Table of Contents I

Modeling Dynamic Domains
- Blocks World: An Example of a Dynamic World
- A General Solution
  - $\mathcal{AL}$ Syntax
  - $\mathcal{AL}$ Semantics
- Examples
  - The Briefcase Domain
  - The Blocks World Revisited
- Concurrency
- Temporal Projection

Reading

- Read Chapter 8, *Modeling Dynamic Domains*, in the KRR book.

Things Change with Time

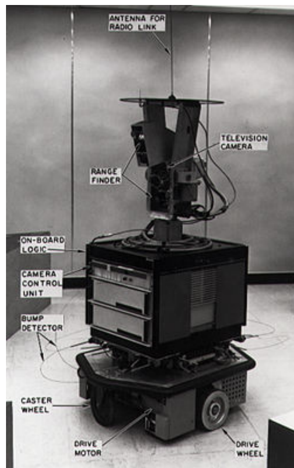
- We want our agents to hold their own in a changing environment.
- To do this, they need to be able to predict the effects of complex actions.
- And for that, they need to know about actions and their effects.

Overview

- Let's start by looking at a simple example of an agent trying to act in a changing environment.
- The plan is to flush out some of the issues that exist in designing such an agent.
- Next, we'll look at a formal theory of actions and change.
- The theory views the world as a dynamic system whose states are changed by actions.
- Language $\mathcal{AL}$ describes these systems.
- $\mathcal{AL}$ has a direct translation into ASP.

Early Research

This is Shakey the robot built by SRI International (currently the Stanford Research Institute). It builds with blocks.



Blocks World: An Example of a Dynamic World

- Building Shakey, and getting it to plan its actions based on a goal given it by humans brought to light many challenges involved in building and programming such a machine.

Blocks World: An Example of a Dynamic World

- Building Shakey, and getting it to plan its actions based on a goal given it by humans brought to light many challenges involved in building and programming such a machine.
- Hardware challenges:
  - creating a robotic arm
  - creating a robotic eye
  - (We now know that much of those challenges are software challenges too.)

Blocks World: An Example of a Dynamic World

- Building Shakey, and getting it to plan its actions based on a goal given it by humans brought to light many challenges involved in building and programming such a machine.
- Hardware challenges:
  - creating a robotic arm
  - creating a robotic eye
  - (We now know that much of those challenges are software challenges too.)
- Software challenges:
  - 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 Only Reasoned about Blocks

- Shakey used LISP and a theorem-proving planner called STRIPS.
- Its planning was domain-specific.
- This was the case for many planners that followed.
- We've learned a lot since then (thanks, in part, to Shakey).
- Recently, I saw the Atlas robot stacking boxes. https://www.youtube.com/watch?v=rVlhMGQgDkY

The Action Language Approach

- The theory of actions is developed independent of the domain the agent will function in.
- Teach the reasoner about the laws of the world is separate from teaching it the various tasks it will perform in the world.
- This does not mean that we can't use domain-specific knowledge to help it plan, etc.

Blocks World in ASP

Let's create a domain-specific blocks world representation in ASP with an eye on a general solution.

While we do this, let's

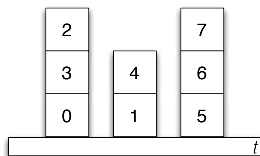
- consider the types of knowledge we need to represent,
- develop a vocabulary for talking about dynamic domains, and
- get motivated for the rigorous formalism.

Defining a Version of the Problem

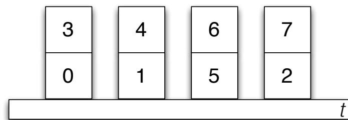
- We have a robotic arm that can manipulate configurations of same-sized cubic blocks on a table.
- It can move *unoccupied* blocks, one at a time, onto other unoccupied blocks or onto the table.
- At any given step, a block can be in at most one location.
- Towers can be as big as we want.
- The table is large enough to fit all blocks, even if they are not stacked.
- We do not take into account spacial relationships of towers, just which *blocks* are on top of each other and which blocks are on the table.

Agent Specifications

Example: Given info that describes this:



and actions that occur (put 2 on the table, put 7 on 2), know that this is what the configuration looks like now:



Objects and Relations

- What are the objects that the agent will be reasoning about?
- What are the relations between them?

Changing Configurations

We can think of the initial configuration as a collection of terms that describe positions of blocks:

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

Then, after action $put(b2, t)$ is performed, the configuration changes to

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

The Complete Trajectory

Once action $put(b7, b2)$ is performed, we get the next configuration:

$$\sigma_2 = \{on(b0, t), on(b3, b0), on(b1, t), on(b4, b1), on(b2, t),$$

$$on(b5, t), on(b6, b5), on(b7, b2)\}.$$

The execution of a sequence of actions of type $put(B, L)$ in configuration σ_0 determines the system's trajectory

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

which describes its behavior.

A Generic Trajectory

In general, 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.
(picture)

Describing our System's Trajectory — Vocabulary

- Use integers from 0 to a finite n to denote the *steps* of the corresponding trajectories.
- Distinguish between
  - **fluents** — properties that can be changed by actions
  - **statics** — properties that cannot

  In our example, we chose to view the property of a block being on top of a location as a fluent and the state of something being a block as a static. (In SPARC, statics are often sorts.)

Describing our System's Trajectory — Relations

- `holds(#fluent,#step)` describes what fluents are true at a given step.
- `occurs(#action,#step)` describes what action occurred at a given step.
- In our example, we define
  - $holds(on(B, L), I)$
    "block B is on location L at step I;"
  - $occurs(put(B, L), I)$,
    "block B was put on location L at step I."
- Note that we have reified actions and fluents.

Declarations

```
#const n = 2.

sorts

#block = [b][0..7].
#location = #block + {t}.
#fluent = on(#block(X),#location(Y)):X!=Y.
#action = put(#block(X),#location(Y)):X!=Y.
#step = 0..n.

predicates

holds(#fluent,#step).
occurs(#action,#step).
```

Initial Configuration

```
%% holds(on(B,L),I): a block B is on location L at step I.
%% This is a particular intial configuration.
%% It can be changed at will:
holds(on(b0,t),0).
holds(on(b3,b0),0).
holds(on(b2,b3),0).
holds(on(b1,t),0).
holds(on(b4,b1),0).
holds(on(b5,t),0).
holds(on(b6,b5),0).
holds(on(b7,b6),0).

%% If block B is not known to be on location L at step 0,
%% then we assume it is not.
-holds(on(B,L),0) :- not holds(on(B,L),0).
```

Defining BW Laws: Direct Effect of Action $put(B, L)$

Effect of action $put(B, L)$:

```
% Putting block B on location L at step I
% causes B to be on L at setp I + 1.

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

Defining BW Laws: Indirect Effect of Action *put*(B, L)

Indirect effects can often be expressed as relationships between fluents.

```
%% A block cannot be in two locations at once:

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

%% Only one block can be set directly on top of another:

-holds(on(B2,B),I) :- #block(B), % not the table
                      holds(on(B1,B),I),
                      B1 != B2.
```

Testing Laws

Given the declarations, initial configuration, and three laws plus
`occurs(put(b2,t),0).`
what does our program know?

- Query `holds(on(b2,t),1)` returns *yes*. So far so good.
- Query `hold(on(b0,t),1)` returns *unknown*, even though we expect it to still be true.
- Why do we expect it to be true?

The Inertia Axiom

Normally things stay as they are.

```
% Inertia:
holds(F,I+1) :- holds(F,I),
                not -holds(F,I+1).

-holds(F,I+1) :- -holds(F,I),
                 not holds(F,I+1).
```

This is a typical representation of defaults. Note that our states are complete and the rule has no weak exceptions, so we don't need *ab*.

Are We Done?

- Let's tell the agent that something impossible happened.

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

- Since b6 was occupied, that action was not allowed.
- Our program thinks that b6 is now on the table and that b7 is still on top.
- We didn't assume that our robot was that good!
- Let's teach the agent which actions are not allowed.

Defining BW Laws: Impossible Actions

```
%% It is impossible to move an occupied block:
-occurs(put(B,L),I) :- holds(on(B1,B),I).

%% It is impossible to move a block onto
%% an occupied block:
-occurs(put(B1,B),I) :- #block(B),
                        holds(on(B2,B),I).
```

Now our program computes what we want. It knows

- The direct effects of actions $put(B, L)$.
- Indirect effects of actions $put(B, L)$.
- The Inertia Axiom: normally things stay as they are.
- Which actions should not be allowed.

Let's add a new fluent that is described in terms of the fluents we have already.

Defining *above*(B, L)

We add the new fluent to our fluent sort:

```
#fluent = on(#block(X),#location(Y)):X!=Y +
          above(#block(X),#location(Y)):X!=Y.
```

and add the following rules to define it:

```
% Block B is located above location L:

holds(above(B,L),I) :- holds(on(B,L),I).
holds(above(B,L),I) :- holds(on(B,B1),I),
                       holds(above(B1,L),I).
%% CWA for above:
-holds(above(B,L),I) :- not holds(above(B,L),I).
```

Not a No-brainer

If we add these rules to our program and also add

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

Since we put b2 on the table, our program should have no answer set. Instead, it has two!

In one, $holds(above(b2, b0), 1)$ is true because of Inertia, and in the other $\neg holds(above(b2, b0), 1)$ holds because of CWA!

Two Types of Fluents

Note that $above(B, L)$ is completely defined in terms of $on(B, L)$ and so should not be made subject to Inertia. As *on* changes, so does *above*.

We solve our problem by splitting fluent into two types: inertial and defined and having Inertia apply only to inertial fluents.

Updated Declarations

```
#inertial_fluent = on(#block(X),#location(Y)):X!=Y.

#defined_fluent = above(#block(X),#location(Y)):X!=Y.

#fluent = #inertial_fluent + #defined_fluent.
```

Updated Inertia

```
% Inertia:
holds(F,I+1) :- #inertial_fluent(F),
                holds(F,I),
                not -holds(F,I+1).

-holds(F,I+1) :- #inertial_fluent(F),
                 -holds(F,I),
                 not holds(F,I+1).
```

A General Solution

- Let's step back and try to see the laws of the blocks world in more general terms.
- One possible way to model a dynamic system is by a **transition diagram** — a directed graph whose
  - nodes correspond to physically possible states of the domain
  - arcs are labeled by actions.
- Such models are called Markovian.

Transitions

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 σ_0, indicates that σ_1 may be a result of simultaneous execution of these actions in σ_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.

Transition Diagrams

- 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 σ_0 and final state σ_n.
- The transition diagram for a system contains all possible trajectories of that system.
- These diagrams get complicated quickly.
- Finding a concise and mathematically accurate description of the diagrams has been a subject of research for over 30 years.
- Things get even more complicated because of the need to specify *what is not changed by actions.*

The Frame Problem

As described by John McCarthy, the **Frame Problem** is the problem of finding a concise and accurate representation in a formal language of what is not changed by actions.

Notice that in our blocks world problem, we showed a solution to the Frame Problem. We simply reduced it to finding a representation of the Inertia Axiom — a default that states that things normally stay as they are.

Causal Laws

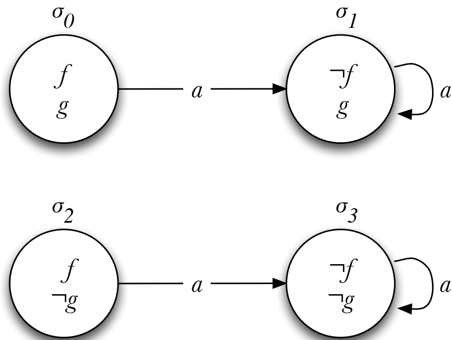
Causal effects of actions can be defined by causal laws of the form:

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

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
$put(Block, Location)$.

Example: Transition Diagram for a Causal Law



$$\mathcal{F} = \{f, g\} \quad \mathcal{A} = \{a\}$$

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

State Constraints

Indirect effects of actions can be defined by state constraints of the form:

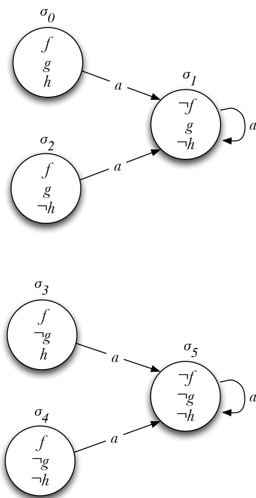
$$f \textbf{ if } p_0, \ldots, p_m$$

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

Finding concise ways for defining these effects is called the **Ramification Problem**.

Together with the Frame Problem discussed above, the Ramification Problem caused substantial difficulties for researchers in their attempts to precisely define transitions of discrete dynamic systems.

Example: Transition Diagram Including a State Constraint



$\mathcal{F} = \{f, g, h\}$ $\mathcal{A} = \{a\}$

a **causes** $\neg f$ **if** f

$\neg h$ **if** $\neg f$

Executability Conditions

Executability conditions are represented by laws of the form

$$\textbf{impossible } a_1 \dots a_k \textbf{ if } p_0, \dots, p_m$$

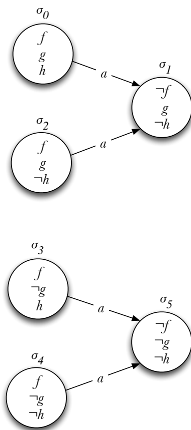
which say that *it is impossible to execute actions* $a_1 \dots a_k$ *simultaneously in a state satisfying conditions* $p_0, \dots, p_m$.

As an example, let's add rule

$$\textbf{impossible } a \textbf{ if } \neg f$$

which says that it is impossible to perform action a in any state which contains fluent $\neg f$.

Example: Transition Diagram Including an Executability Condition



Action Languages

Action languages are formal models of parts of natural language used for describing the behavior of dynamic systems.

Looking at it another way, they are tools for describing transition diagrams.

$\mathcal{AL}$ Syntax – definitions

- 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*.
- **domain properties** — statics and fluents
- **domain literal** — domain property or its negation. (We can have **fluent literals** and **static literals**.)

$\mathcal{AL}$ Syntax – Complete and Consistent

- 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 $\neg p \in S$.

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 $m \geq -1$. Moreover, no negation of a defined fluent can occur in the heads of state constraints.

System Description

A **system description** of $\mathcal{AL}$ is a collection of statements of $\mathcal{AL}$.

$\mathcal{AL}$ Semantics

- A system description serves as a specification of a transition diagram of a dynamic system.
- It describes all possible trajectories of the system.
- So, to define the meaning of $\mathcal{AL}$ statements, we need to define the states and legal transitions of the diagram.

Defining States

- If we didn't have to worry about defined fluents, we could define states as

  a complete and consistent set of domain literals satisfying the system's state constraints

- Unfortunately, this is not good enough to give defined fluents their intended meaning.

Example: The Problem with Defined Fluents

Suppose we have two state constraints:

$$h \textbf{ if } f$$

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

where h is a defined fluent and f and g are inertial.

Let's consider the possible sets as candidate states.
(See board.)
Notice that set $\{\neg f, g, h\}$ doesn't seem to satisfy the definition of h since the truth of h doesn't follow from any of its defining rules. However, it is complete and consistent.

The Solution

- To capture the intended meaning of defined fluents, we turn to ASP.
- To see whether a candidate set is a state, we create an ASP program by:
  1. Taking all the state constraints and replacing the **if** in them by a $\leftarrow$. (And adding a period.)
  2. Adding the CWA for each *defined* fluent.
  3. Adding all the statics and inertial fluents from the candidate set to the program as facts.
- A complete and consistent set of domain literals is a **state** of the transition diagram defined by a system description if it is the unique answer set of the program thus created.

(back to example)

Transitions

- The definition of the transition relation of a diagram 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
  1. logic programming encodings of system description $\mathcal{SD}$,
  2. initial state σ_0, and
  3. set of actions a,

  such that answer sets of this program determine the states the system can move into after the execution of a in σ_0.

Definition of the Encoding of a System Description

- To define the encoding of a given system description, we define the encoding of its signature and its statements.
- Let's look at this step by step.
  (Definition 8.4.5 — in detail on board)

Encoding the Initial State and the Occurrence of Actions

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 σ_0 and the encoding $occurs(a, 0)$ of action(s) a:

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

and

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

(example on board)

Definition of Transition

Let a be a nonempty collection of actions and σ_0 and σ_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 can predict what the state of the world will be once an action is performed in the initial state.

Applying the Theory

To model a dynamic domain, we need to describe what actions cause what effects under what conditions. Thus, we need to identify:

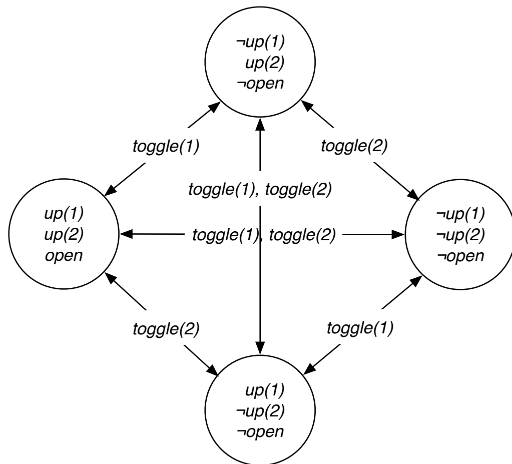
1. the objects, properties, and actions of the domain;
2. the relationships between the properties;
3. the executability conditions and causal effects of actions.

In other words, we must come up with an $\mathcal{AL}$ system description for our domain.

The Briefcase Domain: Story

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.

Briefcase Domain: Diagram



Briefcase Domain: Signature

- sort *clasp* $= \{1, 2\}$
- inertial fluent $up(C)$ which holds iff clasp C is up
- defined fluent *open* which holds iff both clasps are up
- action $toggle(C)$ which toggles clasp C.

Briefcase Domain: System Description

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

$$\begin{array}{l} toggle(C) \;\; \textbf{causes} \;\; up(C) \;\; \textbf{if} \;\; \neg up(C) \\ toggle(C) \;\; \textbf{causes} \;\; \neg up(C) \;\; \textbf{if} \;\; up(C) \\ open \;\; \textbf{if} \;\; up(1), up(2) \end{array}$$

  where C ranges over the sort *clasp*.
- First two contain variables, so are not actual laws, but schemas.
- Individual laws can be obtained from schemas by grounding the variables, so we actually have five laws.

Briefcase: States

- To check which combinations of literals constitute states, we look at the possible complete and consistent sets that comply with our requirement that they be unique answer sets of a program constructed from rules for defined fluents and facts for inertial fluents and statics.
- Our only defined fluent is *open* so the rules are:

$$open \leftarrow up(1), up(2).$$

$$\neg open \leftarrow \ not\ open.$$

- Is $\{\neg up(1), up(2), \neg open\}$ a state?
- Is $\{\neg up(1), up(2), open\}$?

Briefcase: Transitions

- To define transitions, we construct a program using the definition of the encoding of $\Pi(\mathcal{SD})$.
- We'll call our system description $\mathcal{D}_{bc}$, so our program will be called $\Pi(\mathcal{D}_{bc})$.
- Since we'll be using SPARC, our domain signature will look a little different, but it is actually more conducive to representing sorts so, hopefully, it will be straightforward.

(ASP printout for direct translation.)

Briefcase: Declarations

```
#const n = 1.

sorts

#clasp = {1,2}.
#inertial_fluent = up(#clasp).
#defined_fluent = {open}.
#fluent = #inertial_fluent + #defined_fluent.
#action = toggle(#clasp).
#step = 0..n.

predicates

holds(#fluent,#step).
occurs(#action,#step).
```

Briefcase: Causal Laws

```
rules

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

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

Briefcase: State Constraint

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

There are no executability conditions, so we're done with the part of the encoding which is unique to this system description.

Briefcase: Rules for All Domains

```
%% CWA for Defined Fluents
-holds(F,I) :- #defined_fluent(F),
               not holds(F,I).

%% General Inertia Axiom
holds(F,I+1) :- #inertial_fluent(F),
                holds(F,I),
                not -holds(F,I+1).

-holds(F,I+1) :- #inertial_fluent(F),
                 -holds(F,I),
                 not holds(F,I+1).

%% CWA for Actions
-occurs(A,I) :- not occurs(A,I).
```

Comparing Program and Diagram

- Now that we've seen our program, let's compare it to the transition diagram for the briefcase domain.
- To see if

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

  is a valid transition, we add the description of the initial state and the occurs statement for the action to our program, and see whether it produces the correct answer set.
- Testing two parallel actions simply requires two occurs statements.
- Note that we do not need to list defined fluents to describe our initial state because their values will be computed automatically.

BW: Signature

The signature of $\mathcal{D}_{bw}$ consists of:

- sort $block = \{b_0 \dots b_7\}$,
- sort $location = \{t\} \cup block$,
- 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 $B \neq L$.

BW: System Description

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) Bs and Ls stand for blocks and locations, respectively.

Let's translate. (Class exercise.)

Concurrent Actions

- The idea of concurrency is built into the language, so no extra effort is needed on that front.
- However, when working with concurrent actions, we need to be careful to consider side-effects and additional restrictions.

Two-Arm Domain

- Suppose we have two robotic arms capable of avoiding collisions.
- This means that two blocks can be moved simultaneously.
- What would happen if both arms moved the same block to two separate locations? e.g. $occurs(put(b2, t), 0)$ and $occurs(put(b2, b4), 0)$
- What about $occurs(put(b2, t), 0)$ and $occurs(put(b7, b2), 0)$?

Two-Arm Domain — Preventing Simultaneous Actions

Let's teach the system that actions which 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 SPARC we get

```
-occurs(put(B1,L),I) | -occurs(put(B2,B1),I).
```

Temporal Projection

- **Temporal Projection** is the task of predicting the values of fluents after the execution of a sequence of actions.
- Because we have an ASP program that can compute the resulting states for us, we can already do temporal projection.
- We simply add the sequence of actions that we want to know the results of to the program, and let it compute the answer set.

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

- Fluents (and statics) that hold at step n describe the resulting state.