

Languages and Theories for Systematically Designing and Analyzing Autonomous Agents

Chitta Baral

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85233, USA
chitta@asu.edu

Tran Cao Son

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

August 8, 2007

Chapter 1

Introduction and Motivation

In recent years the word ‘agent’ has been used or perhaps overused with respect to many computer programs in various different contexts. In the field of artificial intelligence it has been used with qualifiers such as ‘rational’ and ‘autonomous’. In this paper we discuss a systematic approach for the design and analysis of *autonomous agents* based on recent research in reasoning about actions, goal specification and agent execution languages. To motivate our interpretation of the phrase “autonomous agents” let us start with the dictionary meanings of the terms ‘agent’ and ‘autonomy’ and interpret them in the context of Artificial Intelligence, Computer Science and Asimov’s laws of Robotics.

For the word *agent*, the American Heritage dictionary of the English language gives the meanings (i) one that acts or has the power or authority to act, (ii) one empowered to act for or represent another, (iii) a means by which something is done or caused, and (iv) a force or substance that causes a change. The Internet resource Wordnet gives the meanings (v) an active and efficient cause; capable of producing a certain effect, and (vi) a substance that exerts some force or effect

For the word *autonomous*, the American Heritage dictionary of the English language gives the meanings (i) not controlled by others or by outside forces, (ii) independent in mind or judgement or government; self directed. Wordnet gives the meanings (iii) of political bodies [syn: independent, self-governing, sovereign], (iv) existing as an independent entity and (v) of persons; free from external control and constraint in.

In our context, an artificial computer-driven agent is an entity that acts and that causes change. Since we would like to have some control over the agent, lest it makes us its slave, the qualifier ‘autonomous’ in the context of an artificial computer-driven agent means that its actions are not micro-managed by other entities, but some humans do have control over it. In other words an autonomous artificial computer-driven agent, which we will simply refer henceforth as an *autonomous agent*, can take high-level directives and goals and can figure out what actions it needs to take and execute those actions.

To be able to do this, among other things, an autonomous agent must understand high level directives, must know what actions it can do and what changes these action would cause, must have knowledge about the environment it is in and how it might change or react to its actions, must be able to observe the world (what is true when, what actions occurred when, etc.), must be able to analyze and assimilate those observations so as to make conclusions that it did not directly observe, must be able to contemplate the impact of its actions, must be able to predict how the world might evolve, must be able to contemplate what went wrong when the world does not evolve as the agent expected it to, must be able to make plans

to make the world evolve in a particular way (as directed, or so as to achieve its goal), execute those plans, and modify those plans if interfering actions beyond its control happen (i.e., the environment does not co-operate), and must be able to learn from its interaction with the environment.

In recent years a lot of research has been done with respect to each of the above aspects. In this paper we aim to connect these research into a single framework. To start with let us analyze and elaborate what technical formulations, and algorithms are needed to be able to characterize and correctly implement the various abilities mentioned in the previous paragraph.

1.1 Specifying actions and their effects

To start with we will need to specify what actions an agent can execute in what state of the world, and what its impact on the world would be. Since the number of states of the world is often exponential in the size of the properties of the world, we can not just explicitly specify a table that will tell us the transition between states of the world due to actions. In this paper we will present a brief evolution of various languages for such specification of action and their effects.

1.2 Specifying observations and figuring out the current state of the world

The specifications in the previous sub-section will tell us the transition between states due to actions and possible evolutions of the world. When an agent is situated in a world, it may have complete information (in terms of which fluents¹ are true and which are false) about a particular initial state. But after that the agent executes some actions, other actions are executed by other agents or the environment, and the agent may observe only some of the happenings and may have sensed some of the fluent values at some time points. Based on its partial knowledge the agent needs to periodically figure out its current state so as to evaluate if the world is evolving the way the agent desired it to and if not the agent can then try to change its course of actions to steer the evolution to a desired path.

Here the specification of various kinds of observations is syntactically simple, but analyzing these observations to narrow down the current state, and how the world progressed to it (from the initial state) is challenging. We will present research that have been done on this.

1.3 Expressing directives and Goals

The previous two subsections were about the specification and analysis that an agent needs to make to figure out how the world has evolved and may evolve in the future. As we mentioned earlier, we envision our ‘autonomous’ agent to be ready to take directives from a human. Such directives will specify *how the world should evolve*. To express such directives, as well as for the agent to express its own goals in a declarative manner we need a goal or directive specification language. Since specifying how the world should evolve means specifying the set of acceptable or desired trajectories (of the world), a starting point for a goal specification language are temporal logics, such as LTL and CTL*, that are used in program specification. Although these languages have many useful connectives such as \diamond , \square , \mathcal{U} , A , and E meaning, *eventually*, *always*, *until*, *for all paths*, and *exist path* respectively, many important

¹By fluents we mean dynamic properties of the world.

directives can not be expressed using these languages. We will discuss recent advances in languages to express richers goals and directives.

1.4 Sensing actions and partial observability

Often an agent does not have sensors that can sense all the fluents all the time. In fact certain fluents can not be sensed directly and their values need to be inferred from other sensor values. Certain sensors can only be operated when some specific conditions are satisfied. Finally, even if an agent can sense all the fluents all the time, that may be too time consuming as well as expensive. Thus often an agent will have incomplete knowledge about the world it is in. Thus an agent needs to be able to characterize transitions between its knowledge of the world due to regular world-changing actions and sensing actions. We will discuss how to specify transitions due to sensing actions and how to characterize the difference between a world and the agent's knowledge about the world.

1.5 Control architectures and specification, synthesis and verification of control

Once we analyze an agent's ability (what actions it can do, when and what their impact would be), and are able to give a directive to an agent, the next issue is how the agent executes: what actions it executes and when. This is referred to as the control of an agent and the control program of an agent has several differences with standard computer programs.

Foremost among them is that when a basic action of an agent is performed it may change the truth value of several fluents in the world simultaneously, while similar basic steps in most² computer programs change the value of a single variable. This is because the variables in a program environment are not so tightly linked that changing one of them simultaneously causes change in many others. On the other hand in many of the worlds an agent resides in the fluents are tightly linked; lifting the cup lifts the water in it; moving from A to B simultaneously makes $at(A)$ false and $at(B)$ true. Some of the other differences are as follows:

- The basic actions of an agent may have non-deterministic effects, while the basic statements of a computer program usually change the program in a deterministic way.
- In most programming environments the program in focus is the only one which can change the values of the variables. In the environment of an agent other agents can change the world. Thus an agent in a dynamic environment can not just assume that the value of a fluent remains unchanged, unless the agent itself changes it.
- Automatic synthesis of agent control is an important issue with autonomous agents, while programs are normally written by people. Because of this, an agent control language depends on the assumptions about the environment, and usually the simplest one that is adequate is picked. For example, in a static domain where our agent is the only one that can make changes, and where the actions are deterministic, the agent control can be simply described by a sequence of actions. But even then automatic synthesis is difficult; its NP-complete even under simplifying assumptions.

²Java has object constructs that are tightly linked.

Thus the agent control language may include features whereby the agent designer may specify some domain knowledge that will help the agent synthesize its control.

Based on the above differences we need to look at agent control very differently from standard programming. To start with we need to explore control architectures ranging from reactive architecture consisting of “sense; react” cycles, deliberative architecture consisting of “observe; (re)plan; execute a bit” cycles, and various levels of hybrid architectures that allow reaction for a large number of cases, and judicious deliberation for the other cases. We then need to consider whether plans are simple action sequences, or may have sensing actions, conditionals, and more general features. From the synthesis point of view we need to explore plan synthesis algorithms for various kinds of goals, and various kinds of plan structures. Finally, we need to consider various ways to specify domain knowledge that can help in plan synthesis. This includes procedural knowledge as in Golog, partial ordering knowledge as in HTNs and temporal knowledge specified in a temporal logic.

1.6 The reasoning issues: prediction, planning, explanation, diagnosis, plan verification, and control design

In the previous sections we discussed several different languages: languages to specify actions and their effects; languages to specify observations; languages to specify directives; and languages to specify control. Let us now give names to specifications in each of these languages and tie them together.

- Domain description D : The specification D consists of specification of actions, including sensing actions and their effects, the executability conditions of the actions, and the relationship (possibly causal) between fluents. The characterization of D is a transition function that shows the transition between states due to actions, and in presence of sensing action and incompleteness, the transition between knowledge states due to regular and sensing actions.
- Observation specification O_a and O_b : Observation specifications are observations about action occurrences and value of fluents at particular time points, and ordering information between time points. Observations are grouped into two kinds: abducible observations O_a , the observations that can be explained or that are caused; and basic observations O_b , the observations that record happenings out of free will. The characterization of D , and O_b together is a set of mapped trajectories, where each mapped trajectory consists of a trajectory and a mapping of time points, including the current time point, to states in the trajectory. We will refer to the mapped trajectories of D and O_b together as a *model* of (D, O_b) . The observations in O_a are then used to select those mapped trajectories that explain O_a and filter out the remaining.
- Goal specification G : Goals are high level specification of what one desires in terms of how the world should evolve. We plan to specify it using an appropriate temporal logic. Its characterization will consist of structures made up of states, trajectories, and transition graphs, depending on the particular temporal logic used.
- Control specification C : The control specification is a control program in a particular language. In the simplest case, it will be a sequence of actions, and in more general cases it may include features such as sensing actions, conditional statements, and non-determinism operators together with control knowledge.

- A query Q is of the form G **by_means_of** C **at_timepoint** t , where C is a control specification, G is a goal specification and t is a time point. If C is a simple sequence of actions a_1, \dots, a_n and G is a fluent f , then the query f **by_means_of** a_1, \dots, a_n **at_timepoint** t refers to enquiring if f is true in the situation obtained by executing the sequence a_1, \dots, a_n at the situation corresponding to time point t . On the other hand if G is the temporal goal $\Box f$ then the query $\Box f$ **by_means_of** a_1, \dots, a_n **at_timepoint** t refers to enquiring if f is true in all situations reached during the execution of the sequence a_1, \dots, a_n at the situation corresponding to time point t . Thus “**by_means_of** C ” may refer to after C is executed or during the execution of C depending on the goal G .
- The entailments $M \models Q_1, \dots, Q_n$ and $(D, O_b) \models Q_1, \dots, Q_n$: $M \models Q_1, \dots, Q_n$ is true if $Q_1 \dots Q_n$ evaluate to true with respect to the model M . The entailment $(D, O_b) \models Q_1, \dots, Q_n$ is true if each of the Q_i s evaluate to true with respect to all the models of (D, O_b) . In the entailments above queries can be replaced by observations.

We now illustrate how the models of (D, O_b) and the entailment $(D, O_b) \models Q_1, \dots, Q_n$ is central to the notion of verifying and synthesizing appropriate controls for given directives.

- Control verification: Given D and O_b , the current time point t_n , the goal G , if we would like to verify if G is indeed obtained by means of a control C then we need to check if $(D, O_b) \models G$ **by_means_of** C **at_timepoint** t_n is true or not.
- Control generation or planning: Given a D , and O_b , the current time point t_n , the goal G , planning for the goal G starting from t_n means finding a control C such that $(D, O_b) \models G$ **by_means_of** C **at_timepoint** t_n .

Chapter 2

Representing actions and the environment

If one were to start the design of an autonomous agent from scratch then a first step in the systematic design of an autonomous agent would be to analyze the kind of directives that will be given to the agent, the kind of world the agent would be in and from that design the actuators and sensors for the agent. However, often designers are given an unprogrammed artifact with built-in abilities with respect to acting and sensing and are required to develop the software that would make the artifact an autonomous agent with respect to certain kinds of directives and certain types of worlds. This means the designers would have to develop software that can synthesize controls for given directives or at the minimum can verify a given control and adapt it to new directives. Thus our first research issue becomes the representation of the world and the actions that the agent can do and the impact of these actions on the world.

Representing actions and their effects on the world has a long history in AI. The realization that the number of states of the world is often exponential with respect to the number of fluents – properties of the world, immediately ruled out a representation that stores the explicit mapping from states and actions to states. Some of the early historical landmarks on action representation include the use of STRIPS operators for planning [FN71], the invention of situation calculus [MH69], the discovery of the frame problem, the early attempt to solve the frame problem using non-monotonic logics [MH69], use of formalization of the frame problem as a benchmark for non-monotonic logics and the Yale shooting misunderstanding [McD87]. Some of the lessons from these were as follows:

1. The original STRIPS syntax is very restrictive. With slight generalization (say actions with non-deterministic effects) the semantics is no longer straightforward.
2. Representing what changes in the world due to an action is comparatively easy and succinct in a logical language. However representing what does not change, is either hard or verbose. Finding a succinct representation of what does not change is often referred to as the frame problem.
3. While non-monotonic logics can be used to for a succinct solution of the frame problem, the semantic nuances of non-monotonic logics are even difficult for specialists and need special care.
4. Representing English statements in non-monotonic logic leads to misunderstandings as the semantics of English is not precise.

Based on the above lessons the early nineties saw a large body of research on high level action description languages whose syntax were English-like and whose semantics were defined using set theory and simple

mathematical notions rather than mathematical logics. The reasoning with respect to these languages could then be done through a translation to a logical language and one now had a way to formally prove that the logical encodings were correct. The use of set theory and simple mathematical notions in defining the semantics of these languages freed the reader from knowing or learning a logical language to understand the semantics. In the rest of this section we describe several of these high level languages. Our presentation will track the evolution of these languages and will focus on the new aspects of each succeeding language. This will make the nuances of each aspect clearer as opposed to if we were to present a single language with all the features. We start with the simplest language called \mathcal{A} .

2.1 The language \mathcal{A} : motivation

The language \mathcal{A} was proposed in [GL93] as a simple action description language with an English-like syntax and a semantics defined using simple set theory. Later in [Kar93] it was used to show the correctness of various logical solution of the frame problem. To illustrate the English-like syntax of \mathcal{A} let us consider the description of the actions in one version of the Yale shooting problem. The description of the actions and their effects consists of the following sentences:

Loading the gun causes the gun to be loaded. Shooting causes the turkey to die. Shooting can be executed only if the gun is loaded.

Using the syntax of \mathcal{A} the above information is expressed as follows:

$$\begin{array}{lll} \textit{load} & \mathbf{causes} & \textit{loaded} \\ \textit{shoot} & \mathbf{causes} & \neg\textit{alive} \\ \mathbf{executable} & \textit{shoot} & \mathbf{if} \textit{loaded} \end{array}$$

The general syntax of the action description part of \mathcal{A} is given through two kinds of constructs, effect propositions and executability conditions. They are of the following form:

- effect proposition:

$$a \mathbf{causes} l \mathbf{if} \varphi \tag{2.1}$$

- executability condition:

$$\mathbf{executable} a \mathbf{if} \varphi \tag{2.2}$$

where a is an action, l is a fluent literal, and φ is a set of fluent literals or a special symbol \top , which is not a fluent and represents *true*. Intuitively, (2.1) says that if a is executed in a situation, in which φ holds, then l will be true in the resulting situation. (2.2) specifies the condition under which a can be executed. When $\varphi = \top$, we will drop the **if**-part in (2.1) and simply write $a \mathbf{causes} l$.

A domain description D in \mathcal{A} consists of a set of effect proposition and executability conditions. We will assume that for each action a , the executability condition $\mathbf{executable} a \mathbf{if} \top$ belongs to D if it does not contain any executability condition whose action is a .

Recall that an autonomous agent would often take a domain description, observations and a given goal to either validate a given control or synthesize the control. In \mathcal{A} the observation language can be used to express observations of the kind: “*The turkey is initially alive.*”. Using the syntax of \mathcal{A} this observation is expressed as follows:

initially *alive.*

In general, observations (O) in \mathcal{A} are a collection of propositions of the form:

initially l (2.3)

where l is a fluent literal.

We often refer to the pair (D, O) where D is a domain description and O is a set of observations as an *action description*.

Queries in \mathcal{A} are used to express questions such as: “Would the turkey be alive after a shoot action was attempted in the initial situation?” This is expressed in \mathcal{A} as follows:

alive **after** [*shoot*]

In general a query Q in \mathcal{A} is of the form:

l **after** [a_1, \dots, a_n] (2.4)

where l is a fluent literal and a_i 's are actions. Intuitively the above query means: *Would l be true if the sequence of actions a_1, \dots, a_n were attempted in the initial situation?*

For convenience, we introduce the following shorthands:

- For a sequence of literals l_1, \dots, l_n ,

initially l_1, \dots, l_n

denotes the collection of observations $\{ \mathbf{initially} \ l_i \mid 1 \leq i \leq n \}$.

- For a sequence of sets of literals $\varphi_1, \dots, \varphi_n$,

a **causes** l **if** $\varphi_1 \vee \dots \vee \varphi_n$

denotes the collection of effect propositions $\{ a \ \mathbf{causes} \ l \ \mathbf{if} \ \varphi_i \mid 1 \leq i \leq n \}$.

2.2 Reasoning with respect to a description in \mathcal{A} : Semantics of \mathcal{A}

The semantics of \mathcal{A} defines the entailment relation $(D, O) \models Q$. In other words, given a domain description D and observation O , can we conclude the query Q from D and O . Even with the simple syntax of queries, the entailment $(D, O) \models Q$ can be used for various kinds of reasoning that an autonomous agent may do. The following example illustrates this.

Example 1 (Formulating Plan Verification) Consider the domain description¹

$$D_1 = \left\{ \begin{array}{l} \textit{load} \textbf{causes} \textit{loaded} \\ \textit{shoot} \textbf{causes} \neg\textit{alive} \textbf{if} \textit{loaded} \end{array} \right\}$$

and the observation

$$O_1 = \{ \textbf{initially} \neg\textit{loaded}, \textit{alive} \}.$$

If the agent's goal is to make $\neg\textit{alive}$ true and it is given a possible plan of $[\textit{load}, \textit{shoot}]$ for achieving this goal, then the agent can verify the plan by checking whether

$$(D_1, O_1) \models \neg\textit{alive} \textbf{after} [\textit{load}, \textit{shoot}]$$

holds. In this case, intuitively the above entailment does hold. □

Example 2 (Formulating Simple Planning) Consider D_1 and O_1 from the previous example. Suppose the agent's goal is still to make $\neg\textit{alive}$ true. However, instead of verifying a given plan, it has to find a plan for achieving his goal. In other words, the agent needs to find a sequence of actions α such that

$$(D_1, O_1) \models \neg\textit{alive} \textbf{after} \alpha$$

holds. In this case, intuitively the above entailment holds when α is \textit{load} followed by \textit{shoot} , i.e., $\alpha = [\textit{load}, \textit{shoot}]$. □

Example 3 (Formulating Conformant Planning) Consider the domain description D_1 from Example 1 and let $O_2 = \{ \textbf{initially} \textit{alive} \}$. Suppose the agent's goal is still to make $\neg\textit{alive}$ true. As in Example 2, the agent needs to find a sequence of actions α such that

$$(D_1, O_2) \models \neg\textit{alive} \textbf{after} \alpha$$

holds. Since O_2 does not completely specifies the initial situations, the plan must be able to achieve the goal no matter which of the possible initial situations is the initial situation. The planning in this situation is referred to as conformant planning. □

Example 4 (Formulating Abduction) Consider D_1 and O_2 from Example 1 and 3 respectively; and suppose the agent wants to find out under what additional conditions (about the initial situation) he can guarantee that $\neg\textit{alive} \textbf{after} [\textit{load}, \textit{shoot}]$. This can be posed as finding O' such that

$$(D_1, O_2 \cup O') \models \neg\textit{alive} \textbf{after} \alpha$$

holds. □

We now give the semantics of the language \mathcal{A} .

¹We omit **executable load if** \top and **executable shoot if** \top from D_1 .

2.3 Semantics of \mathcal{A}

To characterize the semantics of \mathcal{A} let us ponder about the information contained in D and O : O contains information about the initial state and D tells us what will be true in a state after an action is executed. In other words, D tells us about the transition between states due to actions. Moreover, to answer queries of the form

$$f \text{ after } [a_1, \dots, a_n]$$

it is sufficient to know what the initial state is and the state transition due to actions. Thus the semantics of \mathcal{A} involves finding the initial state σ_0 and the transition function Φ (between states due to actions) with respect to a given D and O , and using them to define the entailment between (D, O) and queries.

For a particular domain description D , we have an associated signature consisting of a set of actions Act and a set of fluents Fl . *Fluent literals* are fluents or their negations. A *state* is a subset of Fl . A fluent f holds in a state σ if $f \in \sigma$ and a fluent literal $\neg f$ holds in a state σ if $f \notin \sigma$. Given a fluent literal l , we use \bar{l} to denote its negation. In other words, if $l = f$ for some fluent f then $\bar{l} = \neg f$ and if $l = \neg f$ for some fluent f then $\bar{l} = f$. An action a is *executable* in a state σ if D contains an executable condition **executable a if φ** and φ holds in σ .

Definition 1 A set of fluents σ is an *initial state* corresponding to O , if for all observations of the form **initially l** (where l is a fluent literal), l holds in σ .

The transition function of a domain description is defined next.

Definition 2 Given a domain description D in \mathcal{A} its *transition function* is any function Φ from states and actions to states that satisfies the following conditions:

For all actions a , fluents f , and states σ :

- if a is executable in σ then
 - If a **causes f if φ** belongs to D and φ holds in σ then f must be in $\Phi(a, \sigma)$.
 - If a **causes $\neg f$ if φ** belongs to D and φ holds in σ then f must not be in $\Phi(a, \sigma)$.
 - If D does not include such effect propositions (about a and f) then $f \in \Phi(a, \sigma)$ iff $f \in \sigma$.
- if a is executable in σ then $\Phi(a, \sigma)$ is undefined. □

Proposition 1 For any domain description D , there is at most one transition function.

Proof. Assume that D has two distinct transition function Φ and Ψ . This means that there exists some state s and action a such that $\Phi(a, \sigma) \neq \Psi(a, \sigma)$. Clearly, neither $\Phi(a, \sigma)$ nor $\Psi(a, \sigma)$ can be undefined. This means that there exists some $f \in Fl$ such that $f \in \Phi(a, \sigma)$ and $f \notin \Psi(a, \sigma)$. $f \in \Phi(a, \sigma)$ implies that (i) there exists some a **causes f if φ** in D such that φ holds in σ or (ii) $f \in \sigma$. (i) would indicate that $f \in \Psi(a, \sigma)$ and hence cannot happen. (ii), together with $f \notin \Psi(a, \sigma)$, implies that there exists some a **causes $\neg f$ if ψ** in D such that ψ holds in σ . But this means that f must not be in $\Phi(a, \sigma)$. A contradiction with $f \in \Phi(a, \sigma)$, i.e., a contradiction with our assumption that D has two distinct transition functions. □

Definition 3 A domain description D is said to be *consistent* if it has a transition function.

Remark 1 Following Proposition 1, a consistent domain description has a unique transition function. For convenience, the transition function of a consistent domain description D will be denoted by Φ_D .

The next example shows that not every domain description is consistent.

Example 5 (Inconsistent Domain Description) it is easy to see that the domain description with two effect propositions

$$\begin{aligned} a \text{ causes } f \text{ if } h \\ a \text{ causes } \neg f \text{ if } g \end{aligned}$$

is inconsistent since it does not have any transition function Φ since for the state $\sigma = \{h, g\}$ and action a , $\Phi(a, \sigma)$ is not defined. \square

Remark 2 Inconsistent domain descriptions indicate that there is something wrong and/or something missing in the modeling process. (Need more elaboration ???)

Proposition 2 For any consistent domain description D , its transition function Φ_D satisfies the condition:

- if a is executable in σ then

$$\Phi_D(a, \sigma) = \sigma \cup E_D^+(a, \sigma) \setminus E_D^-(a, \sigma)$$

where

$$E_D^+(a, \sigma) = \{f \mid a \text{ causes } f \text{ if } \varphi \in D \text{ and } \varphi \text{ holds in } \sigma\}$$

and

$$E_D^-(a, \sigma) = \{f \mid a \text{ causes } \neg f \text{ if } \varphi \in D \text{ and } \varphi \text{ holds in } \sigma\}.$$

- if a is not executable in σ then $\Phi_D(a, \sigma)$ is undefined.

Proof. Follows directly from Definition 2. \square

Example 6 Consider the action description (D_1, O_1) from Example 1. Let

$$\sigma_0 = \{alive\}.$$

We have that

$$\Phi_{D_1}(shoot, \sigma_0) = \{alive\}$$

because $E_{D_1}^+(shoot, \sigma_0) = E_{D_1}^-(shoot, \sigma_0) = \emptyset$ and

$$\Phi_{D_1}(load, \sigma_0) = \{alive, loaded\}$$

since $E_{D_1}^+(load, \sigma_0) = \{loaded\}$ and $E_{D_1}^-(load, \sigma_0) = \emptyset$ \square

It is easy to see that if for every fluent f , $\{\text{initially } f, \text{initially } \neg f\} \not\subseteq O$, then there exists at least one initial state corresponding to O . We will say that O is *consistent* if there exists at least one initial state corresponding to O .

Definition 4 For a transition function Φ , a state σ , and a sequence of actions a_1, \dots, a_n , we define the extended transition function of Φ , denoted by $\widehat{\Phi}$, as follows.

$$\widehat{\Phi}([a_1, \dots, a_n], \sigma) = \begin{cases} \sigma & \text{if } n = 0 \\ \Phi(a_n, \widehat{\Phi}([a_1, \dots, a_{n-1}], \sigma)) & \text{otherwise} \end{cases}$$

Definition 5 (σ_0, Φ) is a model of (D, O) iff Φ is the transition function of D and σ_0 is an initial state corresponding to O .

Definition 6 Let (D, O) be an action description.

- (D, O) is said to be *consistent* if it has a model.
- (D, O) is said to be *complete* if it has a unique model.
- For an action sequence α , $(D, O) \models f$ **after** α if for all models (σ_0, Φ) of (D, O) , f holds in $\widehat{\Phi}_D(\alpha, \sigma_0)$.
- An action sequence α is a *plan* for a fluent formula φ if $(D, O) \models \varphi$ **after** α .

Observe that consistency of an action description requires the consistency of its domain description and observations. Because of the uniqueness of the transition function, completeness is equivalent to the uniqueness of the initial state. This means that for each $f \in Fl$, $\{\mathbf{initially} \ f, \mathbf{initially} \ \neg f\} \cap O \neq \emptyset$.

Example 7 Continuing with Example 6, we have that the description (D_1, O_1) is complete and has a unique initial state $\sigma_0 = \{alive\}$. Furthermore, $\Phi_{D_1}(load, \sigma_0) = \{loaded, alive\}$.

Let $\sigma_1 = \{loaded, alive\}$. We have that

$$E_{D_1}^+(shoot, \sigma_1) = \emptyset \quad \text{and} \quad E_{D_1}^-(shoot, \sigma_1) = \{alive\}.$$

Hence, $\widehat{\Phi}([load, shoot], \sigma_0) = \Phi(shoot, \sigma_1) = \{loaded\}$. We have that $\neg alive$ holds in $\{loaded\}$, i.e.,

$$(D_1, O_1) \models \neg alive \text{ after } [load, shoot].$$

□

Exercise 1 Let $D'_1 = D_1 \cup \{shoot \text{ causes } \neg loaded\}$. Prove that

$$(D'_1, O_1) \models \neg alive \text{ after } [load, shoot].$$

2.4 Complexity of reasoning and planning in \mathcal{A}

\mathcal{A} is among the simplest languages for the specification of actions and their effects. In this section we analyze the complexity of reasoning and planning in this simple language. To this end, we say that the size of a domain description D in the language \mathcal{A}_0 is the sum of the number of actions, the number of fluents, and the number of propositions of the form (2.1) and denote it by $|D|$. We consider the following decision problems:

- C1 (*Consistent domain*) Given a domain description D , determine whether D is consistent or not.
- CI (*Consistent initial state*) Given a consistent domain description D , a set of observations O , and a set of fluents σ_0 , determine whether σ_0 is an initial state with respect to (D, O) .
- PC (*Planning with complete initial state*) Given a consistent domain description D , a complete set of observations about the initial state O , and a fluent literal l , a polynomial function $f : N \rightarrow N$, determining an action sequence a_1, \dots, a_m , where m is bounded by $f(|D|)$, such that $(D, O) \models l$ **after** $[a_1, \dots, a_m]$.

Theorem 1 1. CI is in P.

2. CI is in P.

3. PC is NP-complete.

Proof.

1. Φ_D exists iff for every pair of effect propositions of the form

$$a \text{ causes } f \text{ if } \varphi \quad \text{and} \quad a \text{ causes } \neg f \text{ if } \psi,$$

there exists no state σ such that both φ and ψ hold in σ . Due to the consistency of a state, this condition is satisfied iff there exists a fluent literal l such that $l \in \varphi$ and $\bar{l} \in \psi$, i.e., $\varphi \cap \bar{\psi} \neq \emptyset$, where $\bar{\delta} = \{\bar{l} \mid l \in \delta\}$. Clearly, this can be checked in polynomial time (in the size of the number of fluents). The conclusion follows from the fact that there are less than $|D|^2$ pairs of effect propositions.

2. Since O consists of observations of the form **initially** l , it has at most $2 * |Fl|$ propositions where $|Fl|$ is the number of fluents in Fl . Deciding whether or not σ_0 satisfies **initially** l is equivalent to checking whether $l \in \sigma_0$ (if $l \in Fl$) or $l \notin \sigma_0$ (if $l = \neg f$ for some $f \in Fl$). This can be done in at most $|Fl|$ operations. Thus, the problem is polynomial in the size of the number of fluents of D .
3. Given (D, O) , l , and f , let $\sigma_0 = \{f \mid f \text{ is a fluent and } \mathbf{initially} \ f \text{ belongs to } O\}$. Since O is complete, σ_0 is unique. Furthermore, it is easy to see that given a state σ and an action a , computing $\Phi_D(a, \sigma)$ is polynomial. As such, computing $\hat{\Phi}_D([a_1, \dots, a_m], \sigma_0)$ is also polynomial if m is bounded by $f(|D|)$.

(i) **Membership:** It is easy to see that the following guess-and-check NP-algorithm can be used to solve the PC problem.

- *Guess:* a sequence of actions a_1, \dots, a_m where m is bounded by $f(|D|)$.
- *Check:* whether l holds in $\hat{\Phi}_D([a_1, \dots, a_m], \sigma_0)$.

This indicates that PC belongs to the class of NP problems.

(ii) **Hardness:** We now reduce 3SAT to this problem. Let F be the formula

$$(f_1^1 \vee f_1^2 \vee f_1^3) \wedge \dots \wedge (f_n^1 \vee f_n^2 \vee f_n^3).$$

Let g_1, \dots, g_n, g be new propositions not in the language of F . We construct (D, O) as follows.

- The language of D consists of actions a , a' , and a_p for each proposition p in the language of F . D consists of the following effect propositions:

- For each i ,

$$\begin{aligned} a \text{ causes } g_i \text{ if } f_i^1 \\ a \text{ causes } g_i \text{ if } f_i^2 \\ a \text{ causes } g_i \text{ if } f_i^3 \end{aligned}$$

belong to D .

- D contains the effect proposition

$$a' \text{ causes } g \text{ if } g_1, \dots, g_n$$

- For each proposition p in the language of F , D contains the effect propositions

$$a_p \text{ causes } p \text{ if } \neg p$$

and

$$a_p \text{ causes } \neg p \text{ if } p.$$

- O consists of the following observations:
 - **initially** $\neg p$ for every proposition p in the language of F , and
 - **initially** $\neg g_1, \dots, \neg g_n, \neg g$.
- The goal of the planning problem is g .

Let α be an action sequence such that $(D, O) \models g \text{ after } \alpha$. We can observe the following:

- α must contain the actions a and a' and a must occur before a' . (Because a' is the only action for achieving g . Furthermore, for g to be true after the execution of a' , g_1, \dots, g_n need to be true and this can only be achieved by a .)
- $(D, O) \models g \text{ after } \beta$ where β is obtained from α by (i) removing all the action occurrences after the occurrence of a' and (ii) removing all the action occurrences between a and a' .

The above observations, together with the fact that a_p flips the truth value of p , allow us to conclude that there exists a sequence of actions $[a_1, \dots, a_m]$ such that

$$(D, O) \models g \text{ after } [a_1, \dots, a_m]$$

iff there exists a sequence of actions b_1, \dots, b_k, a, a' where $k \leq n_p$ (n_p is the number of propositions in the language of F) and $b_i \neq b_j$ for every pair of i and j , $1 \leq i \neq j \leq n$ such that

$$(D, O) \models g \text{ after } [b_1, \dots, b_k, a, a'].$$

Furthermore, let I be a truth value assignment for propositions in the language of F . Assume that p_1, \dots, p_k are true and other propositions are false w.r.t. I . It is easy to check that $(D, O) \models g \text{ after } [a_{p_1}, \dots, a_{p_k}, a, a']$ iff I is a model of F . This one-to-one correspondence between interpretations of F and plans of length less than or equal to $n_p + 2$ for g shows that F is satisfiable iff there exists a plan for g with respect to (D, O) . \square

2.5 Implementing reasoning and planning in \mathcal{A} by translating to SAT

In this section we show how we can encode domain descriptions of \mathcal{A}_0 in SAT and use model finding and reasoning in SAT to reason and plan with respect to \mathcal{A}_0 . We will address the following problems:

- **Hypothetical reasoning:** Given a domain description D , a complete set of observations O , a fluent literal l , and an action sequence $[a_1, \dots, a_t]$, checking whether or not $(D, O) \models l$ **after** $[a_1, \dots, a_t]$ holds.
- **Planning with complete information:** Given a domain description D , a complete set of observations O , a fluent literal l , an interger t , checking whether or not there exists an action sequence $[a_1, \dots, a_t]$ such that $(D, O) \models l$ **after** $[a_1, \dots, a_t]$.

2.5.1 Encoding for hypothetical reasoning

Given an integer n , we construct a theory $tr_do(D, O, n)$ for reasoning about effects of action sequences which contains at most n actions. The alphabet of $tr_do(D, O, n)$ consists of proposition of the form $f[i]$ and $a[i]$ where f is a fluent, a is an action, and i is an integer between 1 and $n + 1$. Intuitively, $tr_do(D, O, n)$ encodes the evolution of the world through the execution of the action sequence. Intuitively, $f[i]$ (resp. $\neg f[i]$) encodes that f is true (resp. false) in the state after the execution of the first $i - 1$ actions of the sequence. Similarly, $a[i]$ (resp. $\neg a[i]$) states that the action a is executed after the execution of the first $i - 1$ actions of the sequence. We introduce the following notations.

- For each fluent f , let
 - Act_f^+ be the collection of effect propositions of the form a **causes** f **if** φ in D ; and
 - Act_f^- be the collection of effect propositions of the form a **causes** $\neg f$ **if** φ in D .
- For a set of fluent literals φ and an integer i ,
 - $\varphi[i]$ denotes the formula $\bigwedge_{l \in \varphi} l[i]$; and
 - $\neg\varphi[i]$ denotes the formula $\bigvee_{l \in \varphi} \bar{l}[i]$. (Recall that \bar{l} denotes the negation of l .)

The theory $tr_do(D, O, n)$ is defined as follows.

- For each fluent f and an integer i , $1 \leq i \leq n$, the formula

$$f[i+1] \Leftrightarrow \left(\bigwedge_{a \text{ causes } f \text{ if } \varphi \in Act_f^+} (a[i] \wedge \varphi[i]) \right) \vee \left(f[i] \wedge \bigwedge_{a \text{ causes } \neg f \text{ if } \varphi \in Act_f^-} (\neg a[i] \vee \neg \varphi[i]) \right) \quad (2.5)$$

belongs to $tr_do(D, O, n)$.

- For each **initially** f in O , the proposition $f[1]$ is in $tr_do(D, O, n)$.
- If a_1, \dots, a_r are all the actions in the domain, then

- for all $1 \leq i \leq n$, if **executable** a **if** φ belongs to D and $\varphi \neq \top$, $a[i] \Rightarrow \varphi[i]$ belongs to $tr_do(D, O, n)$.
- for all $1 \leq i \leq n$, the formula $a_1[i] \vee \dots \vee a_r[i]$ is in $tr_do(D, O, n)$.
- for all $1 \leq i \leq n$ and $1 \leq j, k \leq r$ where $j \neq k$, the formula $\neg(a_j[i] \wedge a_k[i])$ is in $tr_do(D, O, n)$.

- A query Q given by f **after** a_1, \dots, a_t is translated to the formula

$$a_1[1] \wedge \dots \wedge a_t[t]$$

which we will refer to as $tr_q(Q)$.

The following proposition is about the correctness of the above translation.

Proposition 3 Given a consistent domain description D , a complete set of initial state observations O and a query Q of the form l **after** a_1, \dots, a_t , $(D, O) \models Q$ iff $tr_do(D, O, t) \cup tr_q(Q) \models l[t+1]$.

Proof. Let M be a model of $tr_do(D, O, t)$ and $s_i(M) = \{f \mid f \in Ft, f[i] \text{ is true in } M\}$. The encoding of $tr_do(D, O, t)$ ensures that for each integer i , $1 \leq i \leq t$, there exists a unique action a such that $a[i]$ is true in M and if $a[i]$ is true in M then a is executable in $s_i(M)$. We first prove that $s_1(M) = \sigma_0$ where σ_0 be the initial state of (D, O) . Since O is complete, for every fluent $f \in Ft$, we have that

- $f \in \sigma_0$ iff **initially** f belongs to O iff $f[1]$ belongs to $tr_do(D, O, t)$ iff $f \in s_1(M)$.
- $f \notin \sigma_0$ iff **initially** $\neg f$ belongs to O iff $\neg f[1]$ belongs to $tr_do(D, O, t)$ iff $f \notin s_1(M)$.

The above two items conclude that $s_1(M) = \sigma_0$.

We will now prove that $s_{i+1}(M) = \Phi(a_i, s_i(M))$ for $1 \leq i \leq t+1$ by induction over i where $a_i[i]$ is true in M .

- **Base:** The construction of $tr_do(D, O, t)$ implies that there exists some action a_1 such that $a_1[1]$ is true in M and a_1 is executable in $s_1(M)$. Thus $\Phi(a_1, s_1(M))$ is defined. Let f be a fluent. We have that $f \in \Phi(a_1, s_1(M))$ if one of the following two conditions is satisfied.
 - (i) $f \in E_D^+(a_1, s_1(M))$. This implies that there exists an effect proposition a_1 **causes** f **if** φ in Act_f^+ such that φ is true in $s_1(M)$. By Equation 2.5, we have that $f[2]$ is true in M , i.e., $f \in s_2(M)$.
 - (ii) $f \in s_1(M)$ and $f \notin E_D^-(a_1, s_1(M))$. This implies that $f[1]$ is true in M and for every effect proposition b **causes** $\neg f$ **if** φ in Act_f^- , either $b \neq a_1$, and hence, $\neg b[1]$ is true in M or $b = a_1$, and hence, φ does not hold in $s_1(M)$, i.e., $\neg\varphi[1]$ is true in M . Therefore, $f[1] \wedge \bigwedge_a \text{causes } \neg f \text{ if } \varphi \in Act_f^-(\neg a[1] \vee \neg\varphi[1])$ is true in M , and hence, $f[2]$ is true in M , i.e., $f \in s_2(M)$.
- **Step:** The proof for the inductive step is similar to the base case and is omitted here for brevity.

The above property allows us to show that if M is a model of $tr_do(D, O, t) \cup tr_q(Q)$ then $s_{t+1}(M) = \widehat{\Phi}([a_1, \dots, a_t], \sigma_0)$, which proves the ‘if’-direction of the proposition.

To prove the ‘only if’-direction of the proposition, let M be an interpretation defined by (i) $f[1]$ is true in M iff $f \in \sigma_0$; (ii) for each fluent f and integer $2 \leq i \leq t + 1$, $f[i]$ is true in M iff $f \in \widehat{\Phi}([a_1, \dots, a_{i-1}], \sigma_0)$; and (iii) for each action a , $a[i]$ is true in M iff $a = a_i$. It is easy to see that M is indeed a model of $tr_do(D, O, t) \cup tr_q(Q)$ and $s_{t+1}(M) = \widehat{\Phi}([a_1, \dots, a_t], \sigma_0)$, which concludes the ‘only if’-direction of the proposition. \square

2.5.2 Planning in complete action description

The following proposition follows immediately from Proposition 3.

Proposition 4 Let D be a consistent domain description, O be a complete set of observations about the initial state, and l be a fluent literal. A sequence of actions $[a_1, \dots, a_t]$ is a plan for l iff there is a model of $tr_do(D, O, t) \cup l[t + 1]$ in which $\{a_1[1], \dots, a_t[t]\}$ are true. \square

The above suggests that to find a plan, if we know that a plan of length t exists then we should give a propositional solver the propositional theory $tr_do_2(D, O, t) \cup g[t + 1]$ and ask it to find a model. Each model (if exists) will encode a plan.

If we do not know the plan length but have an idea of an upper bound, then we can have a dummy action which does not affect the world or we can replace $l[t + 1]$ by $l[1] \vee \dots \vee l[t + 1]$.

Example 8 Let us consider the example (D_1, O_1) . $tr_do(D_1, O_1, n)$ contains the following formulas:

- $\neg loaded[1] \wedge alive[1]$ (translating of O_1).
- $loaded[i + 1] \Leftrightarrow (load[i] \vee loaded[i])$ where $1 \leq i \leq n + 1$ (Equation 2.5, for fluent *loaded*).
- $alive[i + 1] \Leftrightarrow (alive[i] \wedge (\neg shoot[i] \vee \neg loaded[i]))$ where $1 \leq i \leq n + 1$ (Equation 2.5, for fluent *alive*).
- $load[i] \vee shoot[i]$ and $\neg(load[i] \vee shoot[i])$ where $1 \leq i \leq n$ (translating of actions).

To reason if $(D_1, O_1) \models alive$ **after** *load, shoot*, we translate the query $Q = \neg alive$ **after** *load, shoot* into the formula $load[1] \wedge shoot[2]$, i.e., $tr_q(Q) = load[1] \wedge shoot[2]$.

Now we can answer $(D_1, O_1) \models Q$ by checking whether or not $tr_do(D_1, O_1, 2) \cup tr_q(Q) \models \neg alive[3]$ holds. \square

2.6 Representing the relation between the fluents in the environment

So far in this section we mostly discussed the representation of actions and their effects on the environment. In this we did not explicitly consider the relationship between various fluents of the world. Any connection between them were addressed by making sure that the actions affecting one also affect the other in an appropriate way. For example, suppose there are two fluents *dead* and *alive* meaning the

turkey is dead and meaning the turkey is alive respectively. Now if we were to use the language \mathcal{A} or \mathcal{A} , for any effect axiom of an action that specifies the impact on one of the two fluents there would be a corresponding effect axiom about the other fluent. For example, whenever we say something about *alive* we must say the opposite about *dead*. Hence to match *shoot causes \neg alive if loaded* we will need to add *shoot causes dead if loaded*.

From the point of view of representing the effect of an action, often an action has some direct effects and because many fluents are linked with each other, it has some indirect effects. To express all the effects of an action directly one would then need to compute the indirect effects. But computation or reasoning is supposed to come later; after the representation. Thus a superior alternative is to express direct effects of actions as well as the relationship between the fluents and let the semantics or the reasoning mechanism compute the indirect effects. With respect to the *alive* and *dead* example, this approach will lead to adding the knowledge ‘dead iff \neg alive’. But that is beyond the syntax of \mathcal{A} . So we consider the language \mathcal{AR} which allows specification of relationship between fluents.

2.6.1 The language \mathcal{AR}

The language \mathcal{AR} was first introduced in [KL94] and was called \mathcal{AR}_0 . It was later improved and extended to deal with nonpropositional fluents in [GKL97]. The language also considers *non-inertial fluents* and indeterminate effect propositions. The following discussion is based on the materials in [GKL97] where we consider only propositional and inertial fluents. We also consider only deterministic effect propositions.

Syntax: The syntax of \mathcal{AR} is a superset of \mathcal{A} . In addition to the propositions allowed by \mathcal{A} , \mathcal{AR} allows classical domain constraints of the form:

$$\mathbf{always} \phi \tag{2.6}$$

where ϕ is a propositional formula. Intuitively, this means that in every state of the world, the formula ϕ is true.

As an example, the formula

$$\mathbf{always} \text{ dead} \Leftrightarrow \neg \text{alive}$$

states that *dead* and \neg *alive* are equivalent. This implies that there exists no state of the world containing $\{\text{dead}, \text{alive}\}$, in which both *dead* and *alive* are true and the formula $\text{dead} \Leftrightarrow \neg \text{alive}$ is false. Example 7 shows that $(D_1, O_1) \models \text{dead after } [\text{load}, \text{shoot}]$. As such, given the domain description $D_2 = D_1 \cup \{\mathbf{always} \text{ dead} \Leftrightarrow \neg \text{alive}\}$, we would like to conclude that $(D_2, O_1) \models \text{dead after } [\text{load}, \text{shoot}]$.

Semantics: Similar to \mathcal{A} , the semantics of \mathcal{AR} is based on defining an initial state and a transition function between states where each state is a set of fluents satisfying *all* domain constraints. In the case of \mathcal{A} the transition from one state to another due to an action was straightforward: the positive effects of the action were made true in the new state and the negative effects were made false in the new state and all other fluent values were left untouched. In case of \mathcal{AR} , we have direct effects that can be computed using E_D^+ and E_D^- , as well as indirect effects. For example, in the action description (D_2, O_1) , the action *shoot*, if executed after the action *load*, has \neg *alive* as its direct effect and *dead* as its indirect effect. Taking into account the indirect effects, that are caused by the classical domain constraints, poses a challenge.

This challenge is addressed by borrowing ideas from the belief update literature. The current state can be thought of as the initial belief and the effect of an action can be thought of as an update to the initial

belief. Thus the resulting state should be a state that satisfies the effect of the actions as dictated by the effect propositions as well as the domain constraints and is minimally different from the initial state.

To define the closeness between the states we use the following well known ordering based on set difference:

$$\sigma_1 \leq_\sigma \sigma_2 \text{ iff } (\sigma_1 \setminus \sigma) \cup (\sigma \setminus \sigma_1) \subseteq (\sigma_2 \setminus \sigma) \cup (\sigma \setminus \sigma_2).$$

As usually, we write $\sigma_1 <_\sigma \sigma_2$ to denote that $\sigma_1 \leq_\sigma \sigma_2$ and $\sigma_2 \not\leq_\sigma \sigma_1$. For a set of states S and a state σ , $\min_{<_\sigma}(S) = \{\sigma' \mid \sigma' \in S, \text{ there exists no } \sigma'' \in S \text{ such that } \sigma'' <_\sigma \sigma'\}$.

Example 9 Consider the action description D_2 . Let $\sigma = \{\text{alive}, \text{loaded}\}$, $\sigma_1 = \{\text{loaded}, \text{dead}\}$, and $\sigma_2 = \{\text{dead}\}$. We have that

$$(\sigma_1 \setminus \sigma) \cup (\sigma \setminus \sigma_1) = \{\text{alive}, \text{alive}\}$$

and

$$(\sigma_2 \setminus \sigma) \cup (\sigma \setminus \sigma_2) = \{\text{dead}, \text{alive}, \text{loaded}\}.$$

This implies that $\sigma_1 \leq_\sigma \sigma_2$. Clearly, we also have that $\sigma_1 <_\sigma \sigma_2$. □

Based on the above intuition we now list the conditions that σ' must satisfy to be a state that can be reached by executing an action a in a state σ .

- σ' must be a valid state, defined as states that satisfy all the domain constraints. (Note that $\sigma \cup E_D^+(a, \sigma) \setminus E_D^-(a, \sigma)$ may not be a valid state.)
- $E_D^+(a, \sigma) \subseteq \sigma'$ must be true.
- $E_D^-(a, \sigma) \cap \sigma' = \emptyset$ must be true.
- σ' must be as close to σ as possible.

As before, we will require that $\Phi(a, \sigma)$ is undefined whenever a is not executable in σ . Let

$$\text{States}(D) = \{\sigma \mid \sigma \subset Ft, \sigma \text{ satisfies all domain constraints in } D\}.$$

The set of possible states after the execution of a in σ is defined by

$$\Omega_D(a, \sigma) = \{\sigma' \mid \sigma' \in \text{States}(D), E_D^+(a, \sigma) \subseteq \sigma', E_D^-(a, \sigma) \cap \sigma' = \emptyset\}.$$

Finally, we define $\Phi(a, \sigma)$ as the elements in $\Omega_D(a, \sigma)$ which are as close to σ as possible:

$$\Phi(a, \sigma) = \begin{cases} \min_{<_\sigma}(\Omega_D(a, \sigma)) & \text{if } a \text{ is executable in } \sigma \\ \emptyset & \text{if } a \text{ is not executable in } \sigma \end{cases}$$

Example 10 Consider the domain description D_2 , the action *shoot*, and the state $\sigma = \{\text{alive}, \text{loaded}\}$. We have that

- $\text{States}(D_2)$ consists of $\{\text{alive}\}$, $\{\text{alive}, \text{loaded}\}$, $\{\text{dead}\}$, and $\{\text{dead}, \text{loaded}\}$.
- $E_{D_2}^+(\text{shoot}, \sigma) = \emptyset$ and $E_{D_2}^-(\text{shoot}, \sigma) = \{\text{alive}\}$.

- $(\sigma \cup E_{D_2}^+(shoot, \sigma)) \setminus E_{D_2}^-(shoot, \sigma) = \{loaded\}$. Observe that $\{loaded\} \notin States(D_2)$ because it violates the domain constraint **always** $dead \Leftrightarrow \neg alive$.
- Since $E_{D_2}^+(shoot, \sigma) \subseteq \sigma'$ and $\sigma' \cap E_{D_2}^-(shoot, \sigma) = \emptyset$ for every element σ' in $\Omega(shoot, \sigma)$ and the only two states in $States(D_2)$ satisfying these conditions are $\{dead\}$ and $\{dead, loaded\}$, we have that $\Omega(shoot, \sigma) = \{\{dead\}, \{dead, loaded\}\}$.
- Example 9 shows that $\{dead, loaded\} <_{\sigma} \{dead\}$. Thus, $\Phi(shoot, \sigma) = \min_{<_{\sigma}}(\Omega(shoot, \sigma)) = \{\{dead, loaded\}\}$. \square

Observe that transitions between states of \mathcal{AR} action descriptions could be non-deterministic.

Example 11 Consider the domain description D_3 ,

$$D_3 = \left\{ \begin{array}{l} \text{toss causes tossed} \\ \text{always tossed} \Leftrightarrow \text{heads} \oplus \text{tails} \end{array} \right\}$$

where \oplus is the exclusive-or operation. Here, $States(D_3) = \{\emptyset, \{tossed, heads\}, \{tossed, tails\}\}$ and $\{tossed\}$ is not a state.

If the action *toss* is executed in the state \emptyset , then, intuitively, the resulting states may be $\{tossed, heads\}$ or $\{tossed, tails\}$. This is verified by $\Phi_{D_3}(toss, \emptyset) = \{\{tossed, heads\}, \{tossed, tails\}\}$. \square

The semantics of \mathcal{AR} action descriptions will be defined by modifying Definitions 1, 4, 5, and 6 to take into account domain constraints. In particular, let (D, O) be an \mathcal{AR} action description. We say that

- σ_0 is an initial state if $\sigma_0 \in States(D)$ and σ_0 satisfies all domain constraints in D .
- $\widehat{\Phi}_D([], \sigma) = \{\sigma\}$, $\widehat{\Phi}_D([a_1, \dots, a_k], \sigma) = \bigcup_{\sigma' \in \widehat{\Phi}_D([a_1, \dots, a_{k-1}], \sigma)} \Phi_D(a_k, \sigma')$, where we say that if $\widehat{\Phi}_D([a_1, \dots, a_{k-1}], \sigma) = \emptyset$ then $\widehat{\Phi}_D([a_1, \dots, a_k], \sigma) = \emptyset$.
- (Φ_D, σ_0) is a model of (D, O) if Φ_D is a transition function and σ_0 is an initial state.
- $(D, O) \models \varphi$ **after** α iff for every model (Φ_D, σ_0) of (D, O) , $\widehat{\Phi}_D(\alpha, \sigma_0) \neq \emptyset$ and φ holds in every state belonging to $\widehat{\Phi}_D(\alpha, \sigma_0)$.

2.6.2 Why classical constraints are not enough?

Domain constraints allows us to compactly represent many interesting constraints between fluents. The following domain description, taken from [Lin95], shows that the use of classical logic formula to represent constraints might sometimes lead to counterintuitive results. In this domain, we have a suitcase with two lock and a spring loaded mechanism which will open the suitcase when both of the locks are in the up position. We can flip the lock to put it in the up position. This is detailed in the next example.

Example 12 (Lin's Suitcase) Consider the action description (D_4, O_4) where

$$D_4 = \left\{ \begin{array}{l} \text{always } up_1 \wedge up_2 \Rightarrow open \\ flip_1 \text{ causes } up_1 \\ flip_2 \text{ causes } up_2 \end{array} \right\}$$

and $O_4 = \{\mathbf{initially} \ up_1, \neg up_2, \neg open\}$.

There are eight possible states of the world

$$\begin{array}{ll} \sigma_1 = \emptyset & \sigma_5 = \{open\} \\ \sigma_2 = \{up_1\} & \sigma_6 = \{open, up_1\} \\ \sigma_3 = \{up_2\} & \sigma_7 = \{open, up_2\} \\ \sigma_4 = \{up_1, up_2\} & \sigma_8 = \{open, up_1, up_2\} \end{array}$$

of which σ_4 does not satisfy the constraint $up_1 \wedge up_2 \Rightarrow open$. Thus we have that

$$States(D_4) = \{\sigma_1, \sigma_2, \sigma_3, \sigma_5, \sigma_6, \sigma_7, \sigma_8\}$$

and σ_2 is the initial state of (D_4, O_4) . We are interested in computing $\Phi_{D_4}(flip_2, \sigma_2)$. We have that

- $E_{D_4}^+(flip_2, \sigma_2) = \{up_2\}$.
- $E_{D_4}^-(flip_2, \sigma_2) = \emptyset$.
- $(\sigma_1 \cup E_{D_4}^+(flip_2, \sigma_2)) \setminus E_{D_4}^-(flip_2, \sigma_2) = \{up_1, up_2\}$ and is not a state.
- $\Omega(flip_2, \sigma_2) = \{\sigma_3, \sigma_7, \sigma_8\}$.
- $(\sigma_3 \setminus \sigma_2) \cup (\sigma_2 \setminus \sigma_3) = \{up_1, up_2\}$.
- $(\sigma_7 \setminus \sigma_2) \cup (\sigma_2 \setminus \sigma_7) = \{open, up_1, up_2\}$.
- $(\sigma_8 \setminus \sigma_2) \cup (\sigma_2 \setminus \sigma_8) = \{open, up_1, up_2\}$.
- $\Phi_{D_4}(flip_2, \sigma_2) = \{\sigma_7, \sigma_8\}$.

Thus we have that executing $flip_2$ in the state $\{up_1\}$ results in either $\{up_2, open\}$ or $\{up_2, open, up_1\}$. In both states, up_2 (the direct effect of $flip_2$) is true and $open$ (the intended indirect effect of $flip_2$) is true as well. However, in one state up_1 becomes false. In other words, $flip_2$ also has another *unintended* indirect effect: it makes up_1 false in one of its possible successor states. \square

The main reason for the counterintuitive result in the above example lies in the fact that $up_1 \wedge up_2 \Rightarrow open$ is logically equivalent to $\neg open \wedge up_2 \Rightarrow \neg up_1$. The later constraint stipulates that if up_2 is true, either $open$ or up_1 should be false. This is certainly not our intention in the specification of the domain constraint **always** $up_1 \wedge up_2 \Rightarrow open$. This demonstrates that relations between the fluents in an environment does not always behave like a classical constraints. Several proposals have been developed to address this issue [Bar95, Lin95, LR94, MT95, Sha99, San96, McI00, Thi97]. Many of these proposals adopt the ‘one-directional’ specification of the relation between fluents. We will discuss one of these proposals in the next subsection.

2.6.3 Adding static causality to \mathcal{A} — The Language \mathcal{B}

The language \mathcal{B} is first presented in [GL98]. Like \mathcal{AR} , the syntax of the language \mathcal{B} is also a superset of \mathcal{A} . Instead of constraint of the form (2.6), \mathcal{B} introduces propositions, called *static causal propositions*, of the form

$$\varphi \text{ s.causes } l \tag{2.7}$$

where φ is a set of fluent literals and l is a fluent literal or the special symbol \perp which denotes *false*. Intuitively, (2.7) says that whenever φ holds so is l . For \perp means to represent *false*, a proposition of the form φ **s-causes** \perp implies that no state of the world satisfies φ .

It should be emphasized that the intended use of a static causal proposition of the form (2.7) differs from a domain constraint in that it is used only in one direction. The semantics of the language \mathcal{B} is formulated, as in the case of \mathcal{A} or \mathcal{AR} , by defining the notion of a state and transitions between states. As in the case of \mathcal{AR} , the execution of an action in a state can possibly result in several states. The key idea in defining the semantics of \mathcal{B} is that each fluent literal in the resulting state must have a reason for its existence. A fluent literal can be true if it is a *directly effect* of an action; an *indirectly effect* of an action; or *by inertia*. Before we formally define these notions, let us introduce some additional notions.

Let D be a domain description in \mathcal{B} . A set of fluent literals is said to be *consistent* if it does not contain f and $\neg f$ for some fluent f . An *interpretation* I of the fluents in D is a maximal consistent set of fluent literals of D . A fluent f is said to be true (resp. false) in I iff $f \in I$ (resp. $\neg f \in I$). The truth value of a fluent formula in I is defined recursively over the propositional connectives in the usual way. For example, $f \wedge g$ is true in I iff f is true in I and g is true in I . We say that a formula φ holds in I (or I satisfies φ), denoted by $I \models \varphi$, if φ is true in I .

Let u be a consistent set of fluent literals and K a set of static causal laws. We say that u is closed under K if for every static causal proposition

$$\varphi \text{ s-causes } l$$

in K , if $u \models \varphi$ then $u \models l$. By $Cl_K(u)$ we denote the least consistent set of literals from D that contains u and is also closed under K . We call $Cl_K(u)$ *the closure of u with respect to K* . It is worth noting that $Cl_K(u)$ might be undefined. For instance, if u contains both f and $\neg f$ for some fluent f , then $Cl_K(u)$ cannot contain u and be consistent; another example is that if $u = \{f, g\}$ and K contains

$$f \text{ s-causes } h \quad \text{and} \quad f, g \text{ s-causes } \neg h,$$

then $Cl_K(u)$ does not exist because it has to contain both h and $\neg h$, which means that it is inconsistent. Abusing the notation, for a domain description D and a set of fluent literals u , we write $Cl_D(u)$ to denote the closure of u with respect to the set of static causal propositions in D . Formally, the notion of a state in \mathcal{B} domain description is defined as follows.

Definition 7 Let D be a \mathcal{B} domain description. A *state* of D is an interpretation of the fluents in Ft that is closed under the set of static causal propositions of D .

An action a is *possibly executable* in a state s if there exists an executability proposition

$$\text{executable } a \text{ if } \varphi$$

in D such that $s \models \varphi$.

The *direct effect* of an action a in a state s is the set

$$e_D(a, s) = \{l \mid a \text{ causes } l \text{ if } \varphi \in D, s \models \varphi\}.$$

For a domain description D , $\Phi(a, s)$, the set of states that may be reached by executing a in s , is defined as follows.

$$\Phi(a, s) = \begin{cases} \{s' \mid s' \text{ is a state and } s' = Cl_D(e_D(a, s) \cup (s \cap s'))\} & \text{if } a \text{ is possibly executable in } s \\ \emptyset & \text{otherwise} \end{cases}$$

We will say that a is executable in s if $\Phi_D(a, s) \neq \emptyset$. This is somewhat different than the notion of executability of an action in case of \mathcal{A} . We will return to this issue in a short while. In the definition of $\Phi_D(a, s)$, we can see that a fluent literal l can be true in s' in three ways: (i) $l \in e_D(a, s)$ (direct effects); (ii) $s \cap s'$ (inertial); and $Cn_D(e_D(a, s) \cup (s \cap s')) \setminus (e_D(a, s) \cup (s \cap s'))$ (indirect effects.).

The difference between the semantics of \mathcal{AR} and \mathcal{B} action descriptions is highlighted in the next example. Let us revisit the action description representing the suitcase with the spring loaded mechanism in the notation of the language \mathcal{B} .

Example 13 (Lin's Suitcase Revisited) Consider the action description (D_5, O_5) where

$$D_5 = \left\{ \begin{array}{l} up_1, up_2 \text{ s.causes } open \\ flip_1 \text{ causes } up_1 \\ flip_2 \text{ causes } up_2 \end{array} \right\}$$

and $O_5 = \{\mathbf{initially } up_1, \neg up_2, \neg open\}$. D_5 differs from D_4 only in the use of static causal proposition to express the spring loaded mechanism.

Let $s = \{up_1, \neg up_2, \neg open\}$. Clearly, $flip_2$ is executable in s and $e_{D_5}(flip_2, s) = \{up_2\}$. Consider $u = \{open, up_2, \neg up_1\}$. We have that u is also a state of D_5 . Furthermore

$$Cl_{D_5}(e_{D_5}(flip_2, s) \cup (s \cap u)) = Cl_{D_5}(\{up_2\}) = \{up_2\} \neq u$$

which indicates that $u \notin \Phi_{D_5}(flip_2, s)$. On the other hand, for $v = \{open, up_2, up_1\}$,

$$Cl_{D_5}(e_{D_5}(flip_2, s) \cup (s \cap v)) = Cl_{D_5}(\{up_1, up_2\}) = \{up_1, up_2, open\} = v$$

which indicates that $v \in \Phi_{D_5}(flip_2, s)$. □

The next example shows that even though we do not consider indeterminate actions, the transitions caused by actions can still be non-deterministic.

Example 14 (Non-deterministic Transitions) Consider the domain description

$$D_6 = \left\{ \begin{array}{l} f, g \text{ s.causes } \neg h \\ f, h \text{ s.causes } \neg g \\ make_f \text{ causes } f \end{array} \right\}$$

Let $s = \{\neg f, g, h\}$. We compute $\Phi_{D_6}(make_f, s)$. We have that

- $e_{D_6}(make_f, s) = \{f\}$.
- for $u = \{f, g, \neg h\}$, $s \cap u = \{g\}$, and $Cn_{D_6}(e_{D_6}(make_f, s) \cup (s \cap u)) = Cn_{D_6}(\{f, g\}) = \{f, g, \neg h\} = u$.

- for $v = \{f, \neg g, h\}$, $s \cap v = \{h\}$, and $Cn_{D_6}(E_{D_6}(make_f, s) \cup (s \cap v)) = Cn_{D_6}(\{f, h\}) = \{f, \neg g, h\} = v$.

Thus $\Phi_{D_6}(make_f, s) = \{u, v\}$. □

As we have mentioned earlier the notion of action executability in \mathcal{B} is different than it was defined in \mathcal{A} . In the case of \mathcal{A} , the satisfaction of an executability condition of an action a in a state s is enough to guarantee that a is executable in s . As it turns out, static causal propositions can also indirectly cause actions to become inexecutable. This is illustrated in the following example.

Example 15 (Ramification and Qualification) Let us consider the domain description in which the action *kill* causes the turkey to be not *alive*, the action *make_walk* causes the turkey to be *walking*, and the relation stating that if the turkey is dead, it cannot walk. This can be encoded as follows.

$$D_7 = \left\{ \begin{array}{l} \textit{kill causes } \neg\textit{alive} \\ \textit{make_walk causes } \textit{walking} \\ \neg\textit{alive s.causes } \neg\textit{walking} \end{array} \right\}$$

This domain description has three states: $s_1 = \{\textit{walking}, \textit{alive}\}$, $s_2 = \{\neg\textit{alive}, \neg\textit{walking}\}$, and $s_3 = \{\textit{alive}, \neq \textit{walking}\}$. We have that

$$\Phi_{D_7}(\textit{kill}, s_1) = \{s_2\}$$

since $s_2 \cap s_1 = \emptyset$, $e_{D_7}(\textit{kill}, s_1) = \{\neg\textit{alive}\}$. $Cn_{D_7}(\{\neg\textit{alive}\}) = \{\neg\textit{alive}, \neg\textit{walking}\} = s_2$.

Let us now compute $\Phi_{D_7}(\textit{make_walk}, s_2)$. We have that $e_{D_7}(\textit{make_walk}, s_2) = \{\textit{walking}\}$. Thus any state s belonging to $\Phi_{D_7}(\textit{make_walk}, s_2)$ satisfies that $e_{D_7}(\textit{make_walk}, s_2) \subseteq s$. Since the only state satisfying this condition is s_1 , $s_2 \cap s_1 = \emptyset$, and $Cn_{D_7}(\{\textit{walking}\}) = \{\textit{walking}\} \neq s_1$, we can conclude that $\Phi_{D_7}(\textit{make_walk}, s_2) = \emptyset$. This means that *make_walk* can not be executed in s_2 . □

As we can see in the above example, the presence of the static causal proposition $\neg\textit{alive s.causes } \neg\textit{walking}$ prevents the action *make_walk* to be executable in the state s_2 even though the action is possibly executable in it. (Recall that by default, when no executability condition for *meka_walk* is specified, **executable make_walk if \top** belongs to the description.) Obviously, it is reasonable to consider that *make_walk* should not be executable whenever $\neg\textit{alive}$ is true. This leads us to the question of whether or not it would make sense for us to add to the domain description the executability condition **executable make_walk if *alive***? Clearly, the presence of this executability condition will prevent *make_walk* to be executable in s_2 . What will be the benefit of not adding this executability condition?

Adding **executable make_walk if *alive*** to the domain description might have been good in this situation. Yet, the price we have to pay is that the specification is not elaboration tolerant, i.e., the addition of new propositions to the description might require some modifications of the domain. For example, in addition to the information described in Example 15, we also know that the turkey is not able to walk if it is sick. Intuitively, this will have the effect of preventing *make_walk* to be executable in any state in which *sick* is true. Therefore, the executability condition **executable make_walk if *alive*** is no longer adequate to characterize the situations, when *make_walk* is executable, and needs to be replaced with **executable make_walk if *alive, \neg sick***. On the other hand, adding *sick s.causes } \neg\textit{walking}* to D_7 will achieve the same effect and does not require any changes in its previously encoded propositions.

Finally, we would like to emphasize that static causal propositions do not behave exactly as a logical implication (e.g. Example 13). In general, a static causal proposition

$$\varphi \text{ if } l$$

is not the same as the static causal proposition

$$\varphi, \bar{l} \text{ if } \perp$$

even though both disallow any interpretation satisfying $\varphi \cup \{\bar{l}\}$ to be considered as states. The former represents a ramification constraint while the latter represents a qualification constraint. The former encodes a reason for the literal l to become true, the latter does not.

Example 16 (Ramification vs. Qualification) Consider the domain descriptions

$$D_8 = \left\{ \begin{array}{l} f \text{ s_causes } \neg g \\ \text{make_}f \text{ causes } f \end{array} \right\} \qquad D_9 = \left\{ \begin{array}{l} f, g \text{ s_causes } \perp \\ \text{make_}f \text{ causes } f \end{array} \right\}$$

We have that

- $\Phi_{D_8}(\text{make_}f, \{\neg f, g\}) = \{\{f, \neg g\}\}$. In this domain, the static causal proposition indirectly changes the value of g . In other words, $\neg g$ has a reason to become true since f becomes true.
- $\Phi_{D_9}(\text{make_}f, \{\neg f, g\}) = \emptyset$. Here, the static causal proposition does not represent a reason for g to become false when f becomes true. As such, the value of g should not be changed, which causes the execution of $\text{make_}f$ in the state $\{\neg f, g\}$ to result in no state.

□