# 8

# Modeling Dynamic Domains

So far, we have limited our attention to *static* domains – no attempt was made to represent a domain's evolution in time. Recall from the introduction that we are interested in agents that are intended to populate *dynamic*, changing domains and should therefore be able to plan, explain unexpected observations, and do other types of reasoning requiring the ability to predict effects of series of complex actions. This can be done only if the agent has sufficient knowledge about actions and their effects. In this chapter we discuss one of several current approaches to representing and reasoning with such knowledge. We start by looking at an extended example that illustrates some of the issues that arise when we attempt to represent actions and their effects on the world. Once some of these issues become clear, we present a general, formal theory of actions and change, with further examples on how to apply it to various domains. The theory views the world as a dynamic system whose states are changed by actions, and provides an "action language" for describing such systems. This language allows concise and mathematically accurate descriptions of the system's states and of possible state-action-state transitions; it allows us to represent dynamic domains and their laws. Such representations can be translated into ASP programs that are used to give the precise semantics of the language. Later, we show how this and similar translations can be used to answer queries about the effects of actions in a given situation. In keeping with our plan to separate the representation of our world from using that representation to perform intelligent tasks, we save the discussion of planning and other types of reasoning for later chapters in which we apply the knowledge of answer set programming that we presented previously.

## 8.1 The Blocks World – A Historic Example

In 1966 a group of researchers at SRI International (then known as the Stanford Research Institute) decided to build a reasoning robot. The result

was Shakey (named for its jerky motions). The robot could receive a description of a goal from a human, make a plan, and execute the necessary actions to achieve the goal.

One task that the researchers believed a robot should be able to perform was to move blocks and create various configurations with them. This blocks world became what some have termed the "Drosophila of AI"[1] because this simple domain provided so much research potential. It has been especially popular in the planning community because of its simplicity and its search space that grows rapidly with the addition of blocks.

The work on Shakey that continued through 1972 brought to light many of the challenges involved in building and programming such a machine. Naturally, there were many hardware challenges. For example, making a robotic arm actually capable of picking something up turned out to be very difficult. Vision is also a very complex topic. A robot may have an onboard camera, but how does it parse out the necessary images? Shakey had to settle for pushing blocks around, and it could not evaluate whether it achieved the goal by seeing the result of its actions.

The software challenge turned out to be no less difficult. Many questions arose, such as the following:

- How do we represent knowledge?
- How do we teach a robot to use this knowledge to make plans?
- How do we teach a robot to evaluate if its execution of a plan was successful?
- How should the robot re-plan if there are changes?

Shakey used LISP and a theorem-proving planner called STRIPS. Its planning was domain-specific. Many planners that followed were also domain-specific, difficult to modify, and slow. Huge progress has been made since then, both in efficiency of the planners and in the kind of information they were able to represent. The approach we present separates representation of a domain from the reasoning done in that domain (i.e., teaching a computer about the laws of a world is separate from teaching it how to use this information for various tasks). This separation allows us to use the same method of representation for whatever domain we wish and

[1] The Drosophila fly, commonly known as the fruit fly, was chosen as a subject of study by geneticists because it was so easy to take care of and reproduced so quickly. It turned out that studying this little fly, so seemingly unlike humans, unlocked many secrets of our own genetic code. The blocks world, often criticized as a "toy" example so unlike the real world, nonetheless allowed scientists to learn much about commonsense reasoning in a changing world.

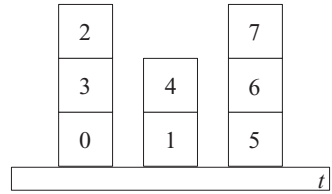



Figure 8.1. Initial Configuration

to use whatever reasoning algorithm we choose with our domain.[2] With the same information, an agent can plan its actions, attempt to explain unexpected events in the world, and so on. Separating the representation from the reasoning component allows for clearer and much more elaboration-tolerant programs.

Let's consider how a blocks-world problem can be encoded in ASP. First, let us define the particular version of the blocks world that we will use:

The **basic blocks world** consists of a robotic arm that can manipulate configurations of same-sized cubic blocks on a table. There are limitations to what the robotic arm can do. It can move *unoccupied* blocks, one at a time, onto other unoccupied blocks or onto the table. (An unoccupied block is one that does not have another block stacked on it.) At any given step, a block can be in at most one location; in other words, a block can be directly on top of one other block, or on the table. We do not impose a limit on how tall our towers can be. Our table is big enough to hold all the blocks, even if they are not stacked. We do not take into account spatial relationships of *towers*, just which *blocks* are on top of each other and which blocks are on the table.

Figures 8.1 and 8.2 illustrate an example problem in this domain. Blocks 0–7 are stacked on table $t$ as shown in Figure 8.1. Our robot could, for example, turn this configuration into the one shown in Figure 8.2 by putting block 2 on the table and block 7 on block 2.

We would like to be able to write a program to model the transformation of the domain caused by the robotic arm's activity; that is, given an

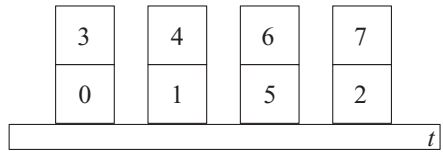



Figure 8.2. Final Configuration

[2] This does not mean that we cannot use domain-specific information to guide our reasoning. We show how we can do this with the logic-programming approach in Chapter 9.

initial position of blocks and a sequence of actions, our program should be able to answer queries about the positions of blocks after the execution of these actions. You should recall that writing a declarative program involves identifying the objects and the relations between the objects that you are interested in. The blocks and the table are some obvious objects. We would also like to talk about possible locations, so that we do not always have to distinguish between blocks and the table. We will denote blocks by $b0 \ldots b7$ and locations will be blocks plus the table (named $t$). To describe a configuration of blocks, we will use terms of the form $on(b, l)$ where $b$ is a block and $l$ is a location; the term states that block $b$ is on location $l$. Thus, a configuration $S$ is a set of terms $on(b, l) \in S$ if and only if $b$ is on $l$. For instance, the configuration from Figure 8.1 can be described by a collection of terms

$$\sigma_0 = \{on(b0, t), on(b3, b0), on(b2, b3), on(b1, t), on(b4, b1),$$
$$on(b5, t), on(b6, b5), on(b7, b6)\}.$$

The action of the robotic arm moving block $B$ to location $L$ will be denoted by terms of the form $put(B, L)$. Action $put(b2, t)$ changes our initial configuration into

$$\sigma_1 = \{on(b0, t), on(b3, b0), on(b1, t), on(b4, b1), on(b2, t),$$
$$on(b5, t), on(b6, b5), on(b7, b6)\}.$$

Action $put(b7, b2)$ transforms block configuration $\sigma_1$ into configuration

$$\sigma_2 = \{on(b0, t), on(b3, b0), on(b1, t), on(b4, b1), on(b2, t),$$
$$on(b5, t), on(b6, b5), on(b7, b2)\}.$$

This matches Figure 8.2. The execution of a sequence of these two actions in configuration $\sigma_0$ determines the system's trajectory[3]

$$\langle \sigma_0, put(b2, t), \sigma_1, put(b7, b2), \sigma_2 \rangle$$

that describes its behavior.

To describe the changes our system undergoes, we use integers from 0 to some finite $n$ to denote *steps* of the corresponding trajectories. (We limit the length of the trajectory for computational reasons.) We also distinguish between **fluents** – properties that can be changed by actions (such as one

[3] By a trajectory of a dynamic system we mean a sequence $\langle \sigma_0, a_0, \sigma_1, \ldots, \sigma_{n-1}, a_{n-1}, \sigma_n \rangle$ where $\langle \sigma_i, a_i, \sigma_{i+1} \rangle$ is a state-action-state transition of the system.

block being on top of another), and **statics** – properties that cannot (such as the state of something being a block). Blocks, locations, configurations, steps, actions, and fluents are the objects in our domain. Now it's time to consider the relationships between them. Let $\langle \sigma_0, a_0, \sigma_1, \ldots, a_n, \sigma_{n+1} \rangle$ be a trajectory of our system. Two new predicates, $holds(fluent, step)$ and $occurs(action, step)$, can be used to describe what fluents are true and what actions occurred at any given step. In our example, we define relation $holds(on(B, L), I)$, which says that block $B$ is on location $L$ at step $I$. When we want to say that block $B$ was put on location $L$ at step $I$, we simply say $occurs(put(B, L), I)$.[4]

We begin our program by defining the objects of the domain. Note that in our definition of steps, we use a new construct `#const` to define a constant `n`, which is then used in a range declaration `step(0..n)` to specify the maximum number of steps.

```
%% blocks:
block(b0).  block(b1).  block(b2).  block(b3).
block(b4).  block(b5).  block(b6).  block(b7).

%% A location can be a block or the table.
location(X) :- block(X).
location(t).

#const n = 2.
step(0..n).

%% "Block B is on location L" is a property that
%% changes with time.
fluent(on(B,L)) :- block(B), location(L).

%% "Put block B on location L" is a possible action
%% provided we don't try to put a block onto itself.
action(put(B,L)) :- block(B), location(L),
                    B != L.
```

[4] We could have made our time steps part of our fluent and action predicates and just said $on(B, L, I)$ and $put(B, L, I)$ instead of using $holds$ and $occurs$. However, reifying actions and fluents allows us to introduce rules involving these concepts themselves, rather than their specific instances. We see examples of such rules later in the chapter.

To illustrate the behavior of our program, we fix the initial configuration to the one shown in Figure 8.1. We can describe it by specifying the locations of the blocks at step 0 and including the closed world assumption for the $holds$ relation for the initial situation:

```
20 %% holds(on(B,L),I):block B is on location L at step I.
21 holds(on(b0,t),0).
22 holds(on(b3,b0),0).
23 holds(on(b2,b3),0).
24 holds(on(b1,t),0).
25 holds(on(b4,b1),0).
26 holds(on(b5,t),0).
27 holds(on(b6,b5),0).
28 holds(on(b7,b6),0).
29
30 %% If block B is not known to be on location L at
31 %% step 0, then we assume it is not.
32 -holds(on(B,L),0) :- block(B), location(L),
33                      not holds(on(B,L),0).
```

Note that the logic program on lines 1–33 completely defines the initial configuration of the blocks (i.e., we know the values of all the fluents of the domain at step 0 of our trajectory).

Now let's define the theory of the blocks world. To do this, we describe the effects of its actions. Since each action takes one step, the following rule describes an effect of action $put(B, L)$:

```
34 holds(on(B,L),I+1) :- occurs(put(B,L),I),
35                       I < n.
```

It states that putting block $B$ on location $L$ at step $I$ causes $B$ to be on $L$ at step $I + 1$. We assume for now that the robot never drops a block and is otherwise perfect in its execution of actions.

This rule can be viewed as a special case of a **causal law** – a statement of the form

$$a \textbf{ causes } f \textbf{ if } p_0, \ldots, p_m$$

that says that action $a$ executed in a state of the domain satisfying conditions $p_0, \ldots, p_m$ causes fluent $f$ to become true in the resulting state. Such general laws are discussed in the next section.

The new location of block $B$ can be viewed as a "direct" effect of action $put$. There are also "indirect" effects caused by relationships between fluents. For example, performing action $put(b2, t)$ in the initial situation has the direct effect of placing $b2$ on the table and the indirect effect of $b2$ being removed from $b3$. This type of indirect conclusions can often be obtained from direct ones by using relations between fluents. In this case, it is sufficient to know that a block occupies a single location. This can be expressed by the following rule:

```
36 -holds(on(B,L2),I) :- holds(on(B,L1),I),
37                       location(L2),
38                       L1 != L2.
```

The fact that no block can support more than one block directly on top can be expressed by the following rule[5]:

```
39 -holds(on(B2,B),I) :- block(B),
40                       holds(on(B1,B),I),
41                       block(B2),
42                       B1 != B2.
```

Later we show that both of these rules can be viewed as special cases of **state constraints** – statements of the form

$$f \textbf{ if } p_0, \ldots, p_m$$

that say that every state satisfying $p_0, \ldots, p_m$ must also satisfy $f$.

As usual, we test the success of our representation by inputting it into STUDENT and checking what it knows. Of course, we gave a complete initial situation, so it knows which blocks are where at step 0. Now let's expand our program by a new statement:

```
occurs(put(b2,t),0).
```

This statement allows us to ask the agent questions about what is true at step 1. Try giving STUDENT the program on lines 1–42 and the $occurs$ statement above and asking it query $holds(on(b2, t), 1)$. It should be able to figure out the answer on its own, based on the laws we encoded, and say $yes$. Now try query $holds(on(b0, t), 1)$. What do you expect? What does STUDENT answer?

[5] It is worth noting that statements $block(B)$ and $block(B2)$ in the body of the rule play substantially different roles. The latter is simply added for safety. The former, however, is necessary to correctly convey the meaning. If this statement were omitted, the rule would claim that not only a block but also a table cannot support more than one block.

A person would assume that $b0$ is still on the table because it was not moved, but STUDENT will answer "maybe." Is STUDENT not smart enough, or does it not have enough information? In most cases, unless they are told otherwise, humans live with an operative assumption that *things normally stay as they are*. Picking up an object does not normally affect properties of every other thing around us. We go on with our lives, feeling pretty safe that bunnies are still furry and the ocean is still big. It seems reasonable to assume that moving a single block does not cause other blocks to move so much that their locations change. At least for purposes of this example, we had assumed that our robot is good at manipulating blocks, so we would like STUDENT to believe that the only change that occurred is that $b2$ is now on the table and no longer on $b3$. We must teach it that, lacking evidence to the contrary, it should assume that *normally things stay as they are*. This principle is known as the **Inertia Axiom** and can be expressed by two rules:

```
43 holds(F,I+1) :- holds(F,I),
44                 not -holds(F,I+1),
45                 I < n.
46
47 -holds(F,I+1) :- -holds(F,I),
48                  not holds(F,I+1),
49                  I < n.
```

The rules state that without explicit evidence to the contrary, the value of fluent $F$ remains constant at step $I + 1$. This is a typical representation of defaults as in Chapter 5. Note that $not\ ab(d(F, I))$ is not included, which simplifies the program. Since our states are complete and the rule has no weak exceptions, the omission is justified. (Recall from Section 5.1.1 that a strong exception refutes the default's conclusion and a weak one renders the default inapplicable.)

The rules on lines 34–49 give a complete description of a configuration $\sigma_1$ that results from executing action $put(B, L)$. Create a program consisting of lines 1–49 and the $occurs$ statement and ask STUDENT some questions about its new configuration. Its answers should be more intuitive now. Later we show how to define "successor" states in more complex dynamic domains.

Is our description of the blocks world complete? To answer this question, change the $occurs$ statement of the program to contain an action that should not be executable:

```
occurs(put(b6,t),0).
```

Since $b6$ is occupied, this action should not be allowed. Ask STUDENT to describe the next state by asking it for all answers to query $holds(on(B, L), 1)$. You will see that the program derives that block $b6$ is on the table at step 1 and that $b7$ is still on top of it. This answer does not match our intuition unless we believe the robot arm to be very coordinated. Once again, we see that STUDENT needs more information; namely, we need to tell it which actions are not allowed. The next two rules are restrictions on the executability of actions.

```
50 -occurs(put(B,L),I) :- location(L),
51                        holds(on(B1,B),I).
52
53 -occurs(put(B1,B),I) :- block(B1),
54                         block(B),
55                         holds(on(B2,B),I).
```

The first rule says that it is impossible to move a block that is occupied. The second says that it is impossible to move a block onto an occupied block. These rules are examples of **executability conditions** whose general form is

**impossible** $a_1 \ldots a_k$ **if** $p_0, \ldots, p_m$.

Intuitively, the law states that it is impossible to execute actions $a_1 \ldots a_k$ simultaneously in a state satisfying conditions $p_0, \ldots, p_m$.

Add these rules and ask STUDENT to describe again what is true at Step 1. It should answer that there are no models, which indicates that the suggested scenario is inconsistent with the rules of the program, as we would expect.

The rules on lines 34–55 constitute a simple theory of the blocks world. Let's see how well STUDENT does if we add another step. To ask questions about the state of the world after block $b2$ is moved on the table and $b7$ is put on $b2$, we need to get rid of our bad $occurs$ statement and write the following two statements instead:

```
56 occurs(put(b2,t),0).
57 occurs(put(b7,b2),1).
```

Let us name lines 1–57 `blocks1.lp`. Querying it with $holds(on(B, L), 2)$ gives us everything that is true after the two actions have been executed. Finally our representation mirrors what we expect the program to know, and it can answer our queries intelligently. For practice, run STUDENT with `blocks1.lp`. Try changing the two $occurs$ statements in

the program to the nonexecutable statements $occurs(put(b2, b4), 0)$ and $occurs(put(b7, b4), 1)$. What is the result? Is the rule on lines 53–55 necessary for STUDENT to give the right answers?

Thus far our model of the blocks world contained only fluents formed by relation $on$. Let us now see if we can expand our model by introducing a new fluent, $above(B, L)$ – "block $B$ is located above location $L$." This can be done by adding the line

```
58 fluent(above(B,L)) :- block(B), location(L).
```

and the following recursive definition:

```
59 holds(above(B,L),I) :- holds(on(B,L),I).
60 holds(above(B,L),I) :- holds(on(B,B1),I),
61                        holds(above(B1,L),I).
62 %% CWA
63 -holds(above(B,L),I) :- block(B), location(L), step(I),
64                         not holds(above(B,L),I).
```

This is very similar to other recursive definitions we discussed in Chapter 4, such as the definition of ancestors in Section 4.1.3 or subclasses in Section 4.3. But definitions of fluents in dynamic domains are slightly more subtle. To see the problem, let us consider program `blocks2.lp` given by lines 1–64 and the following statement:

```
:- -holds(above(b2,b0),1).
```

Since intuitively $\neg holds(above(b2, b0), 1)$ should be true, the resulting program should be inconsistent. But this is not what happens: The solver returns an answer set containing $holds(above(b2, b0), 1)$. The problem is caused by the unintended interplay between the inertia axiom and the CWA part of the definition of *above*. Informal reasoning that corresponds to this answer set goes as follows: $b2$ is above $b0$ at step 0. The inertia axiom allows us to conclude that $b2$ is still above $b0$ even after $b2$ is put on the table. The CWA from the definition is blocked, the constraint is satisfied, and the answer set is found. Of course, if in the attempt to construct an answer set we were to apply the CWA first, then the inertia axiom would be blocked, and we would not be able to satisfy the constraint. (It may be instructive to check that program `blocks2.lp` without the constraint has two answer sets: one that contains $holds(above\ (b2,\ b0),\ 1)$ and one that contains $\neg holds(above\ (b2,\ b0),\ 1)$.

To remedy the problem, it is sufficient to notice that the new fluent, *above*, is uniquely defined by the values of fluent *on* and therefore should not be made subject to the inertia axiom. This observation suggests a division of fluents of our domain into two classes – **inertial** and **defined**. Intuitively, an inertial fluent is subject to the law of inertia; its value can be (directly or indirectly) changed by an action. If no such action occurs, the value of the fluent remains unchanged. A defined fluent is not subject to the inertia axiom and cannot be directly caused by any action; instead it is defined in terms of other fluents.

To reflect the division of fluents into two types, let's create `blocks3.lp` by modifying `blocks2.lp` as follows:

1. Replace the definition of fluent on line 14 by

   ```
   fluent(inertial,on(B,L)) :- block(B), location(L).
   ```

2. Change the rules stating the inertia axiom (lines 43–49) to allow their application only to inertial fluents by adding condition

   ```
   fluent(inertial,F)
   ```

   to their bodies.
3. List *above* as a defined fluent in our list of fluents by replacing line 58 with

   ```
   fluent(defined,above(B,L)) :- block(B), location(L).
   ```

Now the notion of *above* is properly introduced. Run `blocks3.lp` and check that the results correspond to our intuition.

## 8.2 A General Solution

The knowledge base from the example in the previous section defines a dynamic system containing all possible trajectories of the blocks world. In our general theory of actions and change, a dynamic system is modeled by a **transition diagram** – a directed graph whose nodes correspond to physically possible states of the domain and whose arcs are labeled by actions. Such models are called Markovian.

A transition $\langle \sigma_0, \{a_1, \ldots, a_k\}, \sigma_1 \rangle$ of the diagram, where $\{a_1, \ldots, a_k\}$ is a set of actions executable in state $\sigma_0$, indicates that $\sigma_1$ may be a result of simultaneous execution of these actions in $\sigma_0$. Our representation guarantees that the effect of an action depends only on the state in which that action was executed. The way in which this state was reached is irrelevant.

A path $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ of the diagram represents a possible trajectory of the system with initial state $\sigma_0$ and final state $\sigma_n$. The transition diagram for a system contains all possible trajectories of that system.

In the dynamic system from the blocks-world example, states correspond to configurations of blocks satisfying the constraints from lines 36–42. Actions are of the form $put(Block, Location)$. Program rules on lines 34–55 define the corresponding state transitions.

A system may often have a large and complex diagram. The problem of finding its concise and mathematically accurate description is not trivial and has been a subject of research for more that 30 years. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents. An additional level of complexity is added by the need to specify *what is not changed by actions*. As noted by John McCarthy, the problem of finding a concise and accurate representation of this statement in a formal language, known as the **Frame Problem**, can be reduced to finding a representation of the inertia axiom – a default that states that things normally stay as they are. In our blocks-world example, we represented this axiom by logic programming rules on lines 43–49. Notice that the methodology of representing defaults from Chapter 5 was instrumental in solving this problem.

As we have seen, causal effects of actions can be defined by causal laws of the form:

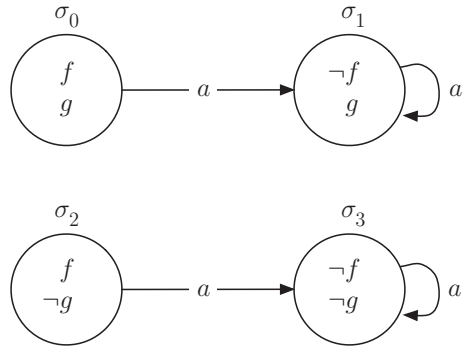
$$a \textbf{ causes } f \textbf{ if } p_0, \ldots, p_m.$$

The law says that *action $a$, executed in a state satisfying conditions $p_0, \ldots, p_m$, causes fluent $f$ to become true in the resulting state.* We saw the use of such laws in our blocks-world example when we defined the effect of action $put(Block, Location)$.

Consider another, simpler example of a dynamic system whose states are described by two Boolean inertial fluents, $f$ and $g$, and whose arcs are labeled by one action $a$ whose effect is described by a single causal law, $a$ **causes** $\neg f$ **if** $f$. We denote this description of our system by $\mathcal{D}_0$. Common sense suggests that if $a$ is executed in $\sigma_0 = \{f, g\}$, the new, successor state will contain $\neg f$ implied by the causal law. Note also that $g$ will remain true by the inertia axiom. Figure 8.3 shows the transition diagram for $\mathcal{D}_0$.

Causal and other relations between fluents can be described by state constraints – statements of the form

$$f \textbf{ if } p_0, \ldots, p_m$$




$\mathcal{F} = \{f, g\}$ $\mathcal{A} = \{a\}$
$a$ **causes** $\neg f$ **if** $f$

Figure 8.3. Transition Diagram of System $\mathcal{D}_0$

that say that *every state satisfying conditions* $p_0, \ldots, p_m$ *must also satisfy* $f$. They are used to define indirect effects of actions. Finding concise ways of defining these effects is called the **Ramification Problem**. Together with the frame problem discussed earlier the ramification problem caused substantial difficulties for researchers in their attempts to precisely define transitions of discrete dynamic systems.

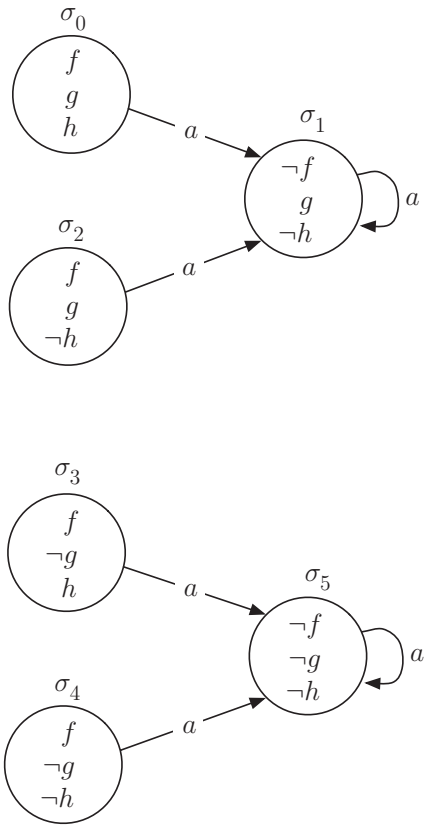
As we have seen, blocks-world rules on lines 36–42 are examples of state constraints. Note that they are not dependent on actions.

To illustrate this feature, let us expand description $\mathcal{D}_0$ by adding an inertial fluent $h$ and state constraint $\neg h$ **if** $\neg f$. Figure 8.4 shows the transition diagram of $\mathcal{D}_1$. State $\sigma_1$ contains direct effect $\neg f$ of $a$ derived by the causal law, $g$ derived by inertia, and indirect effect $\neg h$ of $a$ derived by the new state constraint.

Executability conditions are represented by laws of the form

**impossible** $a_0, \ldots, a_k$ **if** $p_0, \ldots, p_m$

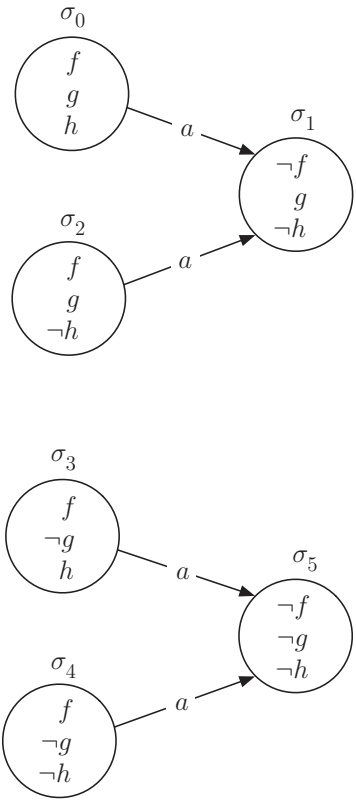
that say that *it is impossible to execute actions* $a_0, \ldots, a_k$ *simultaneously in a state satisfying conditions* $p_0, \ldots, p_m$. To illustrate, let's create $\mathcal{D}_2$ by expanding $\mathcal{D}_1$ by executability condition **impossible** $a$ **if** $\neg f$, which says that it is impossible to perform action $a$ in any state that contains fluent $\neg f$. Its transition diagram is shown in Figure 8.5. The diagram differs from Figure 8.4 in that it has fewer transitions; namely, the cycles in $\sigma_1$ and $\sigma_5$ have been eliminated.




$$\mathcal{F} = \{f, g, h\} \quad \mathcal{A} = \{a\}$$
$$a \textbf{ causes } \neg f \textbf{ if } f$$
$$\neg h \textbf{ if } \neg f$$

Figure 8.4. Transition Diagram of System $\mathcal{D}_1$

Descriptions $\mathcal{D}_0$, $\mathcal{D}_1$, and $\mathcal{D}_2$ can be viewed as theories in action language $\mathcal{AL}$. **Action languages** are formal models of parts of natural language used for describing the behavior of dynamic systems. Another way to look at them is as tools for describing transition diagrams. These examples have given a brief introduction to the syntax of these languages. The semantics was given by our intuitive understanding of what these laws might mean. Now we give a formal, mathematical definition of the syntax and semantics of $\mathcal{AL}$.




$\mathcal{F} = \{f, g, h\}$ $\mathcal{A} = \{a\}$
$a$ **causes** $\neg f$ **if** $f$
$\neg h$ **if** $\neg f$
**impossible** $a$ **if** $\neg f$

Figure 8.5. Transition Diagram of System $\mathcal{D}_2$

## 8.3 $\mathcal{AL}$ Syntax

Let us begin with some basic terminology. Action language $\mathcal{AL}$ is parametrized by a sorted signature containing three special sorts: *statics*, *fluents*, and *actions*. The fluents are partitioned into two sorts: *inertial* and *defined*. We refer to both statics and fluents as **domain properties**. A **domain literal** is a domain property $p$ or its negation $\neg p$. If domain literal $l$ is formed by a fluent, we refer to it as a **fluent literal**; otherwise it is a **static literal**.

A set $S$ of domain literals is called **complete** if for any domain property $p$ either $p$ or $\neg p$ is in $S$; $S$ is called **consistent** if there is no $p$ such that $p \in S$ and $\neg p \in S$.

**Definition 8.3.1.** *(Statements of $\mathcal{AL}$)*
*Language $\mathcal{AL}$ allows the following types of statements:*

1. *Causal Laws:*

$$a \textbf{ causes } l_{in} \textbf{ if } p_0, \ldots, p_m$$

2. *State Constraints:*

$$l \textbf{ if } p_0, \ldots, p_m$$

3. *Executability Conditions:*

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p_0, \ldots, p_m$$

*where $a$ is an action, $l$ is an arbitrary domain literal, $l_{in}$ is a literal formed by an inertial fluent, $p_0, \ldots, p_m$ are domain literals, $k \geq 0$, and $m \geq -1$.*[6] *Moreover, no negation of a defined fluent can occur in the heads of state constraints.*

The collection of state constraints whose head is a defined fluent $f$ is referred to as the *definition* of $f$. As in logic programming definitions, $f$ is true if it follows from the truth of the body of at least one of its defining rules. Otherwise, $f$ is false.

**Definition 8.3.2.** *(System Description)*
*A* **system description** *of $\mathcal{AL}$ is a collection of statements of $\mathcal{AL}$.*

## 8.4 $\mathcal{AL}$ Semantics – The Transition Relation

A system description $\mathcal{SD}$ serves as a specification of the transition diagram $\mathcal{T}(\mathcal{SD})$ defining all possible trajectories of the corresponding dynamic system. Therefore, to define the semantics of $\mathcal{AL}$, we have to precisely define the states and legal transitions of this diagram.

### *8.4.1 States*

We start with the states. If $\mathcal{SD}$ does not contain defined fluents, the definition of a state is simple – a state is simply a complete and consistent set of domain literals satisfying state constraints of $\mathcal{SD}$. This definition was used earlier

[6] If $m = -1$, keyword **if** is omitted.

in examples $\mathcal{D}_0$, $\mathcal{D}_1$, and $\mathcal{D}_2$. For system descriptions with defined fluents the situation is more subtle. The following simple example illustrates the problem.

**Example 8.4.1.** *(State)*
Let us consider a system description $\mathcal{D}_3$ with two inertial fluents, $f$ and $g$, and a fluent $h$ defined by the following rules:

$$h \textbf{ if } f$$
$$h \textbf{ if } \neg g.$$

Clearly, $\{f, g, h\}$ is a state of $\mathcal{D}_3$ and $\{f, g, \neg h\}$ is not. But what about $\{\neg f, g, h\}$? It does not seem to satisfy the definition of $h$ since the truth of $h$ does not follow from any of its defining rules. So the intended answer to our question seems to be *no*. But $\{\neg f, g, h\}$ satisfies the constraints! The suggested definition does not work. It may be tempting to consider an additional condition requiring a defined fluent to be true *iff* at least one of its defining rules is satisfied. This would give the correct answer to our example, but would not work if we were to expand the definition by an extra rule:

$$h \textbf{ if } h.$$

The tautological rule should not change the states of $\mathcal{D}_3$, but it does. According to the suggested modification, $\{\neg f, g, h\}$ is a state. An attentive reader will notice the similarity between this situation and the use of Clark's completion for defining semantics of logic programs. This similarity led to *the idea of defining states via logic programming under answer set semantics*, as shown later.

We now need the following notation. By $\Pi_c(\mathcal{SD})$ (where $c$ stands for constraints) we denote the logic program defined as follows:

1. For every state constraint

$$l \textbf{ if } p$$

$\Pi_c(\mathcal{SD})$ contains

$$l \leftarrow p.$$

2. For every defined fluent $f$, $\Pi_c(\mathcal{SD})$ contains the CWA:

$$\neg f \leftarrow \ not\ f.$$

For any set $\sigma$ of domain literals let $\sigma_{nd}$ denote the collection of all domain literals of $\sigma$ formed by inertial fluents and statics. (The $_{nd}$ stands for nondefined.)

**Definition 8.4.1.** *A complete and consistent set $\sigma$ of domain literals is a* **state** *of the transition diagram defined by a system description $\mathcal{SD}$ if $\sigma$ is the unique answer set of program $\Pi_c(\mathcal{SD}) \cup \sigma_{nd}$.*

In other words, a state is a complete and consistent set of literals $\sigma$ that is the unique answer set of the program that consists of the nondefined literals from $\sigma$, the encoding of the state constraints, and the CWA for each defined fluent. *Note that (a) every state of system description $\mathcal{SD}$ satisfies the state constraints of $\mathcal{SD}$ and (b) if the signature of $\mathcal{SD}$ does not contain defined fluents, a state is simply a complete, consistent set of literals satisfying the state constraints of $\mathcal{SD}$.*

**Example 8.4.2.** *(Example 8.4.1 Revisited)*
Let us consider the system description $\mathcal{D}_3$ from Example 8.4.1. Program $\Pi_c(\mathcal{SD})$ consists of these rules:

$$h \leftarrow f.$$
$$h \leftarrow \neg g.$$
$$\neg h \leftarrow \ not\ h.$$

It is easy to check that the transition diagram defined by $\mathcal{D}_3$ has the following states: $\{f, g, h\}$, $\{f, \neg g, h\}$, $\{\neg f, \neg g, h\}$, $\{\neg f, g, \neg h\}$. To check that $\sigma_0 = \{f, \neg g, h\}$ is a state, it is sufficient to check that $\sigma_0$ is the only answer set of $\Pi_c(\mathcal{SD}) \cup \{f, \neg g\}$. The process is similar for other states. To see that $\sigma = \{\neg f, g, h\}$ is not a state, it suffices to see that $\sigma$ is not the answer set of $\Pi_c(\mathcal{SD}) \cup \{\neg f, g\}$.

The next example explains the importance of the uniqueness requirement of the definition.

**Example 8.4.3.** *(Mutually Recursive Laws)*
Let us consider a system description $\mathcal{D}_4$ with two defined fluents, $f$ and $g$, which are defined by the following mutually recursive laws:

$$g \textbf{ if } \neg f.$$
$$f \textbf{ if } \neg g.$$

Let us check if $\{f, \neg g\}$ is a state of $\mathcal{T}(\mathcal{D}_4)$. Program $\Pi_c(\mathcal{D}_4)$ from Definition 8.4.1 consists of these rules:

$$
\begin{array}{l}
g \leftarrow \neg f. \\
f \leftarrow \neg g. \\
\neg g \leftarrow \ not\ g. \\
\neg f \leftarrow \ not\ f.
\end{array}
$$

Since all the fluents of $\mathcal{D}_4$ are defined, $\sigma_{nd} = \emptyset$ and program $\Pi_c(\mathcal{D}_4) \cup \sigma_{nd}$ has two answer sets, $\{f, \neg g\}$ and $\{g, \neg f\}$. This violates the uniqueness condition of Definition 8.4.1, and hence $\mathcal{T}(\mathcal{D}_4)$ has no states. This result is intended. Mutually recursive laws of $\mathcal{D}_4$ are not strong enough to uniquely define $f$ and $g$; thus, the definition is rejected.

We conclude our definition of state by giving a sufficient condition that guarantees that defined fluents of a system description are uniquely defined by the system's statics and inertial fluents. To mathematically capture this property we use the following definition:

**Definition 8.4.2.** *(Well-Founded System Description)*
*A system description $\mathcal{SD}$ of $\mathcal{AL}$ is called* **well founded** *if for any complete and consistent set of fluent literals $\sigma$ satisfying the state constraints of $\mathcal{SD}$, the program*

$$\Pi_c(\mathcal{SD}) \cup \sigma_{nd} \tag{8.1}$$

*has at most one answer set.*

We now also need the following notions:

**Definition 8.4.3.** *(Fluent Dependency Graph)*
*The* **fluent dependency graph** *of a system description $\mathcal{SD}$ is the directed graph such that*

- *Its vertices are arbitrary domain literals.*
- *It has an edge*
  - *from $l$ to $l'$ if $l$ is formed by a static or an inertial fluent and $\mathcal{SD}$ contains a state constraint with the head $l$ and the body containing $l'$,*
  - *from $f$ to $l'$ if $f$ is a defined fluent and $\mathcal{SD}$ contains a state constraint with the head $f$ and the body containing $l'$ and not containing $f$.*
  - *from $\neg f$ to $f$ for every defined fluent $f$.*

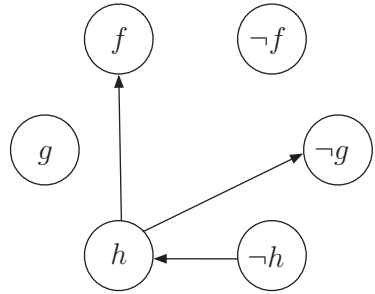



Figure 8.6. Fluent Dependency Graph for Example 8.4.2

The fluent dependency graphs for Examples 8.4.2 and 8.4.3 are given in Figures 8.6 and 8.7, respectively. Recall that $h$ is the defined fluent in the first example, whereas $f$ and $g$ are the defined fluents in the second.

**Definition 8.4.4.** *(Weak Acyclicity)*
*A fluent dependency graph is* **weakly acyclic** *if it does not contain paths from defined fluents to their negations. By extension, a system description with a weakly acyclic fluent dependency graph is also called* weakly acyclic.

Consequently, as expected, the graph in Figure 8.6 is weakly acyclic, whereas the one in Figure 8.7 is not.

**Proposition 8.4.1.** *(Sufficient Condition for Well-Foundedness)*
*If a system description $\mathcal{SD}$ of $\mathcal{AL}$ is weakly acyclic then $\mathcal{SD}$ is well-founded.*

### *8.4.2 Transitions*

Our *definition of transition relation of $\mathcal{T}(\mathcal{SD})$ is also based on the notion of the answer set of a logic program*. To describe a transition $\langle \sigma_0, a, \sigma_1 \rangle$ we construct a program $\Pi(\mathcal{SD}, \sigma_0, a)$ consisting of logic programming encodings of the system description $\mathcal{SD}$, initial state $\sigma_0$, and set of actions

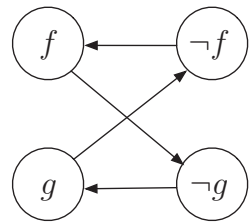



Figure 8.7. Fluent Dependency Graph for Example 8.4.3

$a$, such that answer sets of this program determine the states the system can move into after the execution of $a$ in $\sigma_0$.

**Definition 8.4.5.** *The encoding $\Pi(\mathcal{SD})$ of system description $\mathcal{SD}$ consists of the encoding of the signature of $\mathcal{SD}$ and rules obtained from statements of $\mathcal{SD}$.*

- **Encoding of the Signature**
  *We start with the encoding $sig(\mathcal{SD})$ of the signature of $\mathcal{SD}$.*
  - *For each constant symbol $c$ of sort $sort\_name$ other than $fluent$, $static$ or $action$, $sig(\mathcal{SD})$ contains*

$$sort\_name(c). \tag{8.2}$$

  - *For every static $g$ of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains*

$$static(g). \tag{8.3}$$

  - *For every inertial fluent $f$ of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains*

$$fluent(inertial, f). \tag{8.4}$$

  - *For every defined fluent $f$ of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains*

$$fluent(defined, f). \tag{8.5}$$

  - *For every action $a$ of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains*

$$action(a). \tag{8.6}$$

- **Encoding of Statements of $\mathcal{SD}$**
  *For this encoding we only need two steps, $0$ and $1$, which stand for the beginning and the end of a transition. This is sufficient for describing a single transition; however, later, we describe longer chains of events and let steps range over $[0, n]$ for some constant $n$. To allow an easier generalization of the program we encode steps by using constant $n$ for the maximum number of steps, as follows:*

$$\#const\ n = 1. \tag{8.7}$$

$$step(0..n). \tag{8.8}$$

  *As in our blocks-world example, we introduce a relation $holds(f, i)$ that says that fluent $f$ is true at step $i$. To simplify the description of the encoding, we also introduce a new notation, $h(l, i)$ where $l$ is a domain literal and $i$ is a step. If $f$ is a fluent then by $h(l, i)$ we denote $holds(f, i)$ if $l = f$ or $\neg holds(f, i)$ if $l = \neg f$. If $l$ is a static*

*literal then $h(l, i)$ is simply $l$. We also need relation $occurs(a, i)$ that says that action $a$ occurred at step $i$; $occurs(\{a_0, \ldots, a_k\}, i) =_{def} \{occurs(a_i) : 0 \leq i \leq k\}$.*

*We use this notation to encode statements of $\mathcal{SD}$ as follows:*

– *For every causal law*

$$a \textbf{ causes } l \textbf{ if } p_0, \ldots, p_m$$

$\Pi(\mathcal{SD})$ *contains*

$$\begin{aligned} h(l, I+1) \leftarrow\ & h(p_0, I), \ldots, h(p_m, I), \\ & occurs(a, I), \\ & I < n. \end{aligned} \tag{8.9}$$

– *For every state constraint*

$$l \textbf{ if } p_0, \ldots, p_m$$

$\Pi(\mathcal{SD})$ *contains*

$$h(l, I) \leftarrow h(p_0, I), \ldots, h(p_m, I). \tag{8.10}$$

– $\Pi(\mathcal{SD})$ *contains the CWA for defined fluents:*

$$\begin{aligned} \neg holds(F, I) \leftarrow\ & fluent(defined, F), \\ & not\ holds(F, I). \end{aligned} \tag{8.11}$$

– *For every executability condition*

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p_0, \ldots, p_m$$

$\Pi(\mathcal{SD})$ *contains*

$$\neg occurs(a_0, I)\ or\ \ldots\ or\ \neg occurs(a_k, I) \leftarrow h(p_0, I), \ldots, h(p_m, I). \tag{8.12}$$

– $\Pi(\mathcal{SD})$ *contains the inertia axiom:*

$$\begin{aligned} holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & holds(F, I), \\ & not\ \neg holds(F, I+1), \\ & I < n. \end{aligned} \tag{8.13}$$

$$\begin{aligned} \neg holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & \neg holds(F, I), \\ & not\ holds(F, I+1), \\ & I < n. \end{aligned} \tag{8.14}$$

– $\Pi(\mathcal{SD})$ *contains CWA for actions:*

$$\neg occurs(A, I) \leftarrow\ not\ occurs(A, I). \tag{8.15}$$

*This completes the construction of encoding* $\Pi(\mathcal{SD})$ *of system description* $\mathcal{SD}$.

To continue with our definition of transition $\langle \sigma_0, a, \sigma_1 \rangle$ we describe the two remaining parts of program $\Pi(\mathcal{SD}, \sigma_0, a)$ – the encoding $h(\sigma_0, 0)$ of initial state $\sigma_0$ and the encoding $occurs(a, 0)$ of action $a$:

$$h(\sigma_0, 0) =_{def} \{h(l, 0) : l \in \sigma_0\}$$

and

$$occurs(a, 0) =_{def} \{occurs(a_i, 0) : a_i \in a\}.$$

To complete program $\Pi(\mathcal{SD}, \sigma_0, a)$ we simply gather our description of the system's laws, together with the description of the initial state and the actions that occur in it.

**Definition 8.4.6.**

$$\Pi(\mathcal{SD}, \sigma_0, a) =_{def} \Pi(\mathcal{SD}) \cup h(\sigma_0, 0) \cup occurs(a, 0).$$

Now we are ready to define the notion of transition of $\mathcal{T}(\mathcal{SD})$.

**Definition 8.4.7.** *Let* $a$ *be a nonempty collection of actions and* $\sigma_0$ *and* $\sigma_1$ *be states of the transition diagram* $\mathcal{T}(\mathcal{SD})$ *defined by a system description* $\mathcal{SD}$. *A state-action-state triple* $\langle \sigma_0, a, \sigma_1 \rangle$ *is a* **transition** *of* $\mathcal{T}(\mathcal{SD})$ *iff* $\Pi(\mathcal{SD}, \sigma_0, a)$ *has an answer set* $A$ *such that* $\sigma_1 = \{l : h(l, 1) \in A\}$.

We now have a program that, like a rational human reasoner, can predict what the state of the world will be once an action is performed in a given state. In the next section, we give examples of specific domains that can be described by $\mathcal{AL}$ and how they can be translated into ASP programs using Definition 8.4.6.

## 8.5 Examples

As we have seen, to model a dynamic domain, we need to describe what actions cause what effects under what conditions. To do this, we need to identify

1. the objects, properties, and actions of the domain;
2. the relationships between the properties;
3. the executability conditions and causal effects of actions.

In other words, we must come up with an $\mathcal{AL}$ system description for our domain. In this section we give several examples of such descriptions and walk through the steps of constructing the corresponding transition diagrams.

### 8.5.1 The Briefcase Domain

Consider a briefcase with two clasps. We have an action, toggle, which moves a given clasp into the up position if the clasp is down, and vice versa. If both clasps are in the up position, the briefcase is open; otherwise, it is closed. Create a (simple) model of this domain.

The signature of the briefcase domain consists of sort $clasp = \{1, 2\}$, inertial fluent $up(C)$ that holds iff clasp $C$ is up, defined fluent $open$ that holds iff both clasps are up, and action $toggle(C)$ that toggles clasp $C$.

The system description $\mathcal{D}_{bc}$ of our domain consists of axioms

$$\begin{aligned} &toggle(C) \ \textbf{causes} \ up(C) \ \textbf{if} \ \neg up(C) \\ &toggle(C) \ \textbf{causes} \ \neg up(C) \ \textbf{if} \ up(C) \\ &open \ \textbf{if} \ up(1), up(2) \end{aligned}$$

where $C$ ranges over the sort *clasp*. Since these laws contain variables, they are not, strictly speaking, proper statements of our action language; such laws are often referred to as **schemas**. Individual laws can be obtained from schemas by grounding the variables. Since our signature is sorted and variable $C$ ranges over clasps, the grounding will respect this sorting information and replace $C$ by 1 and 2. For instance, the first schema can be viewed as shorthand for two laws:

$$\begin{aligned} &toggle(1) \ \textbf{causes} \ up(1) \ \textbf{if} \ \neg up(1) \\ &toggle(2) \ \textbf{causes} \ up(2) \ \textbf{if} \ \neg up(2). \end{aligned}$$

Now let us figure out what the states of our domain look like. According to Definition 8.4.1 we need a program $\Pi_c(\mathcal{D}_{bc})$ that consists of the

following two rules:

$$open \leftarrow up(1), up(2) \tag{1}$$

$$\neg open \leftarrow \ not\ open. \tag{2}$$

Consider a collection

$$\sigma = \{\neg up(1), up(2), \neg open\}.$$

Is this a state? First we need to check if it is complete and consistent. By definition, it is. Next we need to consider

$$\sigma_{nd} = \{\neg up(1), up(2)\}$$

and check if $\sigma$ is the only answer set of the program $\Pi_c(\mathcal{D}_{bc}) \cup \sigma_{nd}$ consisting of rules (1) and (2) given earlier, and facts $\neg up(1)$ and $up(2)$. Clearly, it is, and hence, as expected, $\sigma$ is a state of our transition diagram.

Now let $\sigma$ be $\{\neg up(1), up(2), open\}$. According to the description of the briefcase, this is a physical impossibility and hence should not be a state. Indeed this is the case according to our definition. Although it is complete and consistent, it is not the answer set of the program consisting of rules (1) and (2) and facts $\neg up(1)$ and $up(2)$. Therefore, $\sigma$ is not a state.

Moving on to defining transitions of our system, we construct program $\Pi(\mathcal{D}_{bc})$, which will be written in the syntax of Gringo and named `bc.lp`. In its construction we slightly deviate from the standard encoding in Definition 8.4.5 to avoid a long listing of fluents, actions, and other sorts. Instead we define sorts using logic programming rules with variables. Definitions of fluents and actions in the following program can serve as an example of this technique. Of course, the use of rules is just a matter of convenience; they can be eliminated and replaced by collections of atoms. Here is the translation of our system description $\mathcal{D}_{bc}$ into a logic program:

```
1 %% Domain Signature
2 clasp(1).
3 clasp(2).
4
5 fluent(inertial, up(C)) :- clasp(C).
6 fluent(defined, open).
7 action(toggle(C)) :- clasp(C).
```

Clearly the definition of inertial fluent

```
fluent(inertial, up(C)) :- clasp(C).
```

can be replaced by two atoms

```
fluent(inertial, up(1)).
fluent(inertial, up(2)).
```

The same can be done for actions. This completes our definition of sort $clasp$ and the fluents and actions of the domain.

Next, we translate our axioms. The first two axioms of our system description are causal laws with variables. It is not difficult to see that rule (8.9) from Definition 8.4.5 can be adapted to apply to such laws. As the result we obtain the following encoding where $C$ ranges over sort $clasp$:

```
#const n = 1.
step(0..n).

%% toggle(C) causes up(C) if -up(C)
holds(up(C),I+1) :- occurs(toggle(C),I),
                    -holds(up(C), I),
                    I < n.

%% toggle(C) causes -up(C) if up(C)
-holds(up(C),I+1) :- occurs(toggle(C),I),
                     holds(up(C),I),
                     I < n.
```

The last law of $\mathcal{D}_{bc}$ is a state constraint. Using rule (8.10) we get

```
%% open if up(1), up(2).
holds(open,I) :- holds(up(1),I),
                 holds(up(2),I).
```

We add the closed world assumption for defined fluents as dictated by rule (8.11):

```
%% CWA for Defined Fluents
-holds(F,I) :- fluent(defined,F),
               step(I),
               not holds(F,I).
```

(Notice that, in accordance with the general translation, this rule encodes the CWA for all defined fluents. Currently, we only have one such fluent, but we might wish to add more. The CWA for defined fluents is taken care of once and for all.)

Then we add the inertia axiom as defined by rules (8.13) and (8.14), allowing us to derive values of inertial fluents, even if no action explicitly dictated their change.

```
%% General Inertia Axiom
holds(F,I+1) :- fluent(inertial,F),
                holds(F,I),
                not -holds(F,I+1),
                I < n.
-holds(F,I+1) :- fluent(inertial,F),
                 -holds(F,I),
                 not holds(F,I+1),
                 I < n.
```

Last we add the closed world assumption for actions as given by rule (8.15):

```
%% CWA for actions
-occurs(A,I) :- action(A), step(I),
                not occurs(A,I).
```

This concludes our encoding of the briefcase domain.

Figure 8.8 shows the transition diagram for this system. Using this program one can check, for instance, that the system contains transitions

$\langle\{\neg up(1), up(2), \neg open\}, toggle(1), \{up(1), up(2), open\}\rangle,$
$\langle\{up(1), up(2), open\}, toggle(1), \{\neg up(1), up(2), \neg open\}\rangle,$
$\langle\{\neg up(1), \neg up(2), \neg open\}, \{toggle(1), toggle(2)\}, \{up(1), up(2), open\}\rangle,$
etc.

Simply add the appropriate information about the initial state and the action that occurred in it as in Definition 8.4.7. For example, to check the first transition, add these statements:

```
%% Initial Situation
-holds(up(1),0).
holds(up(2),0).
-holds(open,0).
%% Action
occurs(toggle(1),0).
%% Display
#show holds/2.
#show -holds/2.
```

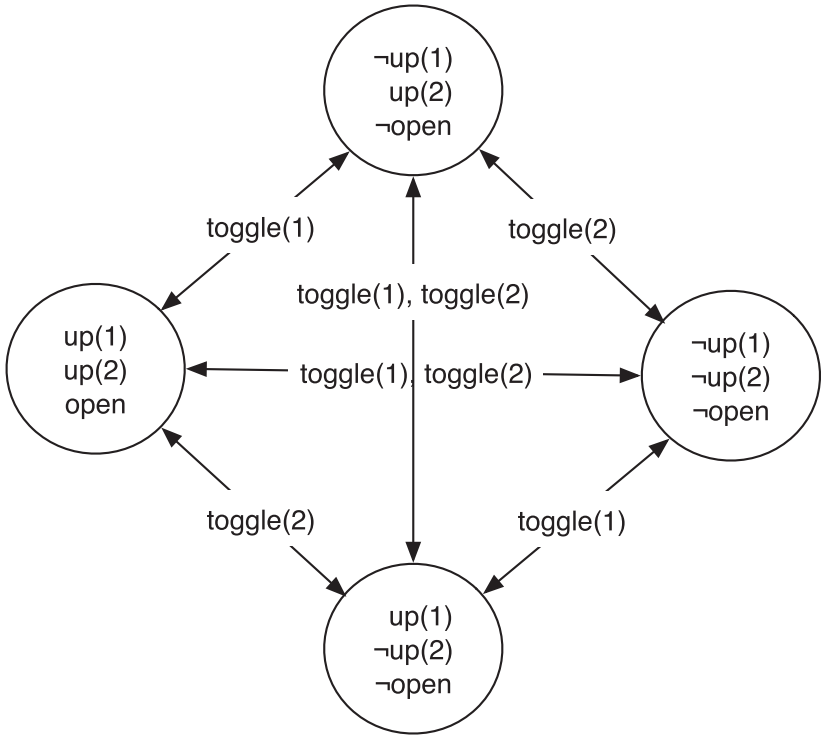



Figure 8.8. The Two-Clasp Briefcase Domain

and invoke the program to get the values of the fluents. The transition with $toggle(1)$ and $toggle(2)$ occurring simultaneously can be tested in the same manner by using two actions, $occurs(toggle(1), 0)$ and $occurs(toggle(2), 0)$.

Note that the encoding of our initial state can be simplified by omitting `-holds(open,0)`. This is true because *open* is a defined fluent and its value is computed by the corresponding default of $\Pi(\mathcal{D}_{bc})$. This property, of course, holds in general, and from now on we do not include the encoding of the values of defined fluents in the encoding of a state.

### 8.5.2 The Blocks World Revisited

Let's go back to our basic blocks-world example from Section 8.1, with two fluents $on$ and $above$, and construct a system description $\mathcal{D}_{bw}$ in $\mathcal{AL}$ representing the blocks-world transition diagram.

The signature of $\mathcal{D}_{bw}$ consists of sorts $block = \{b_0 \ldots b_7\}$ and $location = \{t\} \cup block$, inertial fluent $on(block, location)$, defined fluent

$above(block, location)$, and action $put(block, location)$. We assume that $put(B, L)$ is an action only if $B \neq L$.

The laws of the blocks world are

1. $put(B, L)$ **causes** $on(B, L)$
2. $\neg on(B, L_2)$ **if** $on(B, L_1), L_1 \neq L_2$
3. $\neg on(B_2, B)$ **if** $on(B_1, B), B_1 \neq B_2$
4. $above(B, L)$ **if** $on(B, L)$
5. $above(B, L)$ **if** $on(B, B1), above(B1, L)$
6. **impossible** $put(B, L)$ **if** $on(B_1, B)$
7. **impossible** $put(B_1, B)$ **if** $on(B_2, B)$

where (possibly indexed) $B$s and $L$s stand for blocks and locations, respectively.

Now let us translate $\mathcal{D}_{bw}$ into the corresponding logic program `bw.lp`. The collection of blocks is represented by

```
block(b0).  block(b1).  block(b2).  block(b3).
block(b4).  block(b5).  block(b6).  block(b7).
```

Instead of listing the collection of facts defining sort $location$, we save ourselves some typing and define locations using logic programming rules

```
location(X) :- block(X).
location(t).
```

We also define fluents, actions, and steps:

```
fluent(inertial, on(B,L)) :- block(B), location(L).
fluent(defined, above(B,L)) :- block(B), location(L).
action(put(B,L)) :- block(B), location(L),
                    B != L.

#const n = 1.
step(0..n).
```

Now let's use the encoding from Definition 8.4.5 to encode the blocks-world laws. As in the briefcase example, we modify this encoding for dealing with variables. In doing so we assume that Bs and Ls range over blocks and locations, respectively, and that $put(B, L)$ occurring in the rules represents actions (i.e., $B \neq L$). Note that, strictly speaking, these conditions should be enforced by adding sort information (including $action(put(B, L))$) in the bodies of our rules. Otherwise, the ground instances of the rule may express meaningless statements about the results of putting a table on a block, for example, which may lead to misleading

answers if the program is used together with statements expressing impossible actions. The same goes for fluents. To simplify the presentation we assume that the program will always be used with meaningful input. Doing so allows us to ignore sorts for actions and fluents in rules of the translation. Of course, we will add sort information, including that about blocks and locations, to some rules to reflect the meaning of the laws and the usual considerations about the rules' safety.

Law 1 is a causal law with fluent $on(B, L)$ and action $put(B, L)$. Following the directions for causal laws, we use encoding (8.9) :

```
holds(on(B,L),I+1) :- occurs(put(B,L),I),
                      I < n.
```

Laws 2–5 are typical state constraints, and we translate them using (8.10) as follows:

```
-holds(on(B,L2),I) :- holds(on(B,L1),I),
                      location(L2),
                      L1 != L2.

-holds(on(B2,B),I) :- holds(on(B1,B),I),
                      block(B),
                      block(B2),
                      B1 != B2.

holds(above(B2,B1),I) :- holds(on(B2,B1),I).

holds(above(B2,B1),I) :- holds(on(B2,B),I),
                         holds(above(B,B1),I).
```

Using (8.12), executability conditions from Laws 6 and 7 become

```
-occurs(put(B,L),I) :-  location(L),
                        holds(on(B1,B),I).

-occurs(put(B1,B),I) :- block(B1), block(B),
                        holds(on(B2,B),I).
```

Last, we add the rules needed for all domains: the CWA for defined fluents (8.11), the inertia axiom (8.13 and 8.14), and the CWA for actions (8.15):

```
%% CWA for Defined Fluents

-holds(F,I) :- fluent(defined,F), step(I),
               not holds(F,I).
```

```
36
37 %% General Inertia Axiom
38
39 holds(F,I+1) :- fluent(inertial,F),
40                  holds(F,I),
41                  not -holds(F,I+1),
42                  I < n.
43
44 -holds(F,I+1) :- fluent(inertial,F),
45                   -holds(F,I),
46                   not holds(F,I+1),
47                   I < n.
48
49 %% CWA for Actions
50
51 -occurs(A,I) :- action(A), step(I),
52                  not occurs(A,I).
```

This concludes the encoding `bw.lp` of system description $\mathcal{D}_{bw}$. The resulting program (lines 1–52) defines the transition diagram of the dynamic system associated with the blocks world.

You might have noticed that our translation algorithm produces the same encoding as that in `blocks3.lp`. This is, of course, not an accident – insights obtained by researchers representing the blocks world and similar problems in ASP led them to the development of $\mathcal{AL}$ and its translation given in Definition 8.4.5. So why use $\mathcal{AL}$? Notice that the description of the blocks world in $\mathcal{AL}$ is substantially shorter than that in ASP. The former does not explicitly mention steps and contains neither inertia axioms nor defaults. These details are hidden in the translation, which can easily be automated. The statements of $\mathcal{AL}$ have a rather clear, intuitive meaning and can be used by knowledge engineers who do not have a clear notion of default negation and other nontrivial ASP concepts. This abstraction should not come as a surprise to computer scientists who are by now very familiar with the idea of high-level languages and language translators.

### *8.5.3 Blocks World with Concurrent Actions*

So far we have dealt with a one-arm blocks-world domain. Suppose that we have two robotic arms capable of avoiding collisions in the air. Now that two blocks can be moved simultaneously, we must make sure that our robot can behave sensibly. Clearly the new system description $\mathcal{D}_{tbw}$ must contain all the statements of $\mathcal{D}_{bw}$, but we may also need some additional executability

conditions. Let's think about what problems our new system might encounter. For example, consider what would happen if both arms were to try to move the same block to two separate locations at the same time; that is, we should not allow $occurs(put(b2, t), 0)$ and $occurs(put(b2, b4), 0)$, even though each individual action is perfectly legal. It can be easily seen, however, that the laws of $\mathcal{D}_{bw}$ already prohibit those two actions from occurring simultaneously. If the two actions were executed in parallel, then by causal law (1), $b2$ would be located on both $b4$ and the table. This would contradict constraint (2), and hence, such a parallel execution is already prohibited. Indeed, if we combine `bw.lp` with the initial situation described in Figure 8.1 (encoded on lines 21–33 of the program in Section 8.1) and actions $occurs(put(b2, t), 0)$ and $occurs(put(b2, b4), 0)$, the program will be inconsistent. So far so good. Now suppose we instead want to simultaneously execute actions $put(b2, t)$ and $put(b7, b2)$. Replacing the previous actions by these ones would produce a program with the answer set containing $b2$ on the table and $b7$ on $b2$. Is this the answer you expected? If you assumed that our robot arms were coordinated in such a way as to be able to stack blocks while moving them and then to put them down, then you answered "yes." But if you, like us, are not as confident in the state of the art of robotics and considered the concurrent execution of these two actions to be physically impossible, then the program's answer is wrong. However, this incorrect answer is to be expected because nothing in $\mathcal{D}_{bw}$ prevents this from happening. To avoid the problem, let's teach the system that actions that put $B_1$ on $B_2$ and simultaneously move $B_2$ are impossible:

$$\textbf{impossible } put(B_1, L), put(B_2, B_1).$$

Translating into ASP we get

```
53 -occurs(put(B1,L),I) | -occurs(put(B2,B1),I) :-
54                                     step(I),
55                                     action(put(B1,L)),
56                                     action(put(B2,B1)).
```

(The sorts in the body of the rule are added for safety.)

To test the new system description let's add lines 53–56 to `bw.lp`, denote the resulting program by `twoarms.lp`, and compute the answer sets of `twoarms.lp` combined with the usual initial situation (lines 21–33 from Section 8.1) and statements

```
occurs(put(b2,t),0).
occurs(put(b7,b2),0).
```

The program should say that there are no models.

If instead we use the program together with concurrent actions

```
occurs(put(b2,t),0).
occurs(put(b4,b7),0).
```

we obtain the expected results. But what would happen if we were to attempt to simultaneously move $b4$ to $b7$ and $b2$ to $b1$? Physically, of course, these actions also depend on the degree of coordination of the robot's arms, but this degree of coordination seems reasonable. Our program, however, will not believe so. Not surprisingly it will tell us that such a move is impossible. This happens because constraint (7) does not allow us to stack a block on top of a block occupied at the beginning of the action. Can we allow such a move? Fortunately, the answer is yes. It is simply sufficient to remove constraint (7) from $\mathcal{D}_{bw}$. Of course we need to make sure that doing so would not allow some impossible moves. After all, the constraint was added for the purpose of prohibiting such moves. Some thought will show that no impossible moves will be allowed after all. Indeed this constraint was unnecessary from the beginning. Moving blocks $B_1$ and $B_2$ onto block $B$ would cause two blocks to be located on the same block, which is prohibited by law (3) of $\mathcal{D}_{bw}$.

What happened is a typical and rather frequently occurring problem called **overspecification**. The condition, which was simply unnecessary in the original situation, caused an unexpected problem when the domain was expanded. In other words overspecification can decrease the degree of elaboration tolerance of the program.

Experience in many areas of computer science shows that programmers must be careful to avoid unwanted side effects when allowing concurrent actions, but the payoff may be substantial. The discussion in this section demonstrates that our approach incorporates concurrent actions and restrictions on them naturally and intuitively. It requires no special accommodations to be made for concurrency other than those required by the nature of the domain. These accommodations, in turn, can be incorporated within the original framework by adding the required executability conditions.

## 8.6 Nondeterminism in $\mathcal{AL}$

A transition diagram $\mathcal{T}$ is called **deterministic** if for every state $\sigma$ and action $a$ there is at most one state $\sigma'$ such that $\langle \sigma, a, \sigma' \rangle \in \mathcal{T}$. Perhaps somewhat surprisingly, system descriptions of $\mathcal{AL}$ are capable of describing non-deterministic diagrams. Consider, for instance, the system described by the following laws:

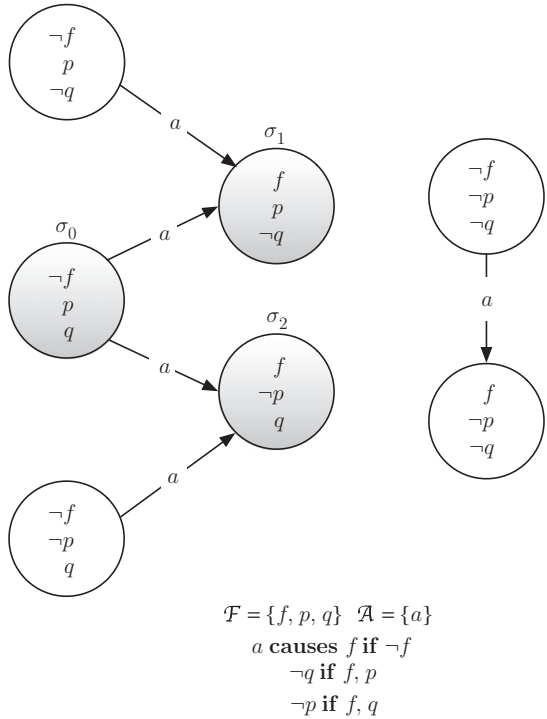



Figure 8.9. Nondeterministic System Description

1. $a$ **causes** $f$ **if** $\neg f$
2. $\neg q$ **if** $f, p$
3. $\neg p$ **if** $f, q$

where $f$, $q$, and $p$ are inertial fluents. Suppose action $a$ is executed in state $\sigma_0 = \{\neg f, p, q\}$. What would be the states the system can move into as a result of this execution? It is clear that every such state must contain $f$. But what about $p$? One may notice that there are two possibilities: First, $p$ may maintain its value by inertia. In this case the system will move into state $\sigma_1 = \{f, p, \neg q\}$ (where $q$ is false due to the second law of our system description). Symmetrically, $q$ may be preserved by inertia, and the system may move into $\sigma_2 = \{f, \neg p, q\}$. Figure 8.9 shows the possible transitions in the system; the nondeterministic transitions are shaded. One can easily check that these are exactly the transitions produced by our semantics.

The nondeterminism of the resulting system can be attributed to the incompleteness of our specification and not to the real nondeterminism of causal effects of action $a$. Yet experts disagree whether actions are really

nondeterministic or whether the impression of nondeterminism is caused by our lack of knowledge. We do not discuss these issues here. Instead we simply advise users of $\mathcal{AL}$ to make sure that their descriptions are sufficiently complete and, hence, the corresponding systems are deterministic.

## 8.7 Temporal Projection

In this chapter we introduced the syntax and semantics of action language $\mathcal{AL}$ that allow a concise and accurate description of transition diagrams of dynamic systems. The fact that the semantics of the language was given in terms of the semantics of ASP allows us to reduce answering questions about the values of fluents of the domain along a given trajectory to computing answer sets of simple logic programs. The task of predicting such values is often referred to as **temporal projection**. We already encountered this task in Section 8.1 where we dealt with predicting positions of blocks after a sequences of moves. The method is rather general. Given a system description $\mathcal{D}$, an initial state $\sigma_0$, and a sequence of consecutive occurrences of actions $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$, one can compute the trajectories of the system by combining $\Pi^n(\mathcal{D}) \cup h(\sigma_0, 0)$ with the encoding

$$\begin{gathered} occurs(a_0, 0). \\ \vdots \\ occurs(a_{n-1}, n-1). \end{gathered}$$

of the sequence and computing the answer sets of the resulting program $\Pi^n(\mathcal{D}, \sigma_0, \alpha)$. It is not difficult to show that a sequence

$$\langle \sigma_0, a_0, \ldots, a_{n-1}, \sigma_n \rangle$$

is a trajectory of $\mathcal{D}$ defined by sequence $\alpha$ iff there is an answer set $A$ of $\Pi^n(\mathcal{D}, \sigma_0, \alpha)$ such that for every $0 \leq i \leq n$

$$\sigma_i = \{l : h(l, i) \in A\}.$$

Consider the briefcase example. To predict what will happen in the state where neither clasp is up if we toggle the first clasp and then toggle the second clasp, we construct $\Pi^2(\mathcal{D}_{bc}, \sigma_0, \langle toggle(1), toggle(2) \rangle)$. This is done by replacing `#const n = 1` by `#const n = 2` and combining $\Pi^2(\mathcal{D}_{bc})$ with the encoding of the initial state

```
-holds(up(1),0).
-holds(up(2),0).
```

and the encoding of the action sequence

```
occurs(toggle(1),0).
occurs(toggle(2),1).
```

The answer set of the resulting program will uniquely define the resulting state.

The following chapters show how we can use material introduced here to do more complex reasoning tasks including planning and diagnostics. It is worth noting that this approach to reasoning about actions and change could not have been possible without a good understanding of the representation of defaults in ASP.

## Summary

In this chapter we discussed an ASP-based methodology for representing knowledge about discrete dynamic systems. Mathematically, such a system is represented by a transition diagram whose states correspond to physically possible states of the domain and whose arcs are labeled by actions. A transition $\langle \sigma_0, a, \sigma_1 \rangle$ belongs to the diagram iff the execution of action $a$ in state $\sigma_0$ can move the system to state $\sigma_1$. Even for simple systems the corresponding diagrams may be huge and difficult to represent. In this chapter we showed how this problem can be solved using a simple action language $\mathcal{AL}$. We showed how transition diagrams can be described by collections of $\mathcal{AL}$ statements called system descriptions. These representations are concise (a comparatively small system description can define a very large diagram), have intuitive informal semantics, and a comparatively high degree of elaboration tolerance. The formal mathematical semantics of $\mathcal{AL}$ was given in terms of logic programs under answer set semantics. The ability of ASP to represent defaults and direct and indirect effects of actions helped solve the frame and ramification problems that, for a long time, prevented researchers from finding precise mathematical definitions of transitions of discrete dynamic systems. We also described some simple but useful mathematical properties of system descriptions of $\mathcal{AL}$ and illustrated how translations of its system descriptions can be used to solve an important computational task, called temporal projection, consisting of computing the states that the system can move into after the execution of a sequence of actions in a given initial state. In the following chapters we show how this connection between system descriptions and logic programs allows answer set programming to be used to solve a number of more complex computational problems. The problem of representing dynamic

systems is an active and important area of knowledge representation and artificial intelligence. There are by now a number of action languages that differ from each other by underlying fundamental assumptions, the type of dynamic systems they are meant to model, and their ability to combine knowledge into modules. For instance, whereas $\mathcal{AL}$ is based on the inertia axiom, another action language called $\mathcal{C}$ uses a different fundamental assumption called the Law of Universal Causation, which says that everything that is true in the world must have a cause. Language $H$ is an extension of $\mathcal{AL}$ that allows representation of so-called hybrid dynamic system (i.e., systems with both discrete and continuous change). Another extension of $\mathcal{AL}$, called $\mathcal{ALM}$, supplies $\mathcal{AL}$ with modular structure, which supports reusability and simplifies the development of knowledge representation libraries. This is, of course, a very incomplete list. Development of action languages and the study of relationships between them are important topics for future research.

## References and Further Reading

Information about Shakey the robot and STRIPS can be found in Nillson (1984) and Fikes and Nilsson (1971). There are several early influential approaches to formalization of reasoning about actions and change. The first such formalism, called situation calculus, was introduced by John McCarthy (1983) and later developed by a large number of researchers; see, for instance, Lin and Reiter (1994), Lin (1995), Pinto (1998), Pinto and Reiter (1995), and Gelfond (1991). The calculus reifies actions and fluents and expresses their properties in the language of classical logic. The underlying semantics of the formalism was given by circumscription. A mathematically precise and intuitively clear account of reasoning about dynamic domains based on a modern version of this formalism can be found in Ray Reiter's book (2001). Another early approach to reasoning about actions and change is based on event calculus. Its original version, formulated in the framework of logic programming, was first introduced by Robert Kowalski and Marek Sergot (1986). A later work (Shanahan 1995) reformulated the logic programming axioms in the logic of circumscription. For more information on event calculus see, for instance, Kowalski (1992), Kakas and Miller (1997), and Kakas, Miller, and Toni (2001). The formalism is covered in detail by Murray Shanahan (1997) and Erik Mueller (2006). Another interesting approach to reasoning about dynamic domains that is based on a form of preferential entailment and combines differential equations and logic is summarized in Sandewall (1994). The formalism gave rise to temporal action logics; see, for instance, Doherty et al. (1998).

Action languages have their roots in STRIPS that evolved into action description language (ADL; Pednault 1994) and planning domain definition language (PDDL; McDermott et al. 1998). The latter language and its variants are commonly used in the planning community. The language $\mathcal{AL}$ used in this book is an extension of action language $\mathcal{B}$ (Gelfond and Lifschitz 1998), which is essentially a subset of the action language from Turner (1997). The investigation of the relationship between these languages and logic programming started in Gelfond and Lifschitz (1993) and continued in a number of papers including Turner (1997), Baral and Lobo (1997), and Balduccini and Gelfond (2003). Action language $\mathcal{C}$, based on the Principle of Universal Causation (Leibniz, 1951), was introduced in Giunchiglia and Lifschitz (1998). Its popular generalization $\mathcal{C}^+$ can be found in Giunchiglia et al. (2004). In a number of respects $\mathcal{C}$-based languages are more general than those based on $\mathcal{B}$. In other respects the situation is reversed. For instance, in $\mathcal{C}$ and its extensions causal laws may contain arbitrary propositional formulas, be defeasible, and so on. In contrast, causal laws in these languages cannot be recursive, which is allowed in $\mathcal{B}$. For comparison of $\mathcal{B}$ and $\mathcal{C}$ one can see Gelfond and Lifschitz (2012). Languages $\mathcal{ALM}$ and $H$ were introduced in Gelfond and Inclezan (2009) and Chintabathina, Gelfond, and Watson (2005), respectively. For a different perspective, one can consult Thielscher (2008). The frame problem was first described by Patrick Hayes and John McCarthy (1969); our formulation of the ramification problem goes back to Ginsberg and Smith (1988). Proposition 8.4.1 and the related definitions are from Gelfond and Inclezan (2013).

## Exercises

1. Prove that
   (a) Every state of system description $\mathcal{SD}$ satisfies the state constraints of $\mathcal{SD}$.
   (b) If the signature of $\mathcal{SD}$ does not contain defined fluents, a state is a complete, consistent set of literals satisfying the state constraints of $\mathcal{SD}$.
2. Given the following $\mathcal{AL}$ system description where fluents $f$ and $g$ are inertial and $h$ is defined,

$$a \textbf{ causes } f \textbf{ if } g$$

$$h \textbf{ if } f, g$$

   (a) Show its translation into the corresponding ASP program.

(b) Check if each of the following is a valid state:

i. $\sigma = \{f, \neg g, \neg h\}$
ii. $\sigma = \{\neg f, \neg g, \neg h\}$
iii. $\sigma = \{\neg f, g, h\}$
iv. $\sigma = \{f, g, \neg h\}$

(c) Draw the transition diagram for the system.

3. Consider the system description presented in Section 8.6.
   (a) Explain why $\{f, p, q\}$ is not a valid state of this system description.
   (b) Add causal law $b$ **causes** $\neg f$ **if** $f$ to the system description and draw the corresponding transition diagram.

4. Given the following story: *Jenny painted the wall white.*
   (a) Represent the story in $\mathcal{AL}$. Assume that to paint a wall a given color, one must have paint of the appropriate color. Initially the wall is yellow and Jenny has the white paint. Jenny paints the wall at step 0. Make sure that your theory entails that at the end of the story the wall is white and not yellow.
   (b) Translate the representation to ASP and run it using an ASP solver to predict the values of fluents if Jenny paints the wall white.
   (c) Now suppose Jenny has black paint as well and, after painting the wall white, decides to paint it black. Use ASP to do some temporal projection about the values of the fluents after both actions are performed sequentially.
   (d) Now go back to the original story, but add another person, say Jill. Jenny and Jill have white paint. Jenny painted a wall white. Jill painted another wall white. Assume two people cannot paint the same wall at the same time, but they can paint different walls concurrently. Modify your $\mathcal{AL}$ representation to accommodate the new law, initial situation, and trajectory. Translate to ASP and run the program to compute the answer set after Jill and Jenny painted concurrently. Make sure that if you told the program that they painted the same wall concurrently, there would be no answer set.

5. Given the following story: *Claire always carries her cell phone with her. Claire is at the library.*
   (a) Represent the story in $\mathcal{AL}$. Include any commonsense knowledge necessary to answer the question, "Where is Claire's cell phone?" Make the representation general enough so that when you add the fact that Claire went home, her cell phone's location changed

accordingly. Also make it general enough that if we change the cell phone to a pet chihuahua, your program will still make the proper conclusions. Assume locations are distinct. *Hint:* Use the inertial fluent $carried(Obj, Person)$ – the object is carried by the person.

(b) Translate the representation to ASP and run it using an ASP solver to predict the values of fluents after Claire returns home.

(c) Modify your program to include that Rod carries a towel with him everywhere and that Claire and Rod are never at the same place at the same time. Make sure that your program can conclude that Claire's cell phone and Rod's towel are also not at the same place at the same time, even when it does not know where Rod is initially.

6. Given the following story: *Amy, Bruce, Carrie, and Don are vendors at a farmers' market. Amy sells apples and carrots, Bruce sells lettuce, Carrie sells apples, and Don sells cabbage and pears. For simplicity, assume that when a customer buys a type of produce from a vendor, he buys the whole quantity that the vendor has to offer.*

   (a) Suppose that *Sally goes to the farmers' market and buys all the lettuce that Bruce sells and all the apples that Carrie sells. Peter goes to the farmers' market afterward.* Represent both parts of the story in $\mathcal{AL}$. Make sure that the temporal projection on your representation entails that Peter cannot buy lettuce but can buy apples.

   (b) Translate the representation to ASP and run it using an ASP solver to answer the questions.

7. Given the following story: *Jonathan has requirements for playing the Wii: He should make sure that his homework is done, the bed is made, and he has practiced Tae Kwon Do. He can only do one thing at a time. Of course, he cannot make the bed if it already made or do his homework if it is already done or if none was assigned.*

   (a) Select an initial situation and a sequence of Jonathan's actions that would allow him to play the Wii. Represent the resulting story in $\mathcal{AL}$. *Hint:* Create a sort $activity$ for homework, make_bed, TKD, and Wii. Then use actions $do(hw)$, $do(make\_bed)$, $do(tkd)$, and $do(wii)$ to make it easy to express the requirement that actions be mutually exclusive.

   (b) Translate the representation to ASP and run it using an ASP solver to answer questions about whether a boy may play the Wii at various future moments. Make sure that the system description part of your program works for other variants of this story.