_0$. From (7.5) it is true that $\alpha(end = t_0, 0) \in B$. In other words, $end(0) = t_0 \in B$. We know that P^A contains the rule

$$start(1) = t_0 : -end(0) = t_0.$$

Since B is closed under this rule, we conclude that $start(1) = t_0 \in B$. From (7.5) it is true that

$$\sigma \models start = t_0 \tag{7.6}$$

Secondly, it is true that P^A contains the fact $end(1) = t_1$ such that $s_1 \models end = t_1$. Since B is closed under rules of P^A, it must be true that $end(1) = t_1 \in B$. From 7.5 it is true that

$$\sigma \models end = t_1 \tag{7.7}$$

As we can see $T_{s_0}(s_1) = \{start = t_0, end = t_1\}$ and from (7.6) and (7.7) we conclude that $T_{s_0}(s_1) \subseteq \sigma$.

We know that P^A contains rules of the form

$$\alpha(p = y, 1) : -\alpha(l_1, 1), \ldots, \alpha(l_m, 1).$$

which encode state constraints of the form

$$p = y \text{ if } l_1, \ldots, l_m$$

where p is a non-process fluent. Since B is closed under such rules, if $\bigcup_{1 \leq i \leq m} \alpha(l_i, 1) \subseteq B$ then $\alpha(p = y, 1) \in B$. From (7.5) it is true that $\sigma \models p = y$ whenever $\sigma \models l_i$ for every i, $1 \leq i \leq m$. We use a similar line of reasoning if p is a process fluent. Hence σ is closed under state constraints of AD.

We have shown that $E_{s_0}(a_0) \subseteq \sigma$, $s_0 \cap s_1 \subseteq \sigma$, $T_{s_0}(s_1) \subseteq \sigma$ and that σ is closed under state constraints of AD. Therefore, it is impossible that

$$\sigma \subset Cn_Z(E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1))$$

Contradiction. Therefore, our assumption that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A is *false*. We conclude that A is the minimal set closed under the rules of P^A.

Hence we proved that A is an answer set of $\Pi^1(AD, \Gamma_1)$. $\square$

Right to left.

We will show that if $\langle s_0, a_0, s_1 \rangle$ is a defined by an answer set of $\Pi^1(AD, \Gamma_1)$ then $\langle s_0, a_0, s_1 \rangle$ is a model of Γ_1.

Let A be an answer set of $\Pi^1(AD, \Gamma_1)$ such that

$$s_i = \{l \mid \alpha(l, i) \subseteq A\} \tag{7.8}$$

for every integer $i \in [0, 1]$ and

$$a_0 = \{a \mid occurs(a, 0) \in A\}$$

From definition 4.2.1, A defines the sequence $\langle s_0, a_0, s_1 \rangle$.

Let us denote $\langle s_0, a_0, s_1 \rangle$ by M and $\Pi^1(AD, \Gamma_1)$ by P. We must show that $M \in TD(AD)$ and that M is a model of Γ_1.

To begin we will show that s_0 and s_1 are complete and consistent. Later on, we will show that s_0 is a state and that s_1 is a successor state w.r.t s_0 and a_0.

s_0 and s_1 are consistent.

We know that A is a consistent set of literals. From (7.8) it is obvious that s_i is a consistent set of literals for every integer $i \in [0, 1]$.

s_0 and s_1 are complete.

We know P^A contains rules of the form

$$\alpha(p = y, 0) : -obs(p, 0, 0, y).$$

where p is a non-process fluent. Since Γ_1 is complete and $\Gamma_1 \subseteq A$, A contains $\alpha(p = y, 0)$ for every non-process fluent p. From (7.8), $p = y \in s_0$, for every non-process fluent p. Therefore, s_0 is complete w.r.t non-process fluents. Next, we will prove that s_0 is complete w.r.t process fluents.

Proof by contradiction. Let p be a process fluent such that $\forall y \in range(process)$ and $\forall t \in [start(0), end(0)]$, $p(t) = y \notin s_0$. P^A contains rules of the form

$$\begin{aligned} p(0, t, y) : - \ & \alpha(l_1, 0), \\ & \ldots\ldots \\ & \alpha(l_n, 0), \\ & start(0) = t_1, \\ & end(0) = t_2, \\ & t_1 \leq t \leq t_2, \\ & t < \omega, \\ & y = g(t). \end{aligned}$$

for every process fluent p. Let us suppose that the body consists of atoms involving only non-process fluents. Since s_0 is complete w.r.t non-process fluents, the body of this rule is satisfied by A. Since A is closed under such rules, we conclude that $p(0, t, y) \in A$. From (7.8), $p(t) = y \in s_0$. Contradiction. Hence s_0 is complete w.r.t process fluents. Therefore, s_0 is complete.

We will now prove that s_1 is complete. Proof will be given in two parts. First we will prove that s_1 is complete w.r.t non-process fluents.

Proof by contradiction. Let p be a non-process fluent such that $p = y \in s_0$ and $\forall y \in range(p)$, $p = y \notin s_1$. P^A contains rules of the form

$$\alpha(p = y, 1) : -\alpha(p = y, 0).$$

Since $p = y \in s_0$, $\alpha(p = y, 0) \in A$. Since A is closed under such rules, we conclude that $\alpha(p = y, 1) \in A$. From (7.8), $p = y \in s_1$. Contradiction. Hence s_1 is complete w.r.t non-process fluents. Next, we will prove that s_1 is complete w.r.t process fluents.
Proof by contradiction. Let p be a process fluent such that $\forall y \in range(process)$ and $\forall t \in [start(1), end(1)]$, $p(t) = y \notin s_1$. P^A contains rules of the form

$$\begin{aligned} p(1, t, y) : - \ & \alpha(l_1, 1), \\ & \ldots\ldots \\ & \alpha(l_n, 1), \\ & start(1) = t_1, \\ & end(1) = t_2, \\ & t_1 \leq t \leq t_2, \\ & t < \omega, \\ & y = g(t). \end{aligned}$$

for every process fluent p. Let us suppose that the body of such a rule does not contain atoms involving process fluents. Since s_1 is complete w.r.t non-process fluents, the body of this rule is satisfied by A. Since A is closed under such rules, we conclude that $p(1, t, y) \in A$. From (7.8), $p(t) = y \in s_1$. Contradiction. Hence s_1 is complete w.r.t process fluents. Thus, s_1 is complete.

A set, B, of $\mathcal{AC}$ literals is *complete* if for every integer $i \in [0, n]$, and for every non-process fluent p, $\exists y \in range(p)$ such that $\alpha(p = y, i) \in B$, and for every process fluent p, $\exists y \in range(process)$ such that $\alpha(p(t) = y, i) \in B$ where $start(i) \leq t \leq end(i) \wedge t < \omega$.

Since both s_0 and s_1 are complete we conclude that A is complete.

s_0 is a state

We will show that s_0 is closed under state constraints of AD. The reduct P^A contains rules of the form

$$\alpha(p = y, 0) : -\alpha(l_1, 0), \ldots, \alpha(l_n, 0).$$

Since A is closed under such rules, if $\bigcup_{1 \leq i \leq n} \alpha(l_i, 0) \subseteq A$ then $\alpha(p = y, 0) \in A$. From (7.8) it is true that $p = y \in s_0$ whenever $\{l_1, \ldots, l_n\} \subseteq s_0$. Hence s_0 is closed under state constraints of AD.

P^A contains the fact $start(0) = 0$. Since A is closed under rules of P^A, $start(0) = 0 \in A$. From (7.8) we conclude that $start = 0 \in s_0$. P^A contains rules of the form

$$end(0) = t : -hpd(a, 0, t).$$

Since Γ_1 is complete, $hpd(a, 0, t) \in A$. Since A is closed under such rules it is true that $end(0) = t \in A$. From (7.8) we conclude that $end = t \in s_0$. Since A is a consistent set of literals closed under rules of the form

$$\begin{aligned} : - \ & start(0) = t_1, \\ & end(0) = t_2, \\ & t_1 > t_2. \end{aligned}$$

$$\begin{aligned} : - \ & start(0) = t, \\ & t \geq \omega \end{aligned}$$

the bodies of these rules are never satisfied by A. As a result the constraint $t_1 \leq t_2 \wedge t_1 < \omega$, where $start = t_1$ and $end = t_2$, is not violated. Since $s_0 \models start = 0$ and $s_0 \models end = t$ we conclude that $t > 0$.

Next, we will show that s_0 is closed under triggers of AD. We know that P^A contains rules of the form

$$end(0) = t_0 : -hpd(a, 0, t_0).$$

Since Γ_1 is complete and $\Gamma_1 \subseteq A$, we conclude that $end(0) = t_0 \in A$. From (7.8), $end = t_0 \in s_0$.

We know that the agent is capable of making correct observations including observations about triggered actions. If an action was triggered earlier than t_0 then it must have been observed. Since such an observation is not part of Γ_1 we conclude that $\neg\exists L$ such that L satisfies atleast one trigger of AD and $s_0 \setminus L = \{end = t_2\}$ and $L \setminus s_0 = \{end = t_1\}$ and $t_1 < t_2$. Therefore, s_0 is closed under triggers of AD.

Since A is an answer set of P, for every process fluent $p \in \Sigma(AD)$ and integer $i \in [0,1]$, $\exists \lambda T.g(T) \in process$ such that $\{start(i) = t_1, end(i) = t_2\} \subseteq A$ and for every t, $t_1 \leq t \leq t_2 \wedge t < \omega$, $\lambda T.g(T)(t) = y$ and $p(i,t,y) \in A$. From (7.8) it is true that $p = \lambda T.g(T) \in s_i$. It is possible to see that the $\lambda T.g(T)$ is defined over the domain $\{t \mid t_1 \leq t \leq t_2 \wedge t < \omega\}$.

Hence we conclude that s_0 is a state.

a_0 is possible in s_0.

Proof by contradiction. Assume that AD contains an executability condition

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n$$

such that $\{e_1, \ldots, e_m\} \subseteq a_0$ and $\{l_1, \ldots, l_n\} \subseteq s_0$. This implies that

$$\{occurs(e_1, 0), \ldots, occurs(e_m, 0)\} \subseteq A$$

and $\bigcup_{1 \leq i \leq n} \alpha(l_i, 0) \subseteq A$. We know that P^A contains the rule

$$\begin{aligned} :- \; & occurs(e_1, 0), \\ & \ldots, \\ & occurs(e_m, 0), \\ & \alpha(l_1, 0), \\ & \ldots, \\ & \alpha(l_n, 0). \end{aligned}$$

which is the $\mathcal{AC}$ encoding of the executability condition. As we can see, the body of this rule is satisfied by A and A is not a consistent set of literals. Contradiction. Hence our assumption is false. We conclude that a_0 is possible in s_0.

a_0 is complete w.r.t s_0.

For every action e that appears in a trigger, we know that P^A contains the rule

$$: -triggered(e, 0).$$

such that $occurs(e, 0) \notin A$. This implies that $e \notin a_0$. Since A is an answer set of P^A it is impossible that $triggered(e, 0) \in A$. Therefore, for every rule of P^A containing $triggered(e, 0)$ in the head

$$\begin{aligned} triggered(e, 0) : - \ & \alpha(l_1, 0), \\ & \ldots, \ldots, \\ & \alpha(l_n, 0). \end{aligned}$$

$\bigcup_{1 \leq i \leq n} \alpha(l_i, 0) \not\subseteq A$. This implies that $\{l_1, \ldots, l_n\} \not\subseteq s_0$. Therefore, s_0 does not satisfy any trigger for e. This is indeed true because $e \notin a_0$.

For every action e that appears in a trigger, P^A contains the rule

$$\begin{aligned} : - \ & \neg triggered(e, 0), \\ & occurs(e, 0). \end{aligned}$$

Consider the case when $occurs(e, 0) \in A$. As we can see, this implies that $e \in a_0$. Since A is an answer set of P^A, $\neg triggered(e, 0) \notin A$. Since we assume closed world assumption for *triggered* atoms, it is true that $triggered(e, 0) \in A$. Therefore, there must be atleast one rule of P^A containing $triggered(e, 0)$ in the head

$$\begin{aligned} triggered(e, 0) : - \ & \alpha(l_1, 0), \\ & \ldots, \ldots, \\ & \alpha(l_n, 0). \end{aligned}$$

such that $\bigcup_{1 \leq i \leq n} \alpha(l_i, 0) \subseteq A$. This implies that $\{l_1, \ldots, l_n\} \subseteq s_0$. Therefore, s_0 satisfies a trigger for e.

Now consider the case when $\neg triggered(e, 0) \in A$ which implies that $occurs(e, 0) \notin A$ (since A is an answer set). This implies that $e \notin a_0$. Therefore, for every rule of P^A

containing $triggered(e, 0)$ in the head

$$\begin{aligned} triggered(e,0) :- \ & \alpha(l_1,0), \\ & \ldots,\ldots, \\ & \alpha(l_n,0). \end{aligned}$$

$\bigcup_{1\leq i\leq n} \alpha(l_i, 0) \not\subseteq A$. This implies that $\{l_1, \ldots, l_n\} \not\subseteq s_0$. Therefore, s_0 does not satisfy any trigger for e. From all our cases we conclude that $e \in a_0$ iff s_0 satisfies a trigger for e. Hence, a_0 is complete w.r.t s_0.

s_1 follows s_0.

We know that P^A contains rules of the form

$$start(1) = t_0 : -end(0) = t_0.$$

Since Γ_1 is complete, we concluded earlier that $end(0) = t_0 \in A$. Since A is closed under such rules we conclude that $start(1) = t_0 \in A$. From (7.8) it is true that $start = t_0 \in s_1$. P^A also contains the fact $end(1) = t_1$ such that $t_1 \geq t_0$. Since A is closed under rules of P^A we conclude that $end(1) = t_1 \in A$. From (7.8) it is true that $end = t_1 \in s_1$. As we can see, $\{start = 0, end = t_0\} \subseteq s_0$ and $\{start = t_0, end = t_1\} \subseteq s_1$. Therefore, s_1 follows s_0.

s_1 satisfies the modified McCain-Turner equation.

We will show that

$$s_1 = Cn_Z(E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1))$$

We will begin by proving that $E_{s_0}(a_0) \subseteq s_1$. We know that P^A contains rules of the form

$$\begin{aligned} \alpha(p = y, 1) :- \ & occurs(a,0), \\ & \alpha(l_1,0), \\ & \ldots, \\ & \alpha(l_n,0). \end{aligned}$$

Since A is closed under such rules, if $\bigcup_{1\leq i\leq n} \alpha(l_i, 0) \subseteq A$ and $occurs(a, 0) \in A$ then $\alpha(p = y, 1) \in A$. From (7.8) it is true that $p = y \in s_1$. Therefore, $E_{s_0}(a_0) \subseteq s_1$.

$s_0 \cap s_1 \subseteq s_1$ is trivially true.

Since s_1 follows s_0, $T_{s_0}(s_1) = \{start = t_0, end = t_1\}$. Since $\{start = t_0, end = t_1\} \subseteq s_1$, we conclude that $T_{s_0}(s_1) \subseteq s_1$.

Next, we will show that s_1 is closed under state constraints of AD. We know that P^A contains rules of the form

$$\alpha(p = y, 1) : -\alpha(l_1, 1), \ldots, \alpha(l_n, 1).$$

where p is a non-process fluent. Since A is closed under such rules, if $\bigcup_{1 \leq i \leq n} \alpha(l_i, 1) \subseteq A$ then $\alpha(p = y, 1) \in A$. From (7.8) it is true that $p = y \in s_1$ whenever $\{l_1, \ldots, l_n\} \subseteq s_1$. We will use a similar line of reasoning if p is a process fluent. Hence, s_1 is closed under state constraints of AD.

We must also show that s_1 is minimal.

Proof by contradiction.

Assume that $\exists \sigma \subset s_1$ such that $E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1) \subseteq \sigma$ and σ is closed under state constraints of AD.

Let A' be obtained from A by removing atoms of the form $\alpha(l, 1)$ such that $l \in s_1 \setminus \sigma$. Since $\sigma \subset s_1$, $A' \subset A$. Both σ and s_1 agree upon atoms from $E_{s_0}(a_0) \cup (s_0 \cap s_1) \cup T_{s_0}(s_1)$. Therefore, for every $l \in s_1 \setminus \sigma$, $\exists$ a state constraint, r, containing l in the head

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

such that $\{l_1, \ldots, l_n\} \subseteq s_1$ and $\{l_1, \ldots, l_n\} \not\subseteq \sigma$. From construction of A' it is true that A' does not satisfy the body of $\tau(r)$

$$\alpha(l, 1) : -\alpha(l_1, 1), \ldots, \alpha(l_n, 1).$$

which is the $\mathcal{AC}$ encoding of the state constraint. Hence A' is closed under rules of P^A. This implies that A is not an answer set of P. Contradiction. Therefore, our assumption is false and we conclude that s_1 is minimal.

s_1 is closed under triggers of AD.

We know that P^A contains the fact $end(1) = t_1$. Since A is closed under such rules, $end(1) = t_1 \in A$. From (7.8), $end = t_1 \in s_1$. From construction of P, t_1 is indeed the end time of s_1. Therefore, $\neg\exists L$ such that L satisfies atleast one trigger of AD and $s_1 \setminus L = \{end = t_2\}$ and $L \setminus s_1 = \{end = t_1\}$ and $t_1 < t_2$. Therefore, s_1 is closed under triggers of AD.

At this point we proved that $M = \langle s_0, a_0, s_1\rangle \in TD(AD)$. We must show that M is a model of Γ_1.

We will show that $a_0 = \{a \mid hpd(a, 0, t) \in \Gamma_1\}$. P^A contains rules of the form

$$occurs(a, 0) : -hpd(a, 0, t).$$

Since Γ_1 is complete and $\Gamma_1 \subseteq A$, we conclude that $occurs(a, 0) \in A$ for every $hpd(a, 0, t) \in \Gamma_1$. From (7.8) it is true that $a_0 = \{a \mid hpd(a, 0, t) \in \Gamma_1\}$.

P^A contains rules of the form

$$end(0) = t : -hpd(a, 0, t).$$

Since $\Gamma_1 \subseteq A$ and A is closed under such rules, we conclude that $end(0) = t \in A$. From (7.8), $end = t \in s_0$.

P^A contains rules of the form

$$\begin{aligned} : - \ & obs(p, i, t, y), \\ & \neg\alpha(p = y, i). \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} : - \ & obs(p, i, t, y), \\ & \neg p(i, t, y). \end{aligned}$$

where p is a process fluent and i is an integer from $[0, 1]$. Let us consider the first rule. Since A is an answer set of P, if $obs(p, i, t, y) \in \Gamma_1$ then $\neg\alpha(p = y, i) \notin A$. Therefore,

the body of every rule containing $\neg\alpha(p = y, i)$ in the head

$$\neg\alpha(p = y, i) :- \ \alpha(p = y_1, i),$$
$$y \neq y_1.$$

is not satisfied by A. Since A is complete we conclude that $\alpha(p = y, i) \in A$. From (7.8), $p = y \in s_i$.

We use a similar argument for the second rule. Since A is an answer set of P, if $obs(p, i, t, y) \in \Gamma_1$ then $\neg p(i, t, y) \notin A$. Therefore, the body of every rule containing $\neg p(i, t, y)$ in the head

$$\neg p(i, t, y) :- \ p(i, t, y_1),$$
$$y \neq y_1.$$

is not satisfied by A. Since A is complete we conclude that $p(i, t, y) \in A$. In other words, $\alpha(p(t) = y, i) \in A$. From (7.8), $p(t) = y \in s_i$. This is possible only if $\exists \lambda T.g(T) \in process$ such that $\lambda T.g(T)(t) = y$ and $p = \lambda T.g(T) \in s_i$.

Therefore, M is a model of Γ_1. □

Inductive step.

We assume that the theorem holds for recorded histories upto moment $n-1$. Therefore, if Γ_{n-1} is complete then $M = \langle s_0, a_0, s_1, \ldots, a_{n-2}, s_{n-1}\rangle$ is a model of Γ_{n-1} iff M is defined by some answer set of $\Pi^{n-1}(AD, \Gamma_{n-1})$.

We must prove that the theorem holds for recorded histories upto moment n. We must show that if Γ_n is complete then $M = \langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n\rangle$ is a model of Γ_n iff M is defined by some answer set of $\Pi^n(AD, \Gamma_n)$.

Left to right:

We must show that if $M = \langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n\rangle$ is a model of Γ_n then M is defined by some answer set of $\Pi^n(AD, \Gamma_n)$.

Since M is a model of Γ_n, M is a path of $TD(AD)$ such that for every i, $0 \leq i < n$,

$$a_i = \{a \mid hpd(a, i, t) \in \Gamma_n\}$$

and for every i, $0 \leq i \leq n$, $s_i \models \{end = t \mid hpd(a, i, t) \in \Gamma_n\}$.

Since s_n follows s_{n-1}, if $s_{n-1} \models \{start = t_{n-2}, end = t_{n-1}\}$ then $s_n \models \{start = t_{n-1}, end = t_n\}$ where $t_{n-1} \leq t_n$.

By $\Pi^n(AD, \Gamma_n)$ we denote the $\mathcal{AC}$ program

$$\Pi^n(AD, \Gamma_n) = sig^n(AD) \cup \bigcup_{r \in AD} \tau(r) \cup \beta(AD) \cup \Gamma_n \cup R(AD) \cup \{end(n) = t_n\}$$

where t_n is the end time of s_n.

For $0 \leq i < n$ let

$$occurs(a_i, i) = \{occurs(a, i) \mid a \in a_i\}$$

and for $0 \leq i \leq n$ let

$$\alpha(s_i, i) = \bigcup_{s_i \models l} \alpha(l, i)$$

These sets represent the $\mathcal{AC}$ encoding of actions and states.

For $i < n$, let $triggered(s_i)$ be the set of all atoms of the form $triggered(e, i)$ such that s_i satisfies the trigger for e. Therefore,

$$triggered(s_i) = \{triggered(e, i) \mid s_i \text{ satisfies } l_1, \ldots l_n \text{ triggers } e \in AD\}$$

We also have $\neg triggered(s_i)$ which is defined as follows.

$$\neg triggered(s_i) = \{\neg triggered(e, i) \mid \begin{array}{l} triggered(e, i) \notin triggered(s_i) \wedge \\ e \text{ appears in a trigger}\} \end{array}$$

Let $def(AD) \subseteq sig^n(AD)$ be the collection of statements of $sig^n(AD)$ that define sorts of $\Pi^n(AD, \Gamma_n)$.

Let A be the set

$$\begin{array}{rl} A = & def(AD) \cup \bigcup_{0 \leq i \leq n} \alpha(s_i, i) \cup \bigcup_{0 \leq i < n} occurs(a_i, i) \\ & \cup\ \Gamma_n \cup \bigcup_{0 \leq i < n} triggered(s_i) \cup \bigcup_{0 \leq i < n} \neg triggered(s_i) \end{array} \tag{7.9}$$

By construction of A it is clear that A defines $M = \langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n \rangle$. We will show that A is an answer set of $\Pi^n(AD, \Gamma_n)$.

We will begin by showing that A is *closed under rules of* $\Pi^n(AD, \Gamma_n)$. Let us denote this program by P. The corresponding reduct P^A contains

- $sig^n(AD)$ which contains declarations and definitions for various sorts of the program. The declarations are directives to the complier and therefore can be safely ignored. However, by construction, A is closed under $def(AD)$ which contains definitions for various sorts.

- Γ_n which is the recorded history. By construction A is closed under Γ_n.

- the fact $end(n) = t_n$. Since $s_n \models end = t_n$, from 7.9 it is true that $\alpha(end = t_n, n) \in A$. In other words, $end(n) = t_n \in A$. Hence A is closed under this rule.

- rules of the form

$$occurs(a, i) : -hpd(a, i, t).$$

  where $i < n$. If $hpd(a, i, t) \in A$ then by 7.9 it is true that $hpd(a, i, t) \in \Gamma_1$. Since M is a model of Γ_n, $a_i = \{a \mid hpd(a, i, t) \in \Gamma_n\}$. Consequently, $occurs(a, i) \in occurs(a_i, i)$. By construction of A, $occurs(a, i) \in A$ and therefore A is closed under such rules.

- rules of the form

$$end(i) = t : -hpd(a, i, t).$$

  where $i < n$. If $hpd(a, i, t) \in A$ then by 7.9 it is true that $hpd(a, i, t) \in \Gamma_n$. Since M is a model of Γ_n, $s_i \models end = t$. Consequently, $\alpha(end = t, i) \in \alpha(s_0, 0)$. By construction $end(i) = t \in A$. Hence A is closed under such rules.

- rules of the form

$$\alpha(p = y, 0) : -obs(p, 0, 0, y).$$

  where p is a non-process fluent and rules of the form

$$p(0, 0, y) : -obs(p, 0, 0, y).$$

where p is a process fluent. Since Γ_n is complete the body of these rules is contained in A. Since M is a model of Γ_n, for every non-process fluent p, if $obs(p, 0, 0, y) \in \Gamma_n$ then $s_0 \models p = y$. By construction of A, $\alpha(p = y, 0) \in A$. Similarly, for every process fluent p, if $obs(p, 0, 0, y) \in \Gamma_n$ then $s_0 \models p(0) = y$. By construction of A, $\alpha(p(0) = y, 0) \in A$. In other words, $p(0, 0, y) \in A$. Hence A is closed under both forms of rules.

- rules of the form

$$\begin{aligned} :- \ & obs(p, i, t, y), \\ & \neg\alpha(p = y, i). \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} :- \ & obs(p, i, t, y), \\ & \neg p(i, t, y). \end{aligned}$$

where p is a process fluent and i is an integer from $[0, n]$. If $obs(p, i, t, y) \notin \Gamma_n$ then A is closed under both rules. Since M is a model of Γ_n, for every non-process fluent p, if $obs(p, i, t, y) \in \Gamma_n$ then $s_i \models p = y$. By construction of A, $\alpha(p = y, i) \in A$. Since states are consistent sets of literals, $s_i \not\models \neg p = y$. By construction of A, $\neg\alpha(p = y, i) \notin A$. Similarly, for every process fluent p, if $obs(p, i, t, y) \in \Gamma_n$ then $s_i \models p(t) = y$. By construction of A, $\alpha(p(t) = y, i) \in A$. Using the same argument $\neg\alpha(p(t) = y, i) \notin A$. In other words, $\neg p(i, t, y) \notin A$. Hence A is closed under both rules.

- rules of the form

$$\begin{aligned} \neg\alpha(p = y_1, i) :- \ & \alpha(p = y_2, i), \\ & y_1 \neq y_2. \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} \neg p(i, t, y_1) :- \ & p(i, t, y_2), \\ & y_1 \neq y_2. \end{aligned}$$

where p is a process fluent and i is an integer from $[0, n]$. If $\alpha(p = y_2, i) \notin A$ then A is closed under the first rule. Similarly, if $p(i, t, y_2) \notin A$ then A is closed under the second rule. On the other hand if $\alpha(p = y_2, i) \in A$ then from 7.9 it is true that $s_i \models p = y_2$. Since states are consistent sets of literals, for any $y_1 \in range(p)$ such that $y_1 \neq y_2$, $s_i \models p \neq y_1$. By construction of A, $\neg\alpha(p = y_1, i) \in A$. We use a similar argument to deduce that $\neg p(i, t, y_1) \in A$. Hence A is closed under both rules.

- rules of the form

$$\alpha(p = y, i) : -\alpha(p = y, i - 1).$$

for every $\alpha(p = y, i) \in A$ and $i \in [1, n]$. These rules are the *reduced* inertia axioms from $\beta(AD)$. Our reasoning is that by construction of A, for every non-process fluent p, $\exists y \in range(p)$ such that $\alpha(p = y, i) \in A$. And for any $y_1 \in range(p)$ such that $y_1 \neq y$, $\neg\alpha(p = y_1, i) \in A$. Hence A is closed under such rules.

- the fact

$$start(0) = 0.$$

Since $s_0 \models start = 0$ it is true that $start(0) = 0 \in A$. Hence A is closed under this rule.

- rules of the form

$$start(i) = t : -end(i - 1) = t.$$

where $i \in [1, n]$. If $end(i - 1) = t \notin A$ then A is closed under such rules. If $end(i - 1) = t \in A$ then from 7.9 it is true that $s_{i-1} \models end = t$. Since M is a path, it is true that s_i follows s_{i-1} and $s_i \models start = t$. By construction of A, $start(i) = t \in A$. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} :- \ & start(i) = t_1, \\ & end(i) = t_2, \\ & t_1 > t_2. \end{aligned}$$

$$\begin{aligned} :- \ & start(i) = t_1, \\ & t_1 \geq \omega. \end{aligned}$$

  where i is an integer from $[0, n]$. Since s_i is a state it is true that if $s_i \models start = t_1$ and $s_i \models end = t_2$ then $t_1 \leq t_2 \wedge t_1 < \omega$. By construction of A, the body of these rules is never satisfied. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} \alpha(p = y, i) :- \ & occurs(a, i-1), \\ & \alpha(l_1, i-1), \\ & \ldots, \\ & \alpha(l_n, i-1). \end{aligned}$$

  where integer $i \in [1, n]$. If the body is not contained in A then A is closed under such rules. If the body is contained in A then from 7.9 it is true that $a \in a_{i-1}$ and $s_{i-1} \models \{l_1, \ldots, l_n\}$. As we can see s_{i-1} satisfies the preconditions of the dynamic causal law and we are given that s_i is the successor state w.r.t s_{i-1} and a_{i-1}. From equation 2.1 we conclude that $s_i \models p = y$. By construction of A, $\alpha(p = y, i) \in A$. Hence A is closed under such rules.

- rules of the form

$$\alpha(p = y, i) : -\alpha(l_1, i), \ldots, \alpha(l_n, i). \tag{7.10}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} p(i,t,y) :- \ & \alpha(l_1,i), \\ & \ldots\ldots \\ & \alpha(l_n,i), \\ & start(i) = t_1, \\ & end(i) = t_2, \\ & t_1 \leq t \leq t_2, \\ & t < \omega, \\ & y = g(t). \end{aligned} \tag{7.11}$$

where p is a process fluent and i is an integer from $[0,n]$. If the bodies of these rules are not contained in A then A is closed under such rules. We know that rules of the form 7.10 encode state constraints of the form

$$p = y \text{ if } l_1, \ldots, l_n$$

If the body of 7.10 is contained in A then from 7.9 it is true that $s_i \models \{l_1, \ldots, l_n\}$. Since s_0 and s_1 are states, they are closed under state constraints of AD. From equation 2.1 we conclude that $s_i \models p = y$. By construction of A, $\alpha(p = y, i) \in A$. Hence A is closed under 7.10. Now suppose that the body of 7.11 is contained in A. From 7.9 it is true that $s_i \models \{l_1, \ldots, l_n\}$ and $s_i \models \{start = t_1, end = t_2\}$. We know that rules of the form 7.11 encode state constraints of the form

$$p = \lambda T.g(T) \text{ if } l_1, \ldots, l_n$$

Since s_i is a state, it is closed under state constraints of AD. From equation 2.1 we conclude that $s_i \models p = \lambda T.g(T)$. By construction of A, $\alpha(p = \lambda T.g(T), i) \subseteq A$. In other words, $p(i,t,y) \in A$. Hence A is closed under 7.11.

- rules of the form

$$\begin{aligned} :- \ & occurs(e_1, i), \\ & \ldots\ldots, \\ & occurs(e_m, i), \\ & \alpha(l_1, i), \\ & \ldots\ldots, \\ & \alpha(l_n, i). \end{aligned} \tag{7.12}$$

  where integer $i \in [0, n]$. These rules encode executability conditions of the form

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n$$

  Since $\langle s_0, a_0, s_1, \ldots, s_n \rangle \in TD(AD)$, a_i is possible in s_i $(i < n)$. This implies that either $\{e_1, \ldots, e_m\} \not\subseteq a_i$ or $s_i \not\models l_k$ for some $k \in [1, n]$. By construction of A, the body of 7.12 is not satisfied. Hence A is closed under such rules.

- rules of the form

$$\begin{aligned} triggered(e, i) :- \ & \alpha(l_1, i), \\ & \ldots\ldots, \\ & \alpha(l_m, i). \end{aligned}$$

  where integer $i \in [0, n)$. These rules encode triggers of the form

$$l_1, \ldots, l_m \text{ triggers } e$$

  If the bodies of these rules are not contained in A then A is closed under such rules. If the bodies are contained in A then by 7.9 it is true that $s_i \models \{l_1, \ldots, l_m\}$. This implies that s_i satisfies the trigger. From 7.9 we conclude that $triggered(e, i) \in A$. Hence A is closed under such rules.

- rules of the form

$$\neg triggered(e, i).$$

  for every $\neg triggered(e, i) \in A$ and $i < n$. It is obvious that A is closed under such rules.

- rules of the form

$$: -triggered(e, i).$$

for every $i < n$ and $occurs(e, i) \notin A$ such that e appears in a trigger. This implies that $e \notin a_i$ and by definition of completeness of actions, s_i does not satisfy any trigger for e. From 7.9 it is true that $triggered(e, i) \notin A$ and $\neg triggered(e, i) \in A$. Hence A is closed under such rules.

- rules of the form

$$: - \neg triggered(e, i),$$
$$occurs(e, i).$$

where e appears in a trigger and $i < n$. We will show that A does not satisfy the body of these rules. In the first case, if $occurs(e, i) \in A$ then from 7.9 it is true that $e \in a_i$ and by definition of completeness of actions, s_i satisfies a trigger for e. From 7.9 it is clear that $triggered(e, i) \in A$ and $\neg triggered(e, i) \notin A$. Hence A is closed under such rules. If $\neg triggered(e, i) \in A$ then from 7.9 it is true that s_i does not satisfy any trigger for e. By definition of completeness of action, $e \notin a_i$. By construction of A, $occurs(e, i) \notin A$. Hence A is closed under such rules. Finally, if neither $\neg triggered(e, i)$ nor $occurs(e, i)$ belong to A then it is obvious that A is closed under such rules.

Our next step is to prove that *A is the minimal set closed under rules of P^A*. We will prove this by assuming that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A which later leads to a contradiction.

Proof by contradiction:
Assume that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A. Let σ be a set of literals of H such that $\sigma \subset s_n$. By $\alpha(\sigma, n)$ we denote the set

$$\alpha(\sigma, n) = \bigcup_{\sigma \models l} \alpha(l, n)$$

Let

$$B = \begin{array}{l} def(AD) \cup \bigcup_{0 \leq i < n} \alpha(s_i, i) \cup \alpha(\sigma, n) \cup \bigcup_{0 \leq i < n} occurs(a_i, i) \\ \cup\ \Gamma_n \cup \bigcup_{0 \leq i < n} triggered(s_i) \cup \bigcup_{0 \leq i < n} \neg triggered(s_i) \end{array} \quad (7.13)$$

such that B is closed under rules of P^A. As we can see $B \subset A$.

We know that s_n satisfies the modified McCain-Turner equation

$$s_n = Cn_Z(E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n))$$

Since $\sigma \subset s_n$, we have

$$\sigma \subset Cn_Z(E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n))$$

Let us see how each component of this equation is related to σ.

Let $D \subseteq P^A$ be the set of all dynamic causal laws of the form

$$a \text{ causes } p = y \text{ if } l_1, \ldots, l_m$$

such that $s_{n-1} \models \{l_1, \ldots, l_m\}$, and $a \in a_{n-1}$. From 7.13 it is true that $\bigcup_{1 \leq k \leq m} \alpha(l_k, n-1) \subseteq B$ and $occurs(a, n-1) \in B$. For every $d \in D$, we know that B is closed under the rule $\tau(d)$

$$\begin{array}{rl} \alpha(p = y, n) : - & occurs(a, n-1), \\ & \alpha(l_1, n-1), \\ & \ldots\ldots, \\ & \alpha(l_m, n-1). \end{array}$$

Since the body is contained in B we conclude that $\alpha(p = y, n) \in B$. From 7.13 it is true that $\sigma \models p = y$. Therefore, $E_{s_{n-1}}(a_{n-1}) \subseteq \sigma$.

We know that P^A contains the reduced inertia axiom

$$\alpha(p = y, n) : -\alpha(p = y, n-1).$$

for every $\alpha(p = y, n) \in A$. From 7.9 it is true that $p = y \in s_n$. If $p = y \in s_{n-1} \cap s_n$ then by 7.13 it is true that $\alpha(p = y, n-1) \in B$. Since B is closed under rules of P^A

we conclude that $\alpha(p = y, n) \in B$. From 7.13 it is true that $\sigma \models p = y$. Therefore, $s_{n-1} \cap s_n \subseteq \sigma$.

Since $\langle s_{n-1}, a_{n-1}, s_n \rangle \in TD(AD)$ it is true that s_n follows s_{n-1}. Therefore, if $s_{n-1} \models end = t_{n-1}$ then $s_n \models start = t_{n-1}$. From 7.13 it is true that $\alpha(end = t_{n-1}, n-1) \in B$. In other words, $end(n-1) = t_{n-1} \in B$. We know that P^A contains the rule

$$start(n) = t_{n-1} : -end(n-1) = t_{n-1}.$$

Since B is closed under this rule, we conclude that $start(n) = t_{n-1} \in B$. From 7.13 it is true that

$$\sigma \models start = t_{n-1} \tag{7.14}$$

Secondly, it is true that P^A contains the fact $end(n) = t_n$ such that $s_n \models end = t_n$. Since B is closed under rules of P^A, it must be true that $end(n) = t_n \in B$. From 7.13 it is true that

$$\sigma \models end = t_n \tag{7.15}$$

As we can see $T_{s_{n-1}}(s_n) = \{start = t_{n-1}, end = t_n\}$ and from 7.14 and 7.15 we conclude that $T_{s_{n-1}}(s_n) \subseteq \sigma$.

We know that P^A contains rules of the form

$$\alpha(p = y, n) : -\alpha(l_1, n), \ldots, \alpha(l_m, n).$$

These rules encode state constraints of the form

$$p = y \text{ if } l_1, \ldots, l_m$$

where p is a non-process fluent. Since B is closed under such rules, if

$$\bigcup_{1 \leq k \leq m} \alpha(l_k, n) \subseteq B$$

then $\alpha(p = y, n) \in B$. From 7.13 it is true that $\sigma \models p = y$ whenever $\sigma \models \{l_1, \ldots, l_m\}$. We use a similar line of reasoning if p is a process fluent. Hence σ is closed under state constraints of AD.

We have shown that $E_{s_{n-1}}(a_{n-1}) \subseteq \sigma$, $s_{n-1} \cap s_n \subseteq \sigma$, $T_{s_{n-1}}(s_n) \subseteq \sigma$ and that σ is closed under state constraints of AD. Therefore, it is impossible that

$$\sigma \subset Cn_Z(E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n))$$

Contradiction. Therefore, our assumption that $\exists B$ such that $B \subset A$ and B is closed under the rules of P^A is *false*. We conclude that A is the minimal set closed under the rules of P^A.

Hence we proved that A is an answer set of $\Pi^n(AD, \Gamma_n)$. $\square$

Right to left.

We will show that if $\langle s_0, a_0, s_1, \ldots, s_n \rangle$ is a defined by an answer set of $\Pi^n(AD, \Gamma_n)$ then $\langle s_0, a_0, s_1, \ldots, s_n \rangle$ is a model of Γ_n.

Let A be an answer set of $\Pi^n(AD, \Gamma_n)$ such that

$$s_i = \{l \mid \alpha(l, i) \subseteq A\} \tag{7.16}$$

for every $i \in [0, n]$ and

$$a_k = \{a \mid occurs(a, k) \in A\}$$

for every $k \in [0, n)$. From definition 4.2.1, A defines the sequence $\langle s_0, a_0, s_1, \ldots, s_n \rangle$.

Let us denote $\langle s_0, a_0, s_1, \ldots, s_n \rangle$ by M and $\Pi^n(AD, \Gamma_n)$ by P. We must show that $M \in TD(AD)$ and that M is a model of Γ_n.

We already know that $M' = \langle s_0, a_0, s_1, \ldots, s_{n-1} \rangle \in TD(AD)$ and that M' is a model of Γ_{n-1}. It is enough to show that s_n is a successor state w.r.t s_{n-1} and a_{n-1}. We begin by showing that s_n is complete and consistent.

s_n is consistent.

We know that A is a consistent set of literals. From 7.16 it is obvious that s_n is a consistent set of literals.

s_n is complete.

Proof will be given in two parts. First we will prove that s_n is complete w.r.t non-process fluents.

Proof by contradiction. Let p be a non-process fluent such that $p = y \in s_{n-1}$ and $\forall y \in range(p)$, $p = y \notin s_n$. P^A contains rules of the form

$$\alpha(p = y, n) : -\alpha(p = y, n - 1).$$

Since $p = y \in s_{n-1}$, $\alpha(p = y, n - 1) \in A$. Since A is closed under such rules, we conclude that $\alpha(p = y, n) \in A$. From 7.16, $p = y \in s_n$. Contradiction. Hence s_n is complete w.r.t non-process fluents. Next, we will prove that s_n is complete w.r.t process fluents.

Proof by contradiction. Let p be a process fluent such that $\forall y \in range(process)$ and $\forall t \in [start(n), end(n)]$, $p(t) = y \notin s_n$. P^A contains rules of the form

$$\begin{aligned} p(n, t, y) : - \ & \alpha(l_1, n), \\ & \ldots\ldots \\ & \alpha(l_m, n), \\ & start(n) = t_1, \\ & end(n) = t_2, \\ & t_1 \leq t \leq t_2, \\ & t < \omega, \\ & y = g(t). \end{aligned}$$

for every process fluent p. Let us suppose that the body of a such a rule does not contain atoms involving process fluents. Since s_n is complete w.r.t non-process fluents, the body of this rule is satisfied by A. Since A is closed under such rules, we conclude that $p(n, t, y) \in A$. From 7.16, $p(t) = y \in s_n$. Contradiction. Hence s_n is complete w.r.t process fluents. Therefore, s_n is complete.

We have proved that for every i, $0 \leq i \leq n$, s_i is complete. Hence, we conclude that A is complete. Next, we will show that a_{n-1} is possible in s_{n-1}.

a_{n-1} is possible in s_{n-1}.

Proof by contradiction. Assume that AD contains an executability condition

$$\text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_k$$

such that $\{e_1, \ldots, e_m\} \subseteq a_{n-1}$ and $\{l_1, \ldots, l_k\} \subseteq s_{n-1}$. This implies that

$$\{occurs(e_1, n-1), \ldots, occurs(e_m, n-1)\} \subseteq A$$

and $\bigcup_{1 \leq i \leq k} \alpha(l_i, n-1) \subseteq A$. We know that P^A contains the rule

$$\begin{aligned} :- \; & occurs(e_1, n-1), \\ & \ldots, \\ & occurs(e_m, n-1), \\ & \alpha(l_1, n-1), \\ & \ldots, \\ & \alpha(l_k, n-1). \end{aligned}$$

which is the $\mathcal{AC}$ encoding of the executability condition. As we can see, the body of this rule is satisfied by A and A is not a consistent set of literals. Contradiction. Hence our assumption is false. We conclude that a_{n-1} is possible in s_{n-1}.

a_{n-1} is complete w.r.t s_{n-1}.

For every action e that appears in a trigger, we know that P^A contains the rule

$$:- triggered(e, n-1).$$

such that $occurs(e, n-1) \notin A$. This implies that $e \notin a_{n-1}$. Since A is an answer set of P^A it is impossible that $triggered(e, n-1) \in A$. Therefore, for every rule of P^A containing $triggered(e, n-1)$ in the head

$$\begin{aligned} triggered(e, n-1) :- \; & \alpha(l_1, n-1), \\ & \ldots\ldots, \\ & \alpha(l_m, n-1). \end{aligned}$$

$\bigcup_{1 \leq i \leq m} \alpha(l_i, n-1) \not\subseteq A$. This implies that $\{l_1, \ldots, l_m\} \not\subseteq s_{n-1}$. Therefore, s_{n-1} does not satisfy any trigger for e. This is indeed true because $e \notin a_{n-1}$.

For every action e that appears in a trigger, P^A contains the rule

$$\begin{aligned} :- \; & \neg triggered(e, n-1), \\ & occurs(e, n-1). \end{aligned}$$

Consider the case when $occurs(e, n-1) \in A$. As we can see, this implies that $e \in a_{n-1}$. Since A is an answer set of P^A, $\neg triggered(e, n-1) \notin A$. Since we assume closed world assumption for *triggered* atoms, it is true that $triggered(e, n-1) \in A$. Therefore, there must be atleast one rule of P^A containing $triggered(e, n-1)$ in the head

$$\begin{aligned} triggered(e, n-1) :- \ & \alpha(l_1, n-1), \\ & \ldots\ldots, \\ & \alpha(l_m, n-1). \end{aligned}$$

such that $\bigcup_{1 \leq i \leq m} \alpha(l_i, n-1) \subseteq A$. This implies that $\{l_1, \ldots, l_m\} \subseteq s_{n-1}$. Therefore, s_{n-1} satisfies a trigger for e.

Now consider the case when $\neg triggered(e, n-1) \in A$ which implies that $occurs(e, n-1) \notin A$ (since A is an answer set). This implies that $e \notin a_{n-1}$. Therefore, for every rule of P^A containing $triggered(e, n-1)$ in the head

$$\begin{aligned} triggered(e, n-1) :- \ & \alpha(l_1, n-1), \\ & \ldots\ldots, \\ & \alpha(l_m, n-1). \end{aligned}$$

$\bigcup_{1 \leq i \leq m} \alpha(l_i, n-1) \not\subseteq A$. This implies that $\{l_1, \ldots, l_m\} \not\subseteq s_{n-1}$. Therefore, s_{n-1} does not satisfy any trigger for e. From all our cases we conclude that $e \in a_{n-1}$ iff s_{n-1} satisfies a trigger for e. Hence, a_{n-1} is complete w.r.t s_{n-1}.

s_n follows s_{n-1}.

We know that P^A contains rules of the form

$$start(n) = t_{n-1} : -end(n-1) = t_{n-1}.$$

Since Γ_n is complete, it is true that $end(n-1) = t_{n-1} \in A$ such that $hpd(a, n-1, t_{n-1}) \in A$. Since A is closed under such rules we conclude that $start(n) = t_{n-1} \in A$. From 7.16 it is true that $start = t_{n-1} \in s_n$. P^A also contains the fact $end(n) = t_n$ such that $t_n \geq t_{n-1}$. Since A is closed under rules of P^A we conclude that $end(n) = t_n \in A$.

From 7.16 it is true that $end = t_n \in s_n$. As we can see, $\{start = t_{n-2}, end = t_{n-1}\} \subseteq s_{n-1}$ and $\{start = t_{n-1}, end = t_n\} \subseteq s_n$. Therefore, s_n follows s_{n-1}.

s_n satisfies the modified McCain-Turner equation.

We will show that

$$s_n = Cn_Z(E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n))$$

We will begin by proving that $E_{s_{n-1}}(a_{n-1}) \subseteq s_n$. We know that P^A contains rules of the form

$$\begin{aligned} \alpha(p = y, n) :- \ & occurs(a, n-1), \\ & \alpha(l_1, n-1), \\ & \ldots, \\ & \alpha(l_m, n-1). \end{aligned}$$

Since A is closed under such rules, if $\bigcup_{1 \leq i \leq m} \alpha(l_i, n-1) \subseteq A$ and $occurs(a, n-1) \in A$ then $\alpha(p = y, n) \in A$. From 7.16 it is true that $p = y \in s_n$. Therefore, $E_{s_{n-1}}(a_{n-1}) \subseteq s_n$.

$s_{n-1} \cap s_n \subseteq s_n$ is trivially true.

Since s_n follows s_{n-1}, $T_{s_{n-1}}(s_n) = \{start = t_{n-1}, end = t_n\}$. Since $\{start = t_{n-1}, end = t_n\} \subseteq s_n$, we conclude that $T_{s_{n-1}}(s_n) \subseteq s_n$.

Next, we will show that s_n is closed under state constraints of AD. We know that P^A contains rules of the form

$$\alpha(p = y, n) : -\alpha(l_1, n), \ldots, \alpha(l_m, n).$$

where p is a non-process fluent. Since A is closed under such rules, if $\bigcup_{1 \leq i \leq m} \alpha(l_i, n) \subseteq A$ then $\alpha(p = y, n) \in A$. From 7.16 it is true that $p = y \in s_n$ whenever $\{l_1, \ldots, l_m\} \subseteq s_n$. We will use a similar line of reasoning if p is a process fluent. Hence, s_n is closed under state constraints of AD.

We must also show that s_n is minimal.

Proof by contradiction.

Assume that $\exists \sigma \subset s_n$ such that $E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n) \subseteq \sigma$ and σ is closed under state constraints of AD.

Let A' be obtained from A by removing atoms of the form $\alpha(l, n)$ such that $l \in s_n \setminus \sigma$. Since $\sigma \subset s_n$, $A' \subset A$.

Both σ and s_n agree upon atoms from $E_{s_{n-1}}(a_{n-1}) \cup (s_{n-1} \cap s_n) \cup T_{s_{n-1}}(s_n)$. Therefore, for every $l \in s_n \setminus \sigma$, $\exists$ a state constraint, r, containing l in the head

$$l \text{ if } l_1, \ldots, l_m$$

such that $\{l_1, \ldots, l_m\} \subseteq s_n$ and $\{l_1, \ldots, l_m\} \not\subseteq \sigma$. From construction of A' it is true that A' does not satisfy the body of $\tau(r)$

$$\alpha(l, n) : -\alpha(l_1, n), \ldots, \alpha(l_m, n).$$

which is the $\mathcal{AC}$ encoding of the state constraint. Hence A' is closed under rules of P^A. This implies that A is not an answer set of P. Contradiction. Therefore, our assumption is false and we conclude that s_n is minimal.

s_n is closed under triggers of AD.

We know that P^A contains the fact $end(n) = t_n$. Since A is closed under such rules, $end(n) = t_n \in A$. From 7.16, $end = t_n \in s_n$. From construction of P, t_n is indeed the end time of s_n. Therefore, $\neg\exists L$ such that L satisfies atleast one trigger of AD and $s_n \setminus L = \{end = t_2\}$ and $L \setminus s_n = \{end = t_1\}$ and $t_1 < t_2$. Therefore, s_n is closed under triggers of AD.

At this point we proved that $M = \langle s_0, a_0, s_1, \ldots, s_n \rangle \in TD(AD)$. We must show that M is a model of Γ_n.

We will show that for every i, $0 \leq i < n$, $a_i = \{a \mid hpd(a, i, t) \in \Gamma_n\}$. P^A contains rules of the form

$$occurs(a, i) : -hpd(a, i, t).$$

Since Γ_n is complete and $\Gamma_n \subseteq A$, we conclude that $occurs(a, i) \in A$ for every $hpd(a, i, t) \in \Gamma_n$. From 7.16 it is true that $a_i = \{a \mid hpd(a, i, t) \in \Gamma_n\}$.

P^A contains rules of the form

$$end(i) = t : -hpd(a, i, t).$$

for every $i \in [0, n)$. Since $\Gamma_n \subseteq A$ and A is closed under such rules, we conclude that $end(i) = t \in A$. From 7.16, $end = t \in s_i$.

P^A contains rules of the form

$$\begin{aligned} : - \ & obs(p, i, t, y), \\ & \neg\alpha(p = y, i). \end{aligned}$$

where p is a non-process fluent and rules of the form

$$\begin{aligned} : - \ & obs(p, i, t, y), \\ & \neg p(i, t, y). \end{aligned}$$

where p is a process fluent and i is an integer from $[0, n]$. Let us consider the first rule. Since A is an answer set of P, if $obs(p, i, t, y) \in \Gamma_n$ then $\neg\alpha(p = y, i) \notin A$. Therefore, the body of every rule containing $\neg\alpha(p = y, i)$ in the head

$$\begin{aligned} \neg\alpha(p = y, i) : - \ & \alpha(p = y_1, i), \\ & y \neq y_1. \end{aligned}$$

is not satisfied by A. Since A is complete we conclude that $\alpha(p = y, i) \in A$. From 7.16, $p = y \in s_i$.

We use a similar argument for the second rule. Since A is an answer set of P, if $obs(p, i, t, y) \in \Gamma_n$ then $\neg p(i, t, y) \notin A$. Therefore, the body of every rule containing $\neg p(i, t, y)$ in the head

$$\begin{aligned} \neg p(i, t, y) : - \ & p(i, t, y_1), \\ & y \neq y_1. \end{aligned}$$

is not satisfied by A. Since A is complete we conclude that $p(i,t,y) \in A$. In other words, $\alpha(p(t) = y, i) \in A$. From 7.16, $p(t) = y \in s_i$. This is possible only if $\exists \lambda T.g(T) \in process$ such that $\lambda T.g(T)(t) = y$ and $p = \lambda T.g(T) \in s_i$.

Therefore, M is a model of Γ_n. $\square$

Given a recorded history upto moment l we proved that the theorem holds for $l = 1$ and then assuming that it holds for $l = n - 1$ we proved that it holds for $l = n$. Therefore, the theorem holds for any arbitrary $l > 0$.

Q.E.D

7.2 Proof of Theorem 6.1

Theorem 6.1 is stated below.

Given a deterministic timed transition table $\mathcal{T}\langle \Sigma, S, S_0, C, E \rangle$,

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \dots$$

is a run of $\mathcal{T}$ over timed word (σ, τ) iff $\langle s'_0, \sigma_1, s'_1, \dots, \dots \rangle$ is a path of $TD(\mathcal{M}(\mathcal{T}))$ such that $\langle s'_0, \sigma_1, s'_1, \dots, \dots \rangle$ is compatible with r.

Proof

left to right

We must show that given a deterministic timed transition table $\mathcal{T}\langle \Sigma, S, S_0, C, E \rangle$ if

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \dots$$

is a run of $\mathcal{T}$ over (σ, τ) then $\exists$ a path $p : \langle s'_0, \sigma_1, s'_1, \dots, \dots \rangle \in TD(\mathcal{M}(\mathcal{T}))$ such that p is compatible with r.

Proof by Induction on the number of elements of (σ, τ) read so far. Let n denote number of elements of (σ, τ) read so far.

Base case:

If $n = 0$ then no symbols from (σ, τ) have been read so far. $\mathcal{T}$ does not have a run

over empty input and therefore the hypothesis of the above statement, *"if r is a run of $\mathcal{T}$"*, is false. Hence the statement is trivially true.

For $n = 1$ we must prove the following.
If $r : \langle s_0, v_0\rangle \longrightarrow^{\sigma_1}_{\tau_1} \langle s_1, v_1\rangle$ is a segment of a run of $\mathcal{T}$ after reading the first element of (σ, τ) then $\langle s'_0, \sigma_1, s'_1\rangle \in TD(\mathcal{M}(\mathcal{T}))$ such that $\langle s'_0, \sigma_1, s'_1\rangle$ is compatible with r.

Since $\langle s'_0, \sigma_1, s'_1\rangle$ is compatible with

$$\langle s_0, v_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1\rangle$$

w.r.t transition $\langle s_0, s_1, \sigma_1, \lambda_1, \delta_1\rangle$ the following is true.

- s'_0 is compatible with $\langle s_0, v_0\rangle$
- $s'_0 \models \mathcal{M}(\delta_1)$
- $s'_0 \models \{state = s_0, start = 0, end = \tau_1\}$
- $s'_1 \models \{state = s_1, start = \tau_1\}$
- For every $x = c \in \lambda_1$, $s'_1 \models \{cur(x) = c, time\_reset(x) = \tau_1\}$
- For every $x \in C$, $s'_1 \models x = \lambda T.(T - t_0) + c$ such that $s'_1 \models \{cur(x) = c, time\_reset(x) = t_0\}$

s'_0 is closed under state constraints and triggers of $\mathcal{M}(\mathcal{T})$ and for every $x \in C$, $s'_0 \models x = \lambda T.(T - t_0) + c$ such that $\lambda T.(T - t_0) + c$ is defined over the interval $[0, \tau_1]$. Therefore, s'_0 is a state of $TD(\mathcal{M}(\mathcal{T}))$

To prove that s'_1 is obtained as result of executing σ_1 in s'_0 we must prove that s'_1 satisfies the McCain-Turner equation.

We know that $\mathcal{M}(\mathcal{T})$ contains dynamic law

$$\sigma_1 \text{ causes } state = s_1 \text{ if } \begin{array}{l} state = s_0, \\ \mathcal{M}(\delta_1) \end{array}$$

For every $x = c \in \lambda_1$, $\mathcal{M}(\mathcal{T})$ contains

$$\sigma_1 \text{ causes } cur(x) = c \text{ if } \begin{array}{l} state = s_0, \\ \mathcal{M}(\delta_1) \end{array}$$

$$\sigma_1 \text{ causes } time\_reset(x) = t_0 \text{ if } \begin{array}{l} end = t_0, \\ state = s_0, \\ \mathcal{M}(\delta_1) \end{array}$$

Therefore, $E_{\sigma_1}(s_0') = \{state = s_1, cur(x) = c, time\_reset(x) = t_0\}$

We know that $s_1' \models end = t$ such that $t > \tau_i$. Therefore, the projection of interval of s_1' w.r.t s_0', $T_{s_0'}(s_1') = \{start = \tau_1, end = t\}$

For every $x \in C$, if x does not appear in λ_1 then it is not reset and its initial value is preserved by inertia. Therefore,

$$s_0' \cap s_1' = \{cur(x) = c, time\_reset(x) = t_0 \mid x \text{ does not appear in } \lambda_1\}$$

Let L be a set of atoms such that

$$L = Cn_Z(E_{\sigma_1}(s_0') \cup (s_0' \cap s_1') \cup T_{s_0'}(s_1'))$$

where Z is the set of state constraints of $\mathcal{M}(\mathcal{T})$. For every $x \in C$, $L \models x = \lambda T.(T - t_0) + c$ such that $L \models \{cur(x) = c, time\_reset(x) = t_0\}$. It is obvious that the set, L, obtained by applying the McCain-Turner equation is the same as s_1'. Therefore, s_1' satisfies the McCain-Turner equation and $\langle s_0', \sigma_1, s_1' \rangle \in TD(\mathcal{M}(\mathcal{T}))$.

Assume that for $n = k$ the following is true. If

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \ldots\ldots \xrightarrow[\tau_k]{\sigma_k} \langle s_k, v_k \rangle$$

is a segment of a run of $\mathcal{T}$ after reading the first k elements of (σ, τ) then $p : \langle s_0', \sigma_1, s_1', \ldots, \ldots, \sigma_k, s_k' \rangle \in TD(\mathcal{M}(\mathcal{T}))$ such that p is compatible with r.

Inductive step: We must prove that for $n = k + 1$ if

$$r' : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \ldots\ldots \xrightarrow[\tau_k]{\sigma_k} \langle s_k, v_k \rangle \xrightarrow[\tau_{k+1}]{\sigma_{k+1}} \langle s_{k+1}, v_{k+1} \rangle$$

is a segment of a run of $\mathcal{T}$ after reading the first $k+1$ elements of (σ, τ) then

$$p' : \langle s'_0, \sigma_1, s'_1, \ldots, \ldots, \sigma_k, s'_k, \sigma_{k+1}, s'_{k+1} \rangle \in TD(\mathcal{M}(\mathcal{T}))$$

such that p' is compatible with r'.

Since p' is the extension of p by $\langle s'_k, \sigma_{k+1}, s'_{k+1} \rangle$ and p is a path of $TD(\mathcal{M}(\mathcal{T}))$, it is enough to prove that $\langle s'_k, \sigma_{k+1}, s'_{k+1} \rangle$ is a transition of $TD(\mathcal{M}(\mathcal{T}))$.

Since p' is compatible with r', the tuple $\langle s'_k, \sigma_{k+1}, s'_{k+1} \rangle$ is compatible with the last segment of r'

$$\langle s_k, v_k \rangle \xrightarrow[\tau_{k+1}]{\sigma_{k+1}} \langle s_{k+1}, v_{k+1} \rangle$$

w.r.t transition $\langle s_k, s_{k+1}, \sigma_{k+1}, \lambda_{k+1}, \delta_{k+1} \rangle$. Thus, the following holds.

- $s'_k \models \{state = s_k, start = \tau_k, end = \tau_{k+1}\}$
- $s'_k \models \mathcal{M}(\delta_{k+1})$
- $s'_{k+1} \models \{state = s_{k+1}, start = \tau_{k+1}\}$
- For every $x = c \in \lambda_{k+1}$, $s'_{k+1} \models \{cur(x) = c, time\_reset(x) = \tau_{k+1}\}$
- For every $x \in C$, $s'_{k+1} \models x = \lambda T.(T - t_0) + c$ such that $s'_{k+1} \models \{cur(x) = c, time\_reset(x) = t_0\}$

Since p is a path of $TD(\mathcal{M}(\mathcal{T}))$, and $s'_k \in p$ we conclude that s'_k is a state of $TD(\mathcal{M}(\mathcal{T}))$

To prove that s'_{k+1} is obtained as result of executing σ_{k+1} in s'_k we must prove that s'_{k+1} satisfies the McCain-Turner equation.

We know that $\mathcal{M}(T)$ contains dynamic law

$$\sigma_{k+1} \text{ causes } state = s_{k+1} \text{ if } \begin{array}{l} state = s_k, \\ \mathcal{M}(\delta_{k+1}) \end{array}$$

For every $x = c \in \lambda_{k+1}$, $\mathcal{M}(T)$ contains

$$\sigma_{k+1} \text{ causes } cur(x) = c \text{ if } \begin{array}{l} state = s_k, \\ \mathcal{M}(\delta_{k+1}) \end{array}$$

$$\sigma_{k+1} \text{ causes } time\_reset(x) = t_0 \text{ if } \begin{array}{l} end = t_0, \\ state = s_k, \\ \mathcal{M}(\delta_{k+1}) \end{array}$$

Therefore, $E_{\sigma_{k+1}}(s'_k) = \{state = s_{k+1}, cur(x) = c, time\_reset(x) = t_0\}$

We know that $s'_{k+1} \models end = t$ such that $t > \tau_{k+1}$. Therefore, the projection of interval of s'_{k+1} w.r.t s'_k, $T_{s'_k}(s'_{k+1}) = \{start = \tau_{k+1}, end = t\}$

For every $x \in C$, if x does not appear in λ_{k+1} then it is not reset and its initial value is preserved by inertia. Therefore,

$$s'_k \cap s'_{k+1} = \{cur(x) = c, time\_reset(x) = t_0 \mid x \text{ does not appear in } \lambda_{k+1}\}$$

Let L be a set of atoms such that

$$L = Cn_Z(E_{\sigma_{k+1}}(s'_k) \cup (s'_k \cap s'_{k+1}) \cup T_{s'_k}(s'_{k+1}))$$

where Z is the set of state constraints of $\mathcal{M}(\mathcal{T})$. For every $x \in C$, $L \models x = \lambda T.(T - t_0) + c$ such that $L \models \{cur(x) = c, time\_reset(x) = t_0\}$. It is obvious that the set, L, obtained by applying the McCain-Turner equation is the same as s'_{k+1}. Therefore, s'_{k+1} satisfies the McCain-Turner equation and $\langle s'_k, \sigma_{k+1}, s'_{k+1} \rangle \in TD(\mathcal{M}(\mathcal{T}))$. The proof follows by induction.

At this point we have shown that if r is a run of $\mathcal{T}$ then there exists a path in $TD(\mathcal{M}(\mathcal{T}))$ that is compatible with r.

Right to left

We must show that given a deterministic timed transition table $\mathcal{T}\langle \Sigma, S, \{s_0\}, C, E \rangle$

and timed word (σ, τ) if $p : \langle s'_0, \sigma_1, s'_1, \ldots, \ldots \rangle$ is a path of $TD(\mathcal{M}(\mathcal{T}))$ that is compatible with

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, v_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \ldots$$

then r is a run of $\mathcal{T}$ over (σ, τ).

Proof by Induction on the length of path p. Let n denote the length of p.

Base case:
If $n = 0$ there are no transitions and the only state we have is s'_0. We are given that s'_0 is compatible with $\langle s_0, v_0 \rangle$. Therefore,

- For every $x \in C$, $s'_0 \models \{cur(x) = 0, time\_reset(x) = 0\}$
- For every $x \in C$, $s'_0 \models x = \lambda T.(T - t_0) + c$ such that $s'_0 \models \{cur(x) = c, time\_reset(x) = t_0\}$
- For every $x \in C$, $s'_0 \models x(start) = c$ iff $v_0(x) = c$

Since $s_0 \in S_0$ and for every $xinC$, $s'_0 \models x(start) = v_0(x) = 0$ we conclude that $\langle s_0, v_0 \rangle$ is the initial extended state for any run of $\mathcal{T}$ over (σ, τ).

For $n = 1$ we must prove that if $\langle s'_0, \sigma_1, s'_1 \rangle$ is a transition of $TD(\mathcal{M}(\mathcal{T}))$ compatible with

$$r : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle$$

then r is a segment of a run of $\mathcal{T}$ after reading the first element of (σ, τ).

We have already concluded that $\langle s_0, v_0 \rangle$ is the initial extended state for any run of $\mathcal{T}$. Since $\langle s'_0, \sigma_1, s'_1 \rangle$ is compatible with r w.r.t transition $\langle s_0, s_1, \sigma_1, \lambda_1, \delta_1 \rangle$ the following is true.

- $s'_0 \models \mathcal{M}(\delta_1)$
- For every $x \in C$, $s'_0 \models x(start) = c$ iff $v_0(x) = c$ and $s'_1 \models x(start) = c$ iff $v_1(x) = c$

In order to prove that r is a segment of a run we must show that δ_1 is satisfied by the clock interpretation $v_0 + \tau_1 - \tau_0$ and $v_1 = [\lambda_1 \rightarrow c](v_0 + \tau_1 - \tau_0)$.

Since we are dealing with clock variables that keep ticking at the same rate the following is true. For all $x \in C$, $s'_0 \models x(end) = x(start) + \tau_1 - \tau_0 = y$ where $s'_0 \models \{start = \tau_0 = 0, end = \tau_1\}$. Since $x(start)$ is the same as $v_0(x)$ we get, for all $xinC$, $s'_0 \models x(end) = v_0(x) + \tau_1 - \tau_0$.

Now replace occurrences of $x(end)$ in $\mathcal{M}(\delta_1)$ by $v_0(x)+\tau_1-\tau_0$. The resulting expression $\delta_1^{v_0+\tau_1-\tau_0}$ is the evaluation of δ_1 using clock values given by interpretation $v_0+\tau_1-\tau_0$. Since $s'_0 \models \mathcal{M}(\delta_1)$ $v_0 + \tau_1 - \tau_0$ satisfies δ_1.

Next, for every $x = c \in \lambda_1$, $s'_1 \models \{cur(x) = c, time\_reset(x) = \tau_1\}$. Therefore, $s'_1 \models x(start) = v_1(x) = \lambda T.(T - \tau_1) + c(\tau_1) = c$.

For all x that do not appear in λ_1, $s'_1 \models \{cur(x) = 0, time\_reset(x) = 0\}$. Therefore, $s'_1 \models x(start) = v_1(x) = \lambda T.(T-0)+0(\tau_1) = \tau_1$. Now consider the clock interpretion $v_0+\tau_1-\tau_0$. Since $\tau_0 = 0$ and for any x, $v_0(x) = 0$ we have $v_0(x)+\tau_1-\tau_0 = \tau_1$. Thus, for variables that do not appear in λ_1, v_1 agrees with $v_0 + \tau_1 - \tau_0$.

Hence we conclude that v_1 is the interpretation $[\lambda_1 \rightarrow c](v_0 + \tau_1 - \tau_0)$.

Since both requirements are met we conclude that

$$r : \langle s_0, v_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1\rangle$$

is a segment of a run of $\mathcal{T}$ after reading the first element of (σ, τ).

Assume that for $n = k$ the following is true.
If $p : \langle s'_0, \sigma_1, s'_1, \ldots, \sigma_k, s'_k\rangle$ is a path of $TD(\mathcal{M}(\mathcal{T}))$ compatible with

$$r : \langle s_0, v_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1\rangle \ldots\ldots \xrightarrow[\tau_k]{\sigma_k} \langle s_k, v_k\rangle$$

then r is a segment of a run of $\mathcal{T}$ after reading the first k elements of (σ, τ).

Inductive step: We must prove that for $n = k + 1$ if

$$p' : \langle s'_0, \sigma_1, s'_1, \ldots, \ldots, \sigma_k, s'_k, \sigma_{k+1}, s'_{k+1}\rangle$$

is path of $TD(\mathcal{M}(\mathcal{T}))$ compatible with

$$r' : \langle s_0, v_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, v_1 \rangle \ldots\ldots \xrightarrow[\tau_k]{\sigma_k} \langle s_k, v_k \rangle \xrightarrow[\tau_{k+1}]{\sigma_{k+1}} \langle s_{k+1}, v_{k+1} \rangle$$

then r' is a segment of a run of $\mathcal{T}$ after reading the first $k+1$ elements of (σ, τ).

Since r is already a segment of a run of T and r' extends r by

$$e : \langle s_k, v_k \rangle \xrightarrow[\tau_{k+1}]{\sigma_{k+1}} \langle s_{k+1}, v_{k+1} \rangle$$

it is enough to prove that e is a segment of a run of $\mathcal{T}$ after reading the $k+1$th element of (σ, τ).

Since p' is compatible with r', the tuple $\langle s'_k, \sigma_{k+1}, s'_{k+1} \rangle$ is compatible with e w.r.t transition $\langle s_k, s_{k+1}, \sigma_{k+1}, \lambda_{k+1}, \delta_{k+1} \rangle$. Thus, the following is true.

- $s'_k \models \mathcal{M}(\delta_{k+1})$
- For every $x \in C$, $s'_k \models x(start) = c$ iff $v_k(x) = c$ and $s'_{k+1} \models x(start) = c$ iff $v_{k+1}(x) = c$

In order to prove that e is a segment of a run we must show that δ_{k+1} is satisfied by the clock interpretation $v_k + \tau_{k+1} - \tau_k$ and $v_{k+1} = [\lambda_{k+1} \rightarrow c](v_k + \tau_{k+1} - \tau_k)$.

Since we are dealing with clock variables that keep ticking at the same rate the following is true. For all $x \in C$, $s'_k \models x(end) = x(start) + \tau_{k+1} - \tau_k = y$ where $s'_k \models \{start = \tau_k, end = \tau_{k+1}\}$. Since $x(start)$ is the same as $v_k(x)$ we get, for all $xinC$, $s'_k \models x(end) = v_k(x) + \tau_{k+1} - \tau_k$.

Now replace occurrences of $x(end)$ in $\mathcal{M}(\delta_{k+1})$ by $v_k(x) + \tau_{k+1} - \tau_k$. The resulting expression $\delta_{k+1}^{v_k + \tau_{k+1} - \tau_k}$ is the evaluation of δ_{k+1} using clock values given by interpretation $v_k + \tau_{k+1} - \tau_k$. Since $s'_k \models \mathcal{M}(\delta_{k+1})$ $v_k + \tau_{k+1} - \tau_k$ satisfies δ_{k+1}.

Next, for every $x = c \in \lambda_{k+1}$, $s'_{k+1} \models \{cur(x) = c, time\_reset(x) = \tau_{k+1}\}$. Therefore, $s'_{k+1} \models x(start) = v_{k+1}(x) = \lambda T.(T - \tau_{k+1}) + c(\tau_{k+1}) = c$.

Every variable x that does not appear in λ_{k+1} is not reset. So the value of x at the end of s'_k is carried over to the start of s'_{k+1} by inertia. Therefore, $s'_{k+1} \models x(start) = v_{k+1}(x) = y$ such that $s'_k \models x(end) = y$. We know that $s'_k \models x(end) = v_k(x) + \tau_{k+1} - \tau_k$. Substituting we get $s'_{k+1} \models x(start) = v_{k+1}(x) = v_k(x) + \tau_{k+1} - \tau_k$. Thus, for variables that do not appear in λ_{k+1}, v_{k+1} agrees with $v_k + \tau_{k+1} - \tau_k$.

Hence we conclude that v_{k+1} is the interpretation $[\lambda_{k+1} \rightarrow c](v_k + \tau_{k+1} - \tau_k)$.

Since both requirements are met we conclude that e is a segment of $\mathcal{T}$ after reading the $k+1$th element of (σ, τ). Therefore, r' is a segment of a run of $\mathcal{T}$ after reading the first $k+1$ elements of (σ, τ). The proof follows by induction.

At this point we have shown that if p is a path of $TD(\mathcal{M}(\mathcal{T}))$ that is compatible with r then r is a run of $\mathcal{T}$.

Q.E.D

CHAPTER VIII
CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this dissertation we use an action language-logic programming approach to building intelligent agents acting in hybrid domains. Our approach is based on representing knowledge of the agent in some language then translating this knowledge into a logic program and computing models of the logic program. Thus, reducing various tasks of the agent to asking questions about models of logic programs. To represent knowledge of an agent acting in a hybrid domain we developed a new action language H which extends $\mathcal{AL}$ with the ability to reason about continuous change. With this new language we are able to

- elegantly describe hybrid domains.
- write elaboration tolerant action descriptions.
- model a variety of domains which shows that it is a good knowledge representation language.
- write concise and simpler action descriptions compared to other approaches.

The language is based on transition diagram based semantics, much like $\mathcal{AL}$, which allows us to distinguish between states and transitions. We developed a methodology for representing non-trivial examples in this language.

We implemented action descriptions of H by translating them into $\mathcal{AC}$ programs and computing answer sets of the resulting $\mathcal{AC}$ programs. In this way, various tasks of the agent can be reduced to asking questions about answer sets of $\mathcal{AC}$ programs. We provided an encoding of action descriptions of H into $\mathcal{AC}$ programs and proved that this encoding is correct. We used existing solvers such as EZCSP and Luna to compute answer sets of our encodings. We encoded prediction and planning problems under both systems and were able to compute answer sets in a reasonable amount

of time. Thanks to our encoding and new solvers we are able to solve problems that could not be solved in the past.

We studied timed automata and discovered that H is more general and expressive than timed automata. We proved that for every deterministic timed transition table there is an equivalent action description of H.

In this dissertation we did not describe how to apply our approach to the agent loop. But an important prerequisite to executing the loop, *representing knowledge of the agent*, has been discussed in detail in this dissertation. We were able to confirm that answer sets of our translations may encode plans to achieve goals of an agent.

8.2 Future Work

Language H is just one step towards representing knowledge of an agent. There are several other features of a domain that are very interesting from a knowledge representation point of view. Some of the features we are interested in are mentioned below.

- We would like to describe non-deterministic effects of actions. From [6] an action description with state constraints is capable of describing non-determinism. However, we would like to express non-determinism as a direct consequence of an action. For example, when a six-sided die is rolled there are six possible outcomes. We would like to express the effect of this action using a dynamic law

  $$roll \text{ causes } die = \{1, 2, 3, 4, 5, 6\}$$

  which states that rolling a die can lead to one of possible six successor states. Languages which are capable of describing such effects already exist. We would like to investigate these languages to understand their approach.

- We showed how to use language H to reason about resources. However, there is no proper methodology for reasoning about resources in general. We would

like to develop a methodology for discrete domains and later extend it to hybrid domains.

It is clear that solvers play a vital role in the development of agents. Solvers can be improved in several ways.

- We would like to test whether existing solvers can solve complex planning problems in H. So far we are able to run small size planning problems in H but in complex planning problems the scheduling of actions must be taken into account to determine whether a plan is feasible. Depending on the results of our tests we can suggest improvements to the solvers.
- We would like to enhance the language of existing solvers to make them more suitable for knowledge representation.
- The efficiency of solvers can be further improved by running industrial size applications of H.

BIBLIOGRAPHY

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP'09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*, July 2009.

[3] M. Balduccini and M. Gelfond. Diagnostic reasoning with A-Prolog. *TPLP*, 3(4-5):425–461, 2003.

[4] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.

[5] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31(1-3):85–117, 1997.

[6] C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.

[7] C. Baral, T. Son, and L. Tuan. A transition function based characterization of actions with delayed and continuous effects. In *Proc. of KR-02*, pages 291–302, 2002.

[8] S. Brass and J. Dix. A characterization of the stable semantics by partial evaluation. In *Proc. of the 10th Workshop on Logic Programming, Zurich*, October 1994.

[9] S. Chintabathina, M. Gelfond, and R. Watson. Modeling hybrid domains using process description language. In *Proc. of ASP-05*, pages 303–317, 2005.

[10] S. Chintabathina, M. Gelfond, and R. Watson. Defeasible laws, parallel actions, and reasoning about resources. In *Proc. of CommonSense'07*, pages 35–40. AAAI Press, 2007.

[11] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. *Lecture Notes in Computer Science*, 1861:807–821, 2000.

[12] E. Erdem and A. Gabaldon. Cumulative effects of concurrent actions on numeric-valued fluents. In *Proc. of AAAI-05*, pages 627–632, 2005.

[13] E. Erdem and A. Gabaldon. Representing action domains with numeric-valued fluents. In *Proc. of JELIA-06*, pages 151–163, 2006.

[14] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. of JELIA-04*, volume 3229, pages 200–212. Springer-Verlag, 2004.

[15] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of ICLP-88*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[16] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[17] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[18] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.

[19] M. Gelfond and T. C. Son. Reasoning with prioritized defaults. *In J.Dix, L. M. Pereira, T. Przymusinski eds, Lecture Notes in Artificial Intelligence, 1471*, pages 164–224, 1998.

[20] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.

[21] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, pages 623–630, 1998.

[22] R. K. Guy. *Unsolved problems in number theory.* New York; Berlin: Springer-Verlag, c1981, first edition, 1981.

[23] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[24] J. Lee and V. Lifschitz. Describing additive fluents in action language C+. In *Proc. of IJCAI-03*, pages 1079–1084, 2003.

[25] J. Lee and V. Lifschitz. A knowledge module: Buying and selling. In *Working Notes of the AAAI Symposium on Formalizing Background Knowledge*, pages 28–32, 2006.

[26] V. Lifschitz. Missionaries and cannibals in the causal calculator. In Anthony G. CohnFausto GiunchigliaBart Selman, editor, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 85–96, San Francisco, 2000. Morgan Kaufmann.

[27] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[28] V. Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

[29] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. of IJCAI-95*, pages 1985–1991, 1995.

[30] W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[31] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In C. Mellish, editor, *Proc. of IJCAI-95*, pages 1978–1984. Morgan Kaufmann, 1995.

[32] N. McCain and H. Turner. Causal theories of action and change. In *Proc. of AAAI-97*, pages 460–465, 1997.

[33] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[34] V. Mellarkod. *Integrating ASP and CLP Systems: Computing Answer Sets from Partially Ground Programs*. Texas Tech University, 2007.

[35] V. Mellarkod and M. Gelfond. Enhancing asp systems for planning with temporal constraints. In *Proc. of LPNMR 2007*, pages 309–314, May 2007.

[36] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating Answer Set Programming and Constraint Logic Programming. In *Proc. of ISAIM'08*, http://isaim2008.unl.edu/index.php, 2008.

[37] A. Ricardo Morales. *Improving Efficiency of Solving Computational Problems with ASP*. PhD Dissertation. Texas Tech University, December 2010.

[38] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[39] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[40] M. Shanahan. *Solving the frame problem*. MIT Press, 1997.

[41] Y. Shoham. Nonmonotonic reasoning and causation. *Cognitive Science*, 14(2):213–252, 1990.

[42] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31(1-3):245–298, 1997.

[43] N. Vinogradova and L.G.Blaine. Exploring the mathematics of bouncing balls. *Mathematics teacher*, pages 192–198, October 2010.

[44] Y. Zhang. Handling defeasibilities in action domains. *TPLP*, 3(3):329–376, 2003.

APPENDIX A

```
% EZCSP code encoding the bouncing ball example from chapter 3.
% We are able to do projection in the presence of triggers and also translate triggers.
% Value of a process fluent will be determined at the end of a state
% depending upon the preconditions and the function associated with it.
% Unable to encode ASP predicates dependent on constraints directly.
% However, found alternative ways that are equivalent.
%*****
% The domain is real numbers
cspdomain(r).
% Defining sorts
% various positions of the ball
pos(ascending).
pos(stationary).
pos(descending).
% Steps of trajectory
step(0).
step(1).
step(2).
step(3).
step(4).
step(5).
step(6).
% Actions in the domain
action(drop).
action(catch).
action(dummy).
action(bounce).
action(fall).
% Fluents
% 1. status ( a multi valued fluent)
fluent(status).
range(status,B) :- pos(B).
% 2. height
cspvar(h(I),0,500) :- step(I).
% 3. ht_changed
cspvar(ch(I),0,500) :- step(I).
% 4. time_changed
cspvar(t(I),0,60) :- step(I).
% 5. velocity
cspvar(v(I),0,200) :- step(I).
% 6. v_changed
```

```
cspvar(cv(I),0,200) :- step(I).
% Declaration of start and end as CSP variables
cspvar(start(I),0,60) :- step(I).
cspvar(end(I),0,60) :- step(I).
% General axioms
% Axioms for start
required(start(0)==0).
required(start(I1)==end(I)) :- step(I),
                               step(I1),
                               I1 = I + 1.
% Inertia for Fluents ( status is the only non-real fluent here)
v(X,F,I1) :- v(X,F,I),
             step(I),
             step(I1),
             I1 = I + 1,
             fluent(F),
             range(F,X),
             not¬v(X,F,I1).
¬v(X,F,I):- v(Y,F,I),
            X! = Y,
            range(F,X),
            range(F,Y),
            fluent(F),
            step(I).
% Inertia axioms for "ht_changed" and "time_changed" are needed.
% Drop, catch and fall effect ht_changed and time_changed
ab(I) :- occurs(drop,I), step(I).
ab(I) :- occurs(catch,I), step(I).
ab(I) :- occurs(fall,I), step(I).
% Bounce effects v_changed and time_changed
ab1(I) :- occurs(bounce,I), step(I).
% Inertia axiom for "ht_changed"
required(ch(I1)==ch(I)) :- not ab(I),
                           step(I),
                           step(I1),
                           I1 = I + 1.
% Inertia axiom for "time_changed"
required(t(I1) == t(I)) :- not ab(I),
                           not ab1(I),
                           step(I),
                           step(I1),
                           I1 = I + 1.
% Inertia axiom for "v_changed"
```

```
required(cv(I1) ==cv(I)) :- not ab1(I),
                            step(I),
                            step(I1),
                            I1 = I + 1.
% Inertia for "height" and "velocity" are not needed because they are defined fluents.
% Causal laws
% 1. Effects of "drop" and "catch"
% a. status
% "drop causes status= descending"
v(descending,status,I+1) :- occurs(drop,I),step(I).
% "impossible drop if status=descending"
-occurs(drop,I) :- v(descending,status,I),step(I).
% "impossible drop if status=ascending"
-occurs(drop,I) :- v(ascending,status,I), step(I).
% "catch causes status=stationary"
v(stationary,status,I+1) :- occurs(catch,I),step(I).
% "impossible catch if status=stationary"
-occurs(catch,I) :- v(stationary,status,I),step(I).
% b. ht_changed
% "drop causes ht_changed=X if height(end)=X"
required(ch(I1)== h(I)) :- occurs(drop,I),step(I1), step(I), I1 = I+1.
% "catch causes ht_changed=X if height(end)=X"
required(ch(I1) == h(I) ):- occurs(catch,I), step(I), step(I1), I1=I+1.
% "impossible drop if height(end)=0"
required(h(I) > 0 ):- occurs(drop,I), step(I).
% "impossible catch if height(end)=0 "
required(h(I) > 0 ) :- occurs(catch,I), step(I).
% c. time_changed
% "drop causes time_changed=T0 if end=T0"
required(t(I1)==end(I)) :- occurs(drop,I), step(I1), step(I), I1=I+1.
% "catch causes time_changed=T0 if end=T0"
required(t(I1)==end(I)) :- occurs(catch,I), step(I1), step(I), I1=I+1.

% 2. Effects of bounce
% a. v_changed
% "bounce causes v_changed = X*0.8 if velocity(end)=X"
required(cv(I1) == 8*v(I)/10 ) :-  occurs(bounce, I),
                                   step(I),
                                   step(I1),
                                   I1 = I + 1.
% b. status
% "bounce causes status=ascending"
v(ascending,status,I+1) :- occurs(bounce,I), step(I).
```

```
% c. time_changed
% "bounce causes time_changed=T0 if end=T0"
required(t(I1)== end(I)):- occurs(bounce,I), step(I1), step(I), I1=I+1.
% "impossible bounce if status=ascending"
-occurs(bounce,I) :- v(ascending,status,I), step(I).
% "impossible bounce if status=stationary"
-occurs(bounce,I) :- v(stationary,status,I), step(I).
% "impossible bounce if velocity(end)=0"
required( v(I) > 0 ) :- occurs(bounce,I), step(I).

% 3. Effects of fall
% "fall causes status=descending"
v(descending,status,I+1) :- occurs(fall,I), step(I).
% fall causes ht_changed=X if height(end)=X
required(ch(I1) == h(I)):- occurs(fall,I), step(I), step(I1), I1=I+1.
% "fall causes time_changed=T0 if end=T0"
required(t(I1)==end(I)) :- occurs(fall,I), step(I), step(I1), I1=I+1.
% "impossible fall if status = descending"
-occurs(fall,I) :- v(descending,status,I), step(I).
% "impossible fall if status=stationary"
-occurs(fall,I) :- v(stationary,status,I), step(I).
% "impossible fall if height(end)=0"
required( h(I)>0 ) :- occurs(fall,I), step(I).

% 4. Velocity
% velocity = λT.9.8 * (T − T0) if status=descending, time_changed=T0
required(v(I) == 98*(end(I)-t(I))/10 ) :-   v(descending, status, I),
                                            step(I).
% velocity = λT.0 if status=stationary
required(v(I) == 0) :- v(stationary,status,I), step(I).
% velocity = λT.max(0, X − 9.8 * (T − T0)) if  status = ascending,
                                               %v_changed = X,
                                               %time_changed = T0
required(v(I)== max(0,cv(I)-98*(end(I)-t(I))/10) ) :-
      v(ascending, status, I),
      step(I).
% 5. height
% We will be defining the value of height at the end of every step.
% height = λT.max(0, X − 4.9 * (T − T0)^2) if  ht_changed = X,
                                               %time_changed = T0,
                                               %status = descending
```

```
 required(h(I) == max(0, ch(I) − 49 ∗ (end(I) − t(I)) ∗ (end(I) − t(I))/10)) : −
        v(descending, status, I),
        step(I).
% height = λT.X if ht_changed=X, status= stationary
required(h(I) == ch(I)) :- v(stationary,status,I), step(I).
% height = λT.X ∗ (T − T0) − 4.9 ∗ (T − T0)^2 if  v_changed = X,
                                                  %time_changed = T0,
                                                  %status = ascending
required(h(I)==cv(I)*(end(I)-t(I))-49*(end(I)-t(I))*(end(I)-t(I))/10):-
         v(ascending, status, I),
         step(I).
% TRIGGERS
% status=descending, height(end)=0 triggers bounce
1{p(I),q(I)}1:- v(descending,status,I), step(I).
required(end(I) == pow(2*ch(I)*10/98,1/2) + t(I) ) :- p(I), step(I).
required(end(I) < pow(2*ch(I)*10/98,1/2) + t(I) ) :- q(I), step(I).

% If height at end==0 and status=descending then bounce occurs
occurs(bounce,I) :-  required(end(I) == pow(2 ∗ ch(I) ∗ 10/98, 1/2) + t(I)),
                     step(I),
                     v(descending, status, I).
% Similarly for fall
1{fall(I),nfall(I)}1 :- v(ascending,status,I), step(I).
% fall time = time it hits ground + u/g
required(end(I)== cv(I)*10/98 + t(I) ) :- fall(I), step(I).
required(end(I) < cv(I)*10/98 + t(I) ) :- nfall(I), step(I).

% status=ascending, velocity(end)=0 triggers fall
occurs(fall,I) :-  required(end(I) == cv(I) ∗ 10/98 + t(I)),
                   step(I),
                   v(ascending, status, I).
% HISTORY
% Initial values of height and velocity
required(ch(0)==500).
required(cv(0)==0).
% Initial value of other fluents
required(t(0)==0).
v(stationary,status,0).
occurs(drop,0).
required(end(0)==5).
occurs(catch,1).
required(end(1)==10).
```

```
occurs(drop,2).
required(end(2)==20).
occurs(dummy,3).
required(end(3)==25).
occurs(catch,6).
required(end(6)==40799/1000).
```

% End of Program

% Answer set returned by EZCSP(only showing relevant atoms)

v(stationary,status,0) v(descending,status,1) v(stationary,status,2)
v(descending,status,3) v(descending,status,4) v(ascending,status,5)
v(descending,status,6) v(stationary,status,7) occurs(drop,0) occurs(catch,1)
occurs(drop,2) occurs(dummy,3) occurs(bounce,4) occurs(fall,5) occurs(catch,6)
ch(0)=500.0 ch(1)=500.0 ch(2)=377.5 ch(3)=377.5 ch(4)=377.5 ch(5)=377.5
ch(6)=241.6 cv(0)=0.0 cv(1)=0.0 cv(2)=0.0 cv(3)=0.0 cv(4)=0.0
cv(5)=68.81 cv(6)=68.81 end(0)=5.0 end(1)=10.0 end(2)=20.0 end(3)=25.0
end(4)=28.77 end(5)=35.79 end(6)=40.79 h(0)=500.0 h(1)=377.5 h(2)=377.5
h(3)=255.0 h(4)=0.0 h(5)=241.60 h(6)=119.10 start(0)=0.0 start(1)=5.0
start(2)=10.0 start(3)=20.0 start(4)=25.0 start(5)=28.77 start(6)=35.79 t(0)=0.0
t(1)=5.0 t(2)=10.0 t(3)=20.0 t(4)=20.0 t(5)=28.77 t(6)=35.79 v(0)=0.0 v(1)=49.0
v(2)=0.0 v(3)=49.0 v(4)=86.01 v(5)=0.0 v(6)=48.99

APPENDIX B

```
% Luna encoding of the brick drop example from chapter 2.
% Given an initial situation and a sequence of actions
% we are able to predict the values of fluents.
% We begin with defining sorts

% Regular sorts
% a.Boolean values
b_val(true).
b_val(false).
#domain b_val(B;B1).

% b.Actions
action(drop).
action(catch).
#domain action(A).

% c.Fluents
fluent(holding).
range(holding,B).

% d.Indices of the trajectory
const n=2.
step(0..n).
#domain step(I;I1;I2).
next(I,I+1):- I<n.

% Constraint sorts
% a.time sort
#csort time(0..60).
% b.meters
#csort meters(0..500).

% Declaration of mixed predicates
#mixed start(step,time).
#mixed end(step,time).
%mixed predicates for ht_changed, time_changed and height
#mixed ch(step,meters).
#mixed tc(step,time).
#mixed height(step,meters).
% Domain Independent axioms
% Axioms to define start and end times of a step
```

```
% Start time of step 0 is 0
<∼ start(0,T), T!=0.
% start of I+1 defined in terms of end of I
<∼   start(I1,T1),
     next(I,I1),
     end(I,T),
     T! = T1.
% Uniqueness axiom
% A fluent has a unique value in every step
¬val(Y1,P,I) :-  val(Y2,P,I),
                 Y1! = Y2,
                 fluent(P),
                 range(P,Y1),
                 range(P,Y2).
% Inertia axiom
val(Y,P,I1) :-  val(Y,P,I),
                not ¬val(Y,P,I1),
                next(I,I1),
                I < n,
                fluent(P),
                range(P,Y).
ab(I) :- occurs(drop,I).
ab(I) :- occurs(catch,I).
¬ab(I) :- not  ab(I).
% Inertia for ht_changed and time_changed
<∼   ch(I,X),
     next(I,I1),
     ch(I1,X1),
     I < n,
     ¬ab(I),
     X! = X1.
<∼   tc(I,T),
     tc(I1,T1),
     next(I,I1),
     I < n,
     ¬ab(I),
     T! = T1.
% Height is well defined so no need to write inertia axiom for it
% Domain Dependent axioms
% 1. Effects of "drop" and "catch"
% 1a. holding
% drop causes ¬holding
val(false,holding,I+1) :- occurs(drop,I).
```

```
% impossible drop if ¬holding
¬occurs(drop,I) :- val(false,holding,I).
% catch causes holding
val(true,holding,I+1) :- occurs(catch,I).
% impossible catch if holding
¬occurs(catch,I) :- val(true,holding,I).
% 1b. ht_changed
% drop causes ht_changed=X if height(end)=X
<∼  occurs(drop, I),
    height(I, X),
    next(I, I1),
    I < n,
    ch(I1, X1),
    X1! = X.
% catch causes ht_changed if height(end) = X
<∼  occurs(catch, I),
    height(I, X),
    next(I, I1),
    I < n,
    ch(I1, X1),
    X1! = X.
% 1c.time_changed
% drop causes time_changed= T0 if end=T0
<∼  occurs(drop, I),
    end(I, T),
    next(I, I1),
    tc(I1, T1),
    T! = T1.
% catch causes time_changed=T0 if end=T0
<∼  occurs(catch, I),
    end(I, T),
    next(I, I1),
    tc(I1, T1),
    T! = T1.
%2. Definition of height
% We use middle rules to define height at end of each step
```

% height $= \lambda T.X - 4.9 * (T - T0)^2$ if $ht\_changed = X$,
$\%time\_changed = T0$,
$\%\neg holding$.

```
compute_ht(I)<∼  val(false,holding,I),
                 ch(I,X),
                 tc(I,T),
                 end(I,T1),
                 height(I,X1),
                 d(X,X1,T,T1).
% Forcing to compute height
¬compute_ht(I):- not  compute_ht(I).
:- ¬compute_ht(I), val(false,holding,I).
% height = λT.X if ht_changed=X, holding.
<∼   val(true,holding,I),
     ch(I,X),
     height(I,X1),
     X1!=X.
% defined rules
{:  d(X,X1,T,T1) <- X1 = max(0,X - 4.9*(T1-T)*(T1-T)). :}
% Executablity conditions involving "height"
% "impossible drop if height(end)=0"
<∼ occurs(drop,I), height(I,X), X==0.
% "impossible catch if height(end)=0"
<∼ occurs(catch,I), height(I,X), X==0.

%History
% Initially the agent is not holding the ball
val(false,holding,0).
% The height is 500 units
<∼ ch(0,X), X!=500.
% Initial value of time_changed
<∼ tc(0,T), T!=0.
% Sequence of actions
occurs(catch,0).
<∼ end(0,T), T!= 2.
occurs(drop,1).
<∼ end(1,T),T!=5.
occurs(catch,2).
<∼ end(2,T), T!=10.

% End of Program
```