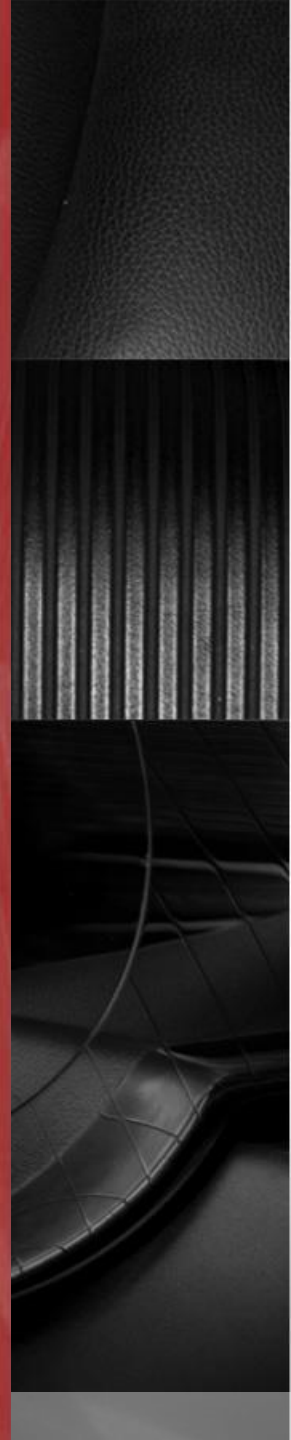
Lecture 5

Planning and Scheduling



Lecture Plan

1. Lab Assignment Discussion
2. Planning:
    I. PDDL
    II. Solving PDDL problems: A* & Planning Graphs
3. (Scheduling)
    I. Dynamic Programming for Scheduling
    II. Resource Constraints and the Minimum Slack Algorithm

PDDL: Planning Domain Definition Language

- States
- Actions
- Goal

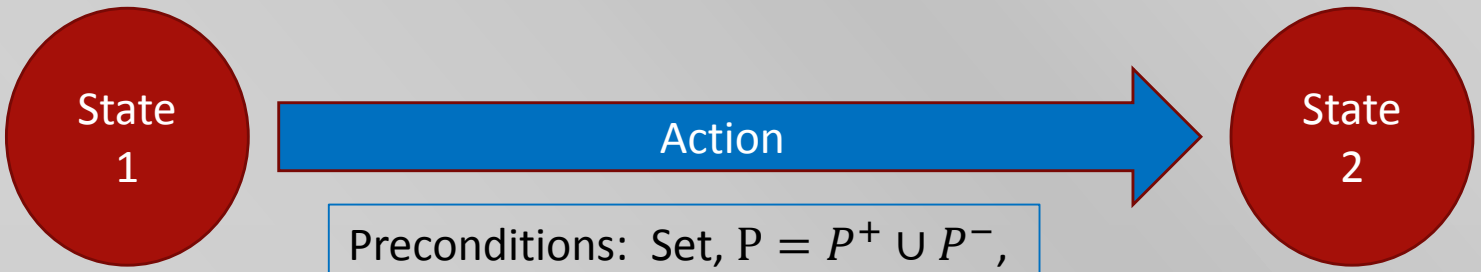
PDDL: Planning Domain Definition Language

- States
  - States are specified by a set of atomic ‘fluents‘ (statements that can be true or false).
  - Atomic means they do not include conjunctions, disjunctions, conditionals or negation.
  - Database semantics is assumed, allowing negation by omission.
    - Unique names assumption: Different names refer to different objects
    - Closed-world assumption: If not true, then false.
    - Domain closure: No unnamed objects.
- Actions
- Goal

PDDL: Planning Domain Definition Language

- States
- Actions
  - Specified by a set of preconditions and effects, both of which are conjunctions of ‘literals’ - atomic propositions or negations of atomic propositions. All preconditions are current, all effects immediate.
  - Actions can be performed at a state where all positive literals are, and all negative literals are not, in the set of fluents at a state.
  - The effect of an action is to add all positive literals, and remove all negative literals, in the action's effects from the current state.
- Goal

How actions function as transitions:



State 1:
Set, F, of Atomic Fluents

Preconditions: Set, $\mathrm{P} = P^{+} \cup P^{-}$, of literals, where:

- $P^{+} \subseteq \mathrm{F}$
- $A(P^{-}) \cap \mathrm{F} = \emptyset$.

Effects: Set, $\mathrm{E} = E^{+} \cup E^{-}$, of literals.

State 2:
Set, $\mathrm{G} = (F \cup E^{+}) \setminus A(E^{-})$, of Atomic Fluents

PDDL: Planning Domain Definition Language

- States
- Actions
- Goal
  - Specified by a conjunction of literals.
  - We have achieved a goal state when all positive literals are, and all negative literals are not, included in the current state specification.

A PDDL Problem

1. An initial state
2. A set of actions:

   Action schema used to represent classes of actions using variables:

   Action: Eat Cake (x,y)

   Precondition: Cake(x), -Eaten(x), Person(y), Hungry(y)

   Effect: Eaten(x), -Hungry(y)
3. A goal specification.

PDDL Problems as Search

1. We must discover the transitions we can make by seeing which actions can be performed.
2. It is easy to search backwards and discover states that can transition to our current state given a possible action.

   To find predecessor state to current state given a particular action: Remove all effects (add negations) and add all preconditions (remove negations) from current state.

PDDL Example: Initial State

Initial state:

Person (mike)

PDDL Example: Actions

Action:	Go Shopping (x)
Preconditions:	Person(x)
Effects:	HasMix(x)

PDDL Example: Actions

Action:	Make Cake(x)
Preconditions:	Person(x)
	HasMix (x)
Effects:	HasCake(x)
	-HasMix(x)

PDDL Example: Actions

Action:	Wait(x)
Preconditions:	Person(x),
	-Hungry(x)
Effects:	Hungry(x)

PDDL Example: Actions

Action:	Eat Cake A (x)
Preconditions:	Person(x),
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	-Hungry(x),
	EatenCake(x)

PDDL Example: Actions

Action:	Eat Cake B (x)
Preconditions:	Person(x),
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	-Person(x)

PDDL Example: Goal

Preconditions: Person(mike),

-Hungry(mike),

HasCake(mike),

EatenCake(mike)

We could have a goal specification that included variables. Eg Let anyone who is a person, not just mike, eat a cake while having a cake and not being hungry:

Person(x),

-Hungry(x),

HasCake(x),

EatenCake(x)

Action:	**Eat Cake B (x)**
Pre.:	Person(x) -Hungry(x), HasCake(x)
Effects:	-HasCake(x) -Person(x)

Goal	
Pre.:	Person(mike), -Hungry(mike), HasCake(mike), EatenCake(mike)

Action:	**Eat Cake A (x)**
Pre.:	Person(x) Hungry(x), HasCake(x)
Effects:	-HasCake(x), EatenCake(x), -Hungry(x)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Make Cake(x)**
Pre.:	Person(x) HasMix(x)
Effects:	HasCake(x) -HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x) -Hungry(x)
Effects:	Hungry(x)

Person(mike)

Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

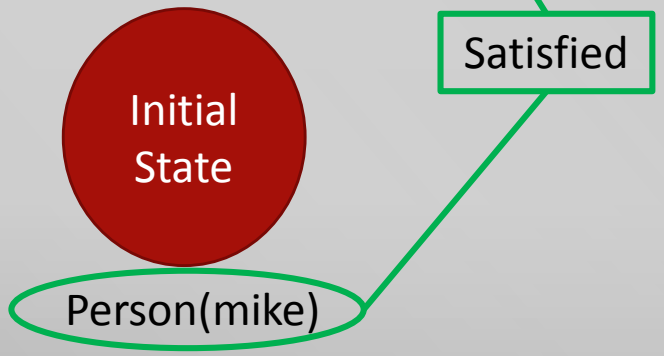
Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)

Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)



Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

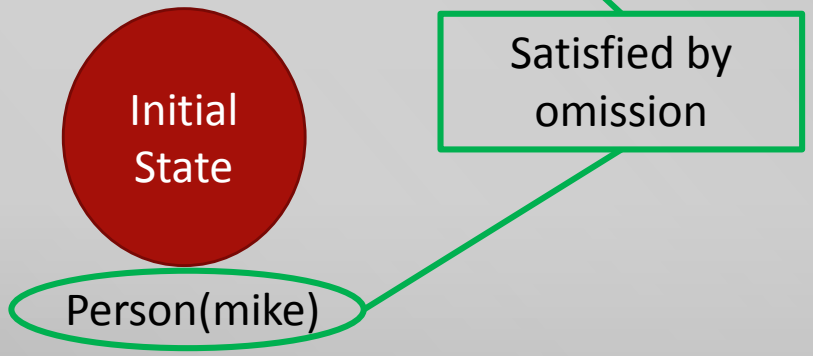
Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)

Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)



Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

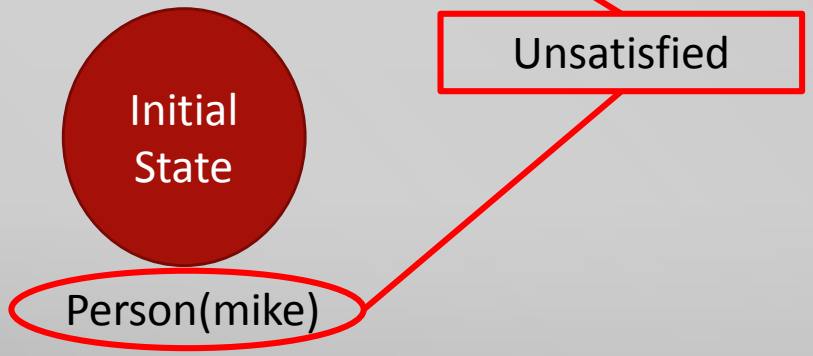
Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

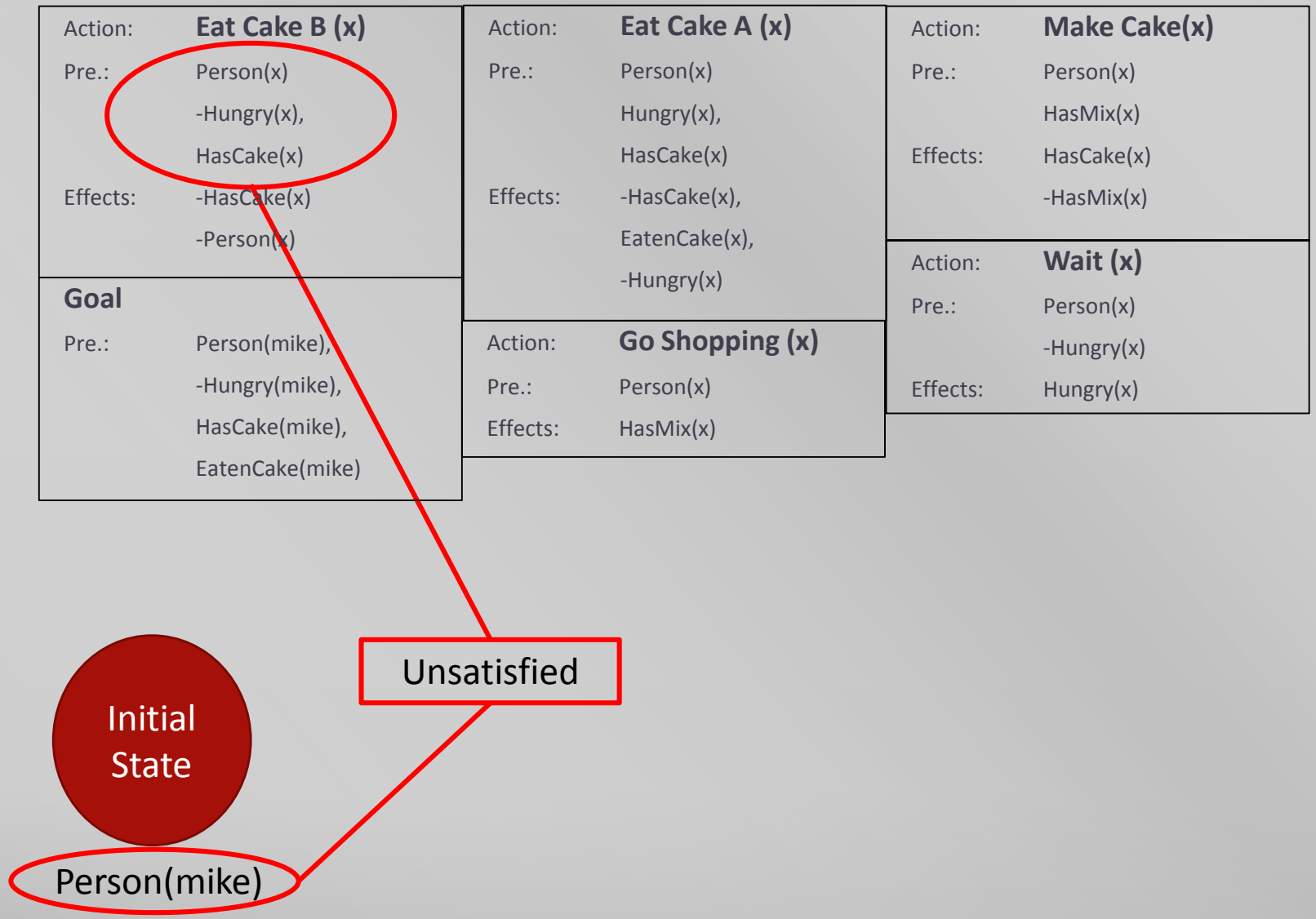
Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

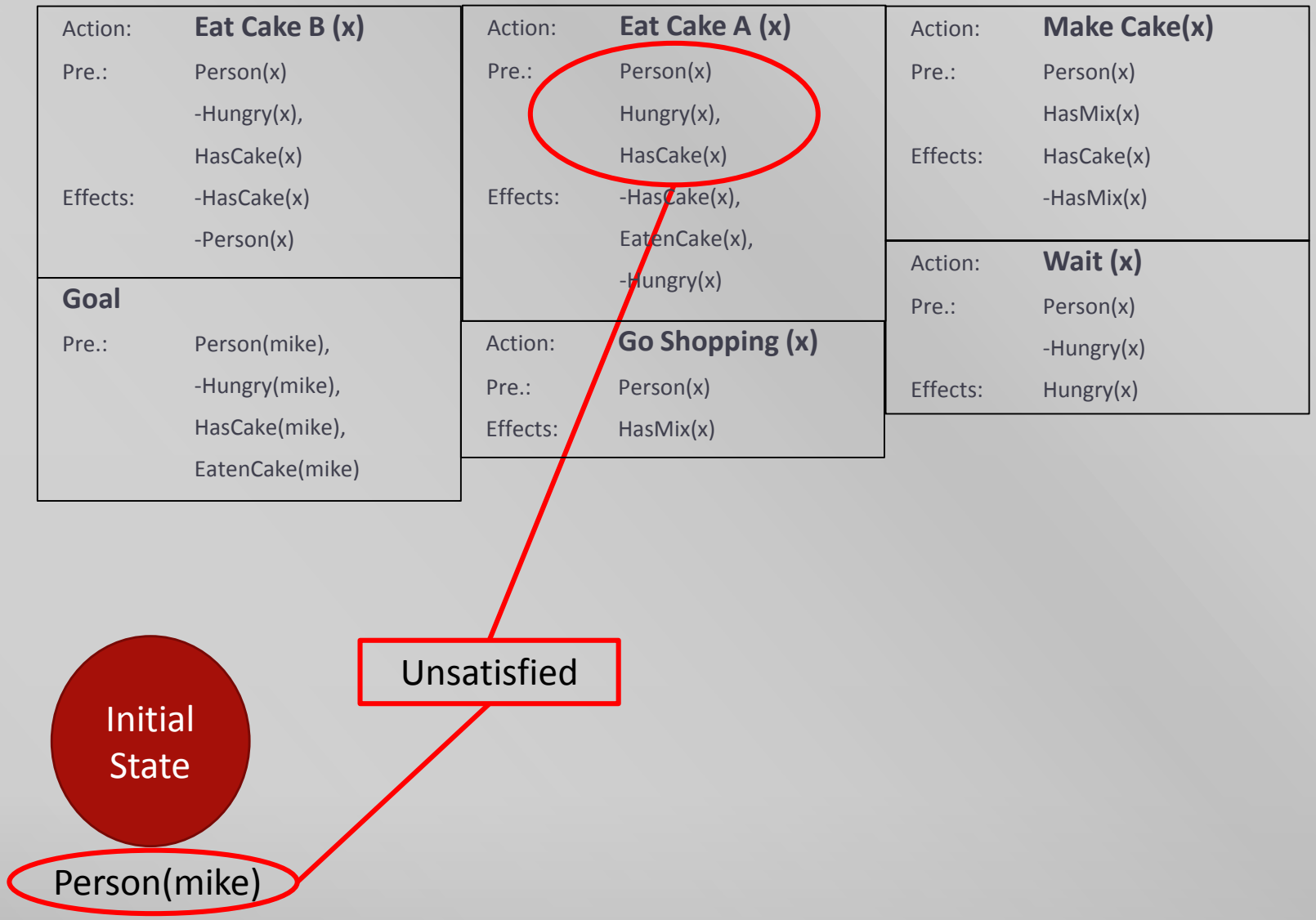
Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)




Action: Eat Cake B (x)
Pre.: Person(x)
-Hungry(x),
HasCake(x)
Effects: -HasCake(x)
-Person(x)
Goal
Pre.: Person(mike),
-Hungry(mike),
HasCake(mike),
EatenCake(mike)
Action: Eat Cake A (x)
Pre.: Person(x)
Hungry(x),
HasCake(x)
Effects: -HasCake(x),
EatenCake(x),
-Hungry(x)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
-HasMix(x)
Action: Wait (x)
Pre.: Person(x)
-Hungry(x)
Effects: Hungry(x)
Unsatisfied
Initial State
Person(mike)


Action: Eat Cake B (x)
Pre.: Person(x)
-Hungry(x),
HasCake(x)
Effects: -HasCake(x)
-Person(x)
Goal
Pre.: Person(mike),
-Hungry(mike),
HasCake(mike),
EatenCake(mike)
Action: Eat Cake A (x)
Pre.: Person(x)
Hungry(x),
HasCake(x)
Effects: -HasCake(x),
EatenCake(x),
-Hungry(x)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
-HasMix(x)
Action: Wait (x)
Pre.: Person(x)
-Hungry(x)
Effects: Hungry(x)
Unsatisfied
Initial State
Person(mike)

Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)

Initial State

Person(mike)

Unsatisfied

Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

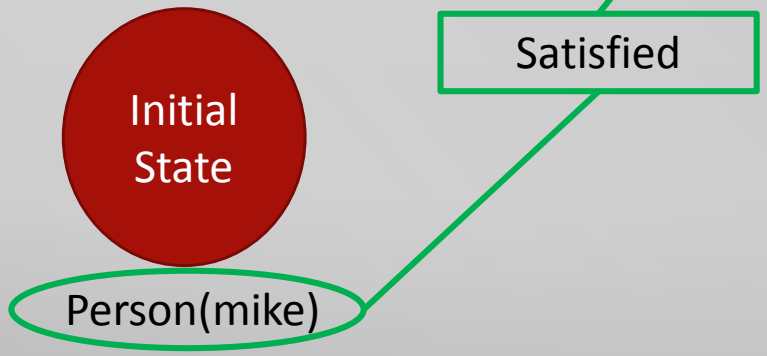
Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)



Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)

Initial State

Person(mike)

Satisfied (the second by omission).

Action:	**Eat Cake B (x)**
Pre.:	Person(x)
	-Hungry(x),
	HasCake(x)
Effects:	-HasCake(x)
	-Person(x)

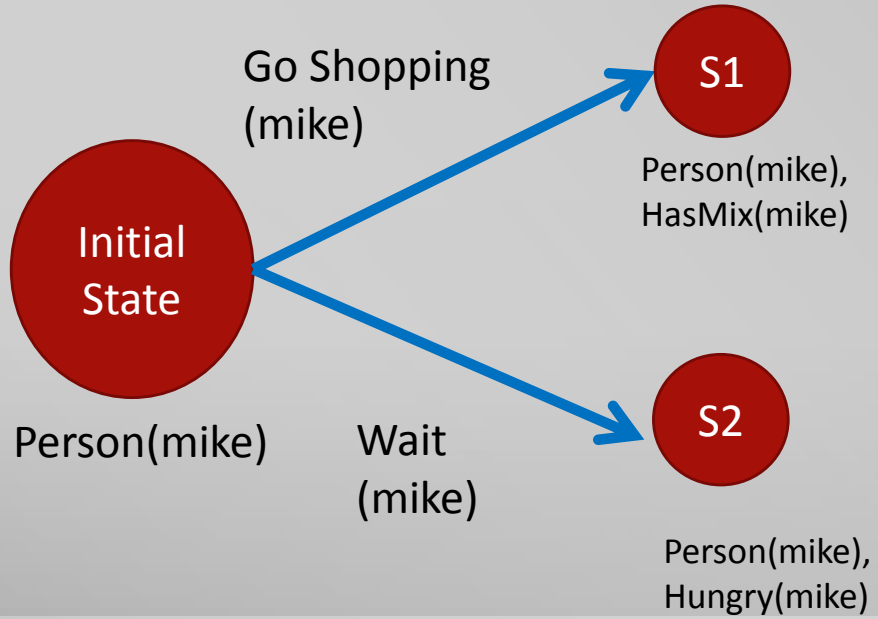
Goal	
Pre.:	Person(mike),
	-Hungry(mike),
	HasCake(mike),
	EatenCake(mike)

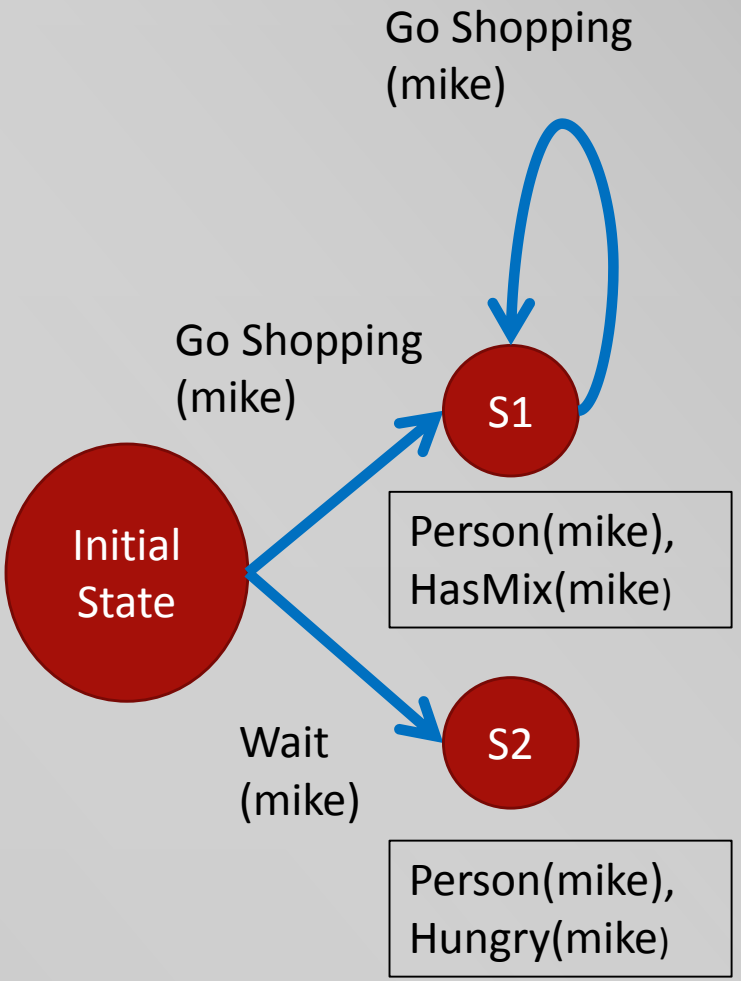
Action:	**Eat Cake A (x)**
Pre.:	Person(x)
	Hungry(x),
	HasCake(x)
Effects:	-HasCake(x),
	EatenCake(x),
	-Hungry(x)

Action:	**Go Shopping (x)**
Pre.:	Person(x)
Effects:	HasMix(x)

Action:	**Make Cake(x)**
Pre.:	Person(x)
	HasMix(x)
Effects:	HasCake(x)
	-HasMix(x)

Action:	**Wait (x)**
Pre.:	Person(x)
	-Hungry(x)
Effects:	Hungry(x)



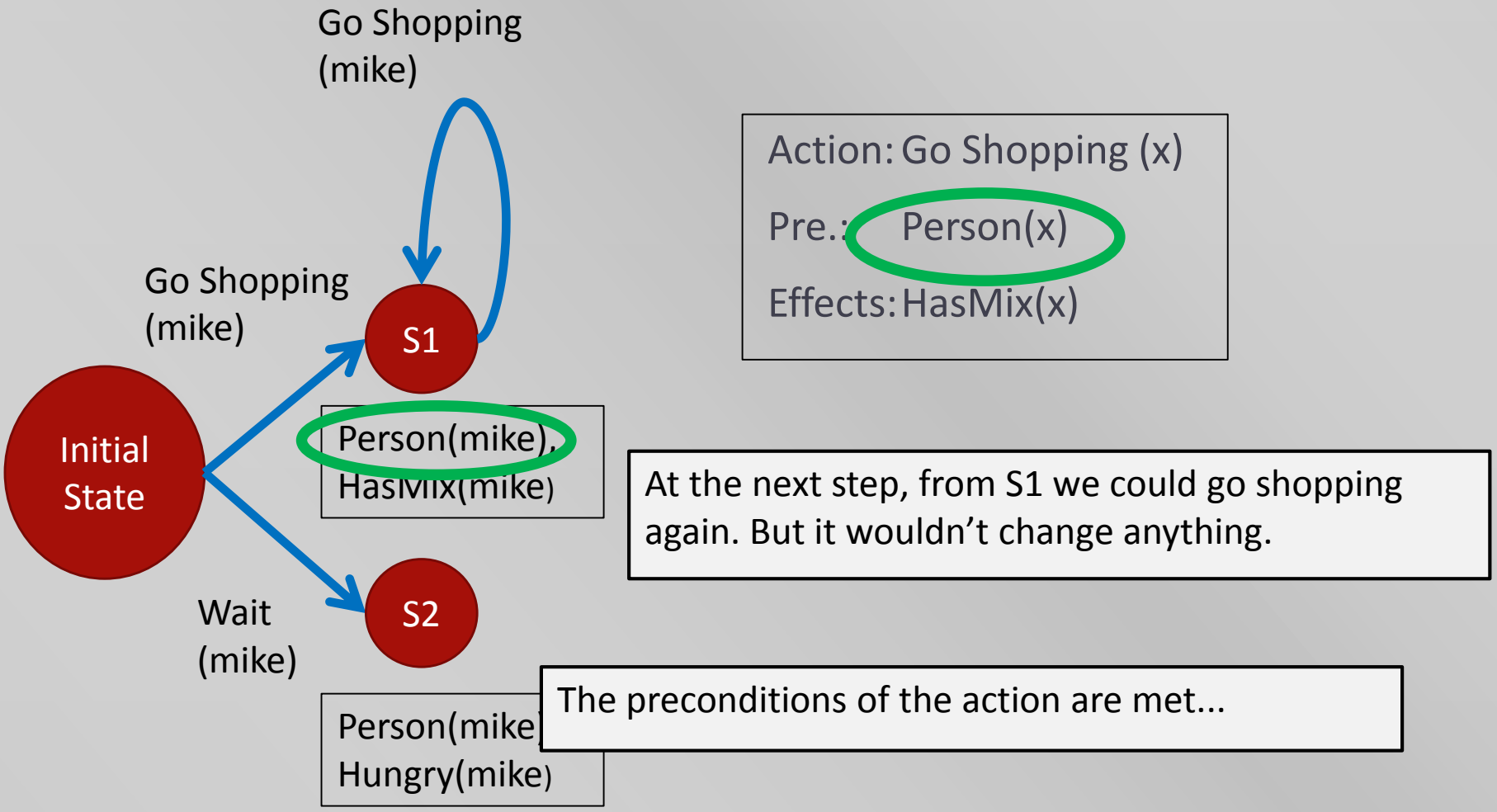


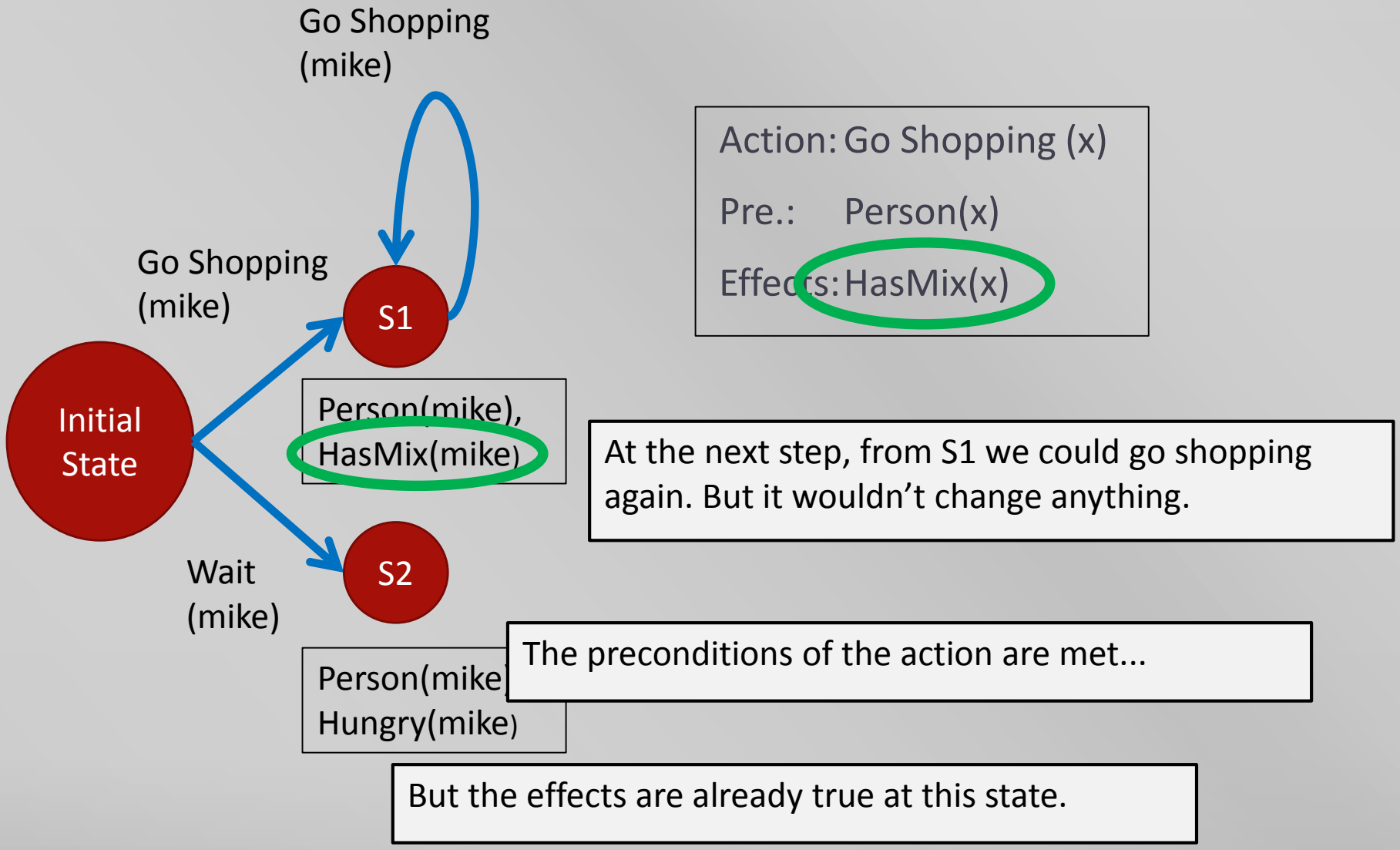
Action: Go Shopping (x)

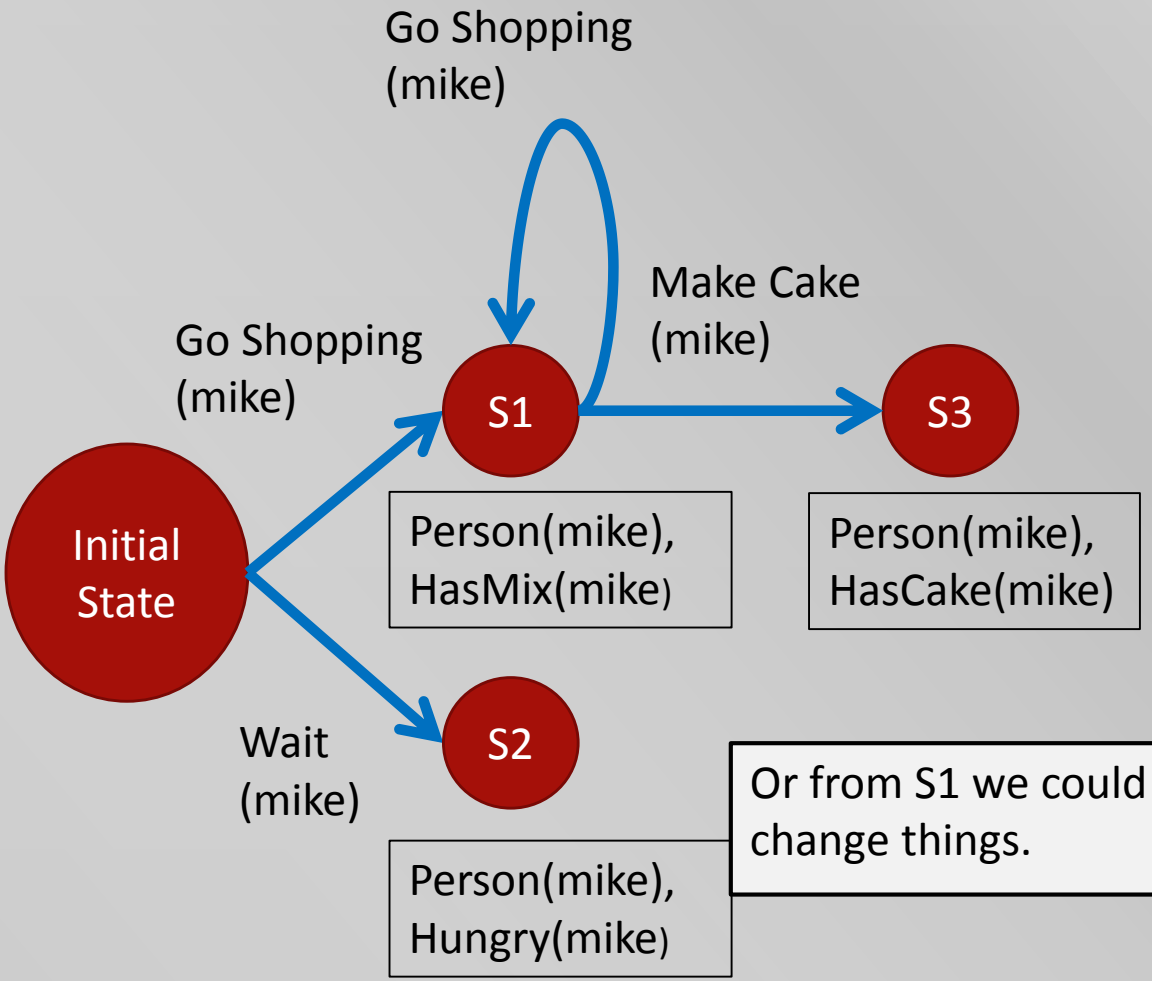
Pre.: Person(x)

Effects: HasMix(x)

At the next step, from S1 we could go shopping again. But it wouldn't change anything.


Go Shopping
(mike)
Go Shopping
(mike)
S1
Initial
State
Person(mike),
HasMix(mike)
Wait
(mike)
S2
Person(mike
Hungry(mike)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
At the next step, from S1 we could go shopping again. But it wouldn't change anything.
The preconditions of the action are met...


Go Shopping
(mike)
Go Shopping
(mike)
S1
Initial
State
Person(mike),
HasMix(mike)
Wait
(mike)
S2
Person(mike
Hungry(mike)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
At the next step, from S1 we could go shopping again. But it wouldn't change anything.
The preconditions of the action are met...
But the effects are already true at this state.



Action: Make Cake(x)

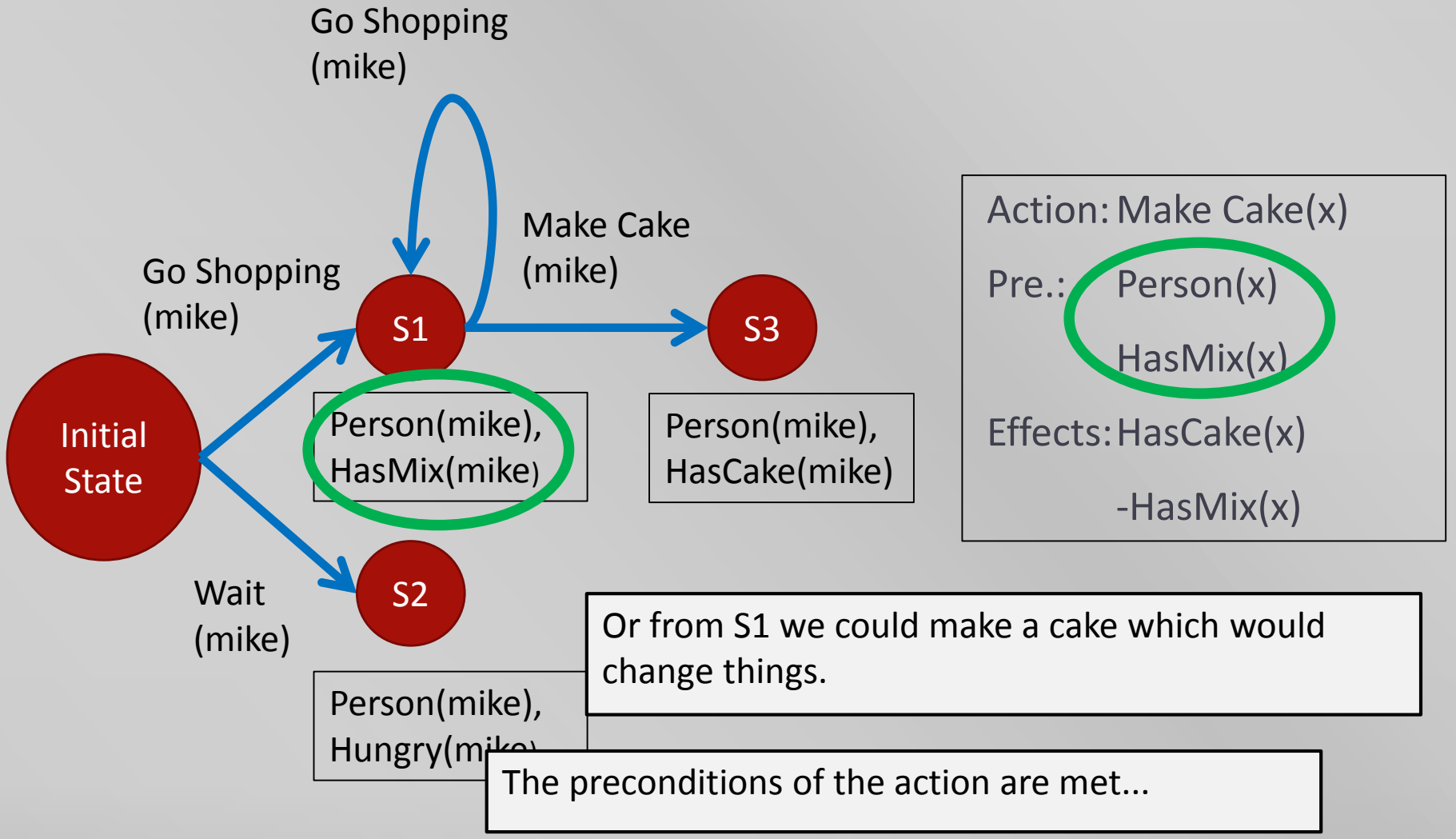
Pre.: Person(x)

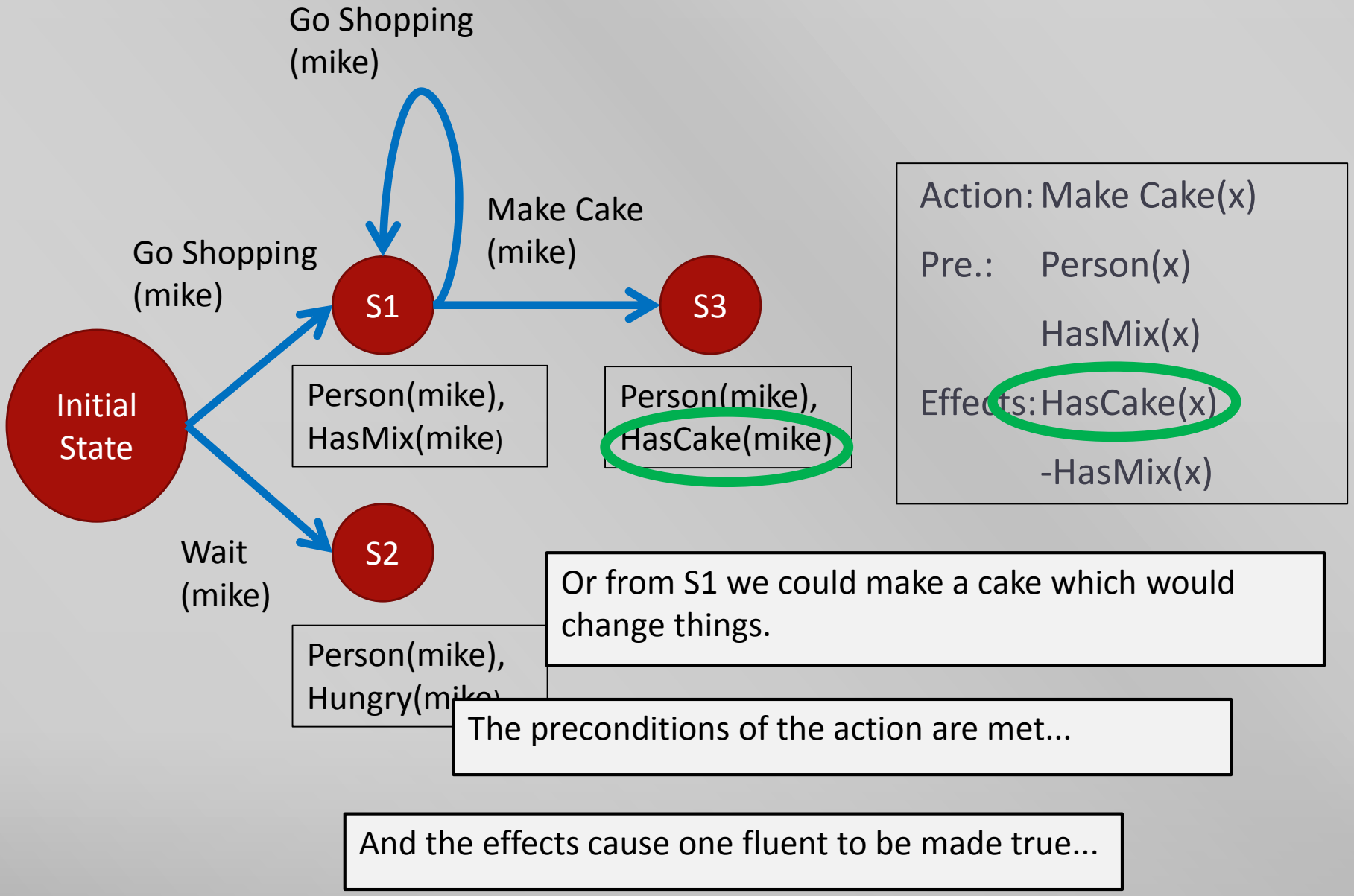
HasMix(x)

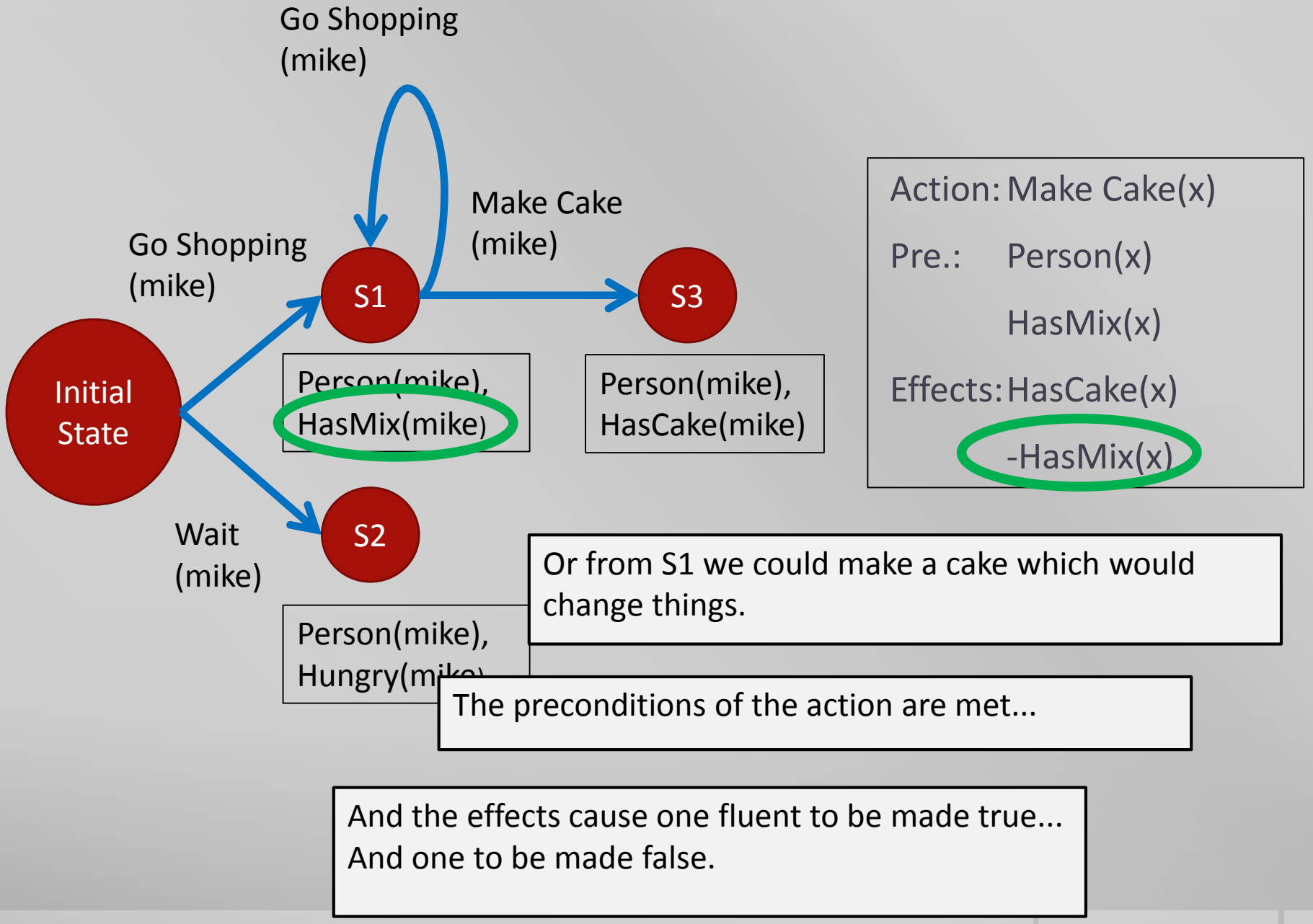
Effects: HasCake(x)

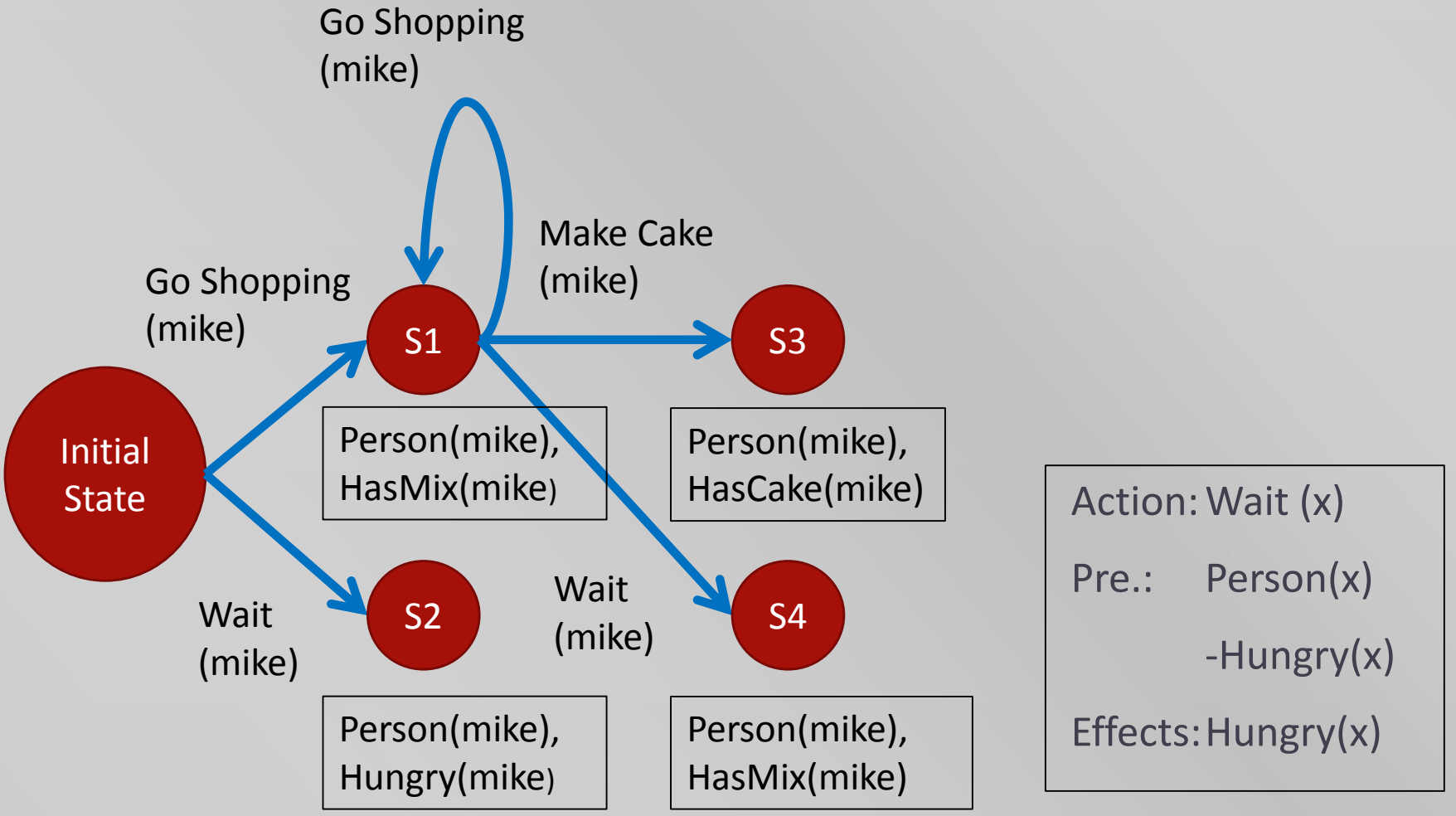
-HasMix(x)

Or from S1 we could make a cake which would change things.


Go Shopping
(mike)
Make Cake
(mike)
Go Shopping
(mike)
S1
S3
Initial
State
Person(mike),
HasMix(mike)
Person(mike),
HasCake(mike)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
-HasMix(x)
Wait
(mike)
S2
Person(mike),
Hungry(mike)
Or from S1 we could make a cake which would change things.
The preconditions of the action are met...


Go Shopping
(mike)
Make Cake
(mike)
Go Shopping
(mike)
S1
S3
Initial
State
Person(mike),
HasMix(mike)
Person(mike),
HasCake(mike)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
-HasMix(x)
Wait
(mike)
S2
Person(mike),
Hungry(mike)
Or from S1 we could make a cake which would change things.
The preconditions of the action are met...
And the effects cause one fluent to be made true...


Go Shopping
(mike)
Make Cake
(mike)
Go Shopping
(mike)
S1
S3
Initial
State
Person(mike),
HasMix(mike)
Person(mike),
HasCake(mike)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
-HasMix(x)
Wait
(mike)
S2
Person(mike),
Hungry(mike)
Or from S1 we could make a cake which would change things.
The preconditions of the action are met...
And the effects cause one fluent to be made true...
And one to be made false.



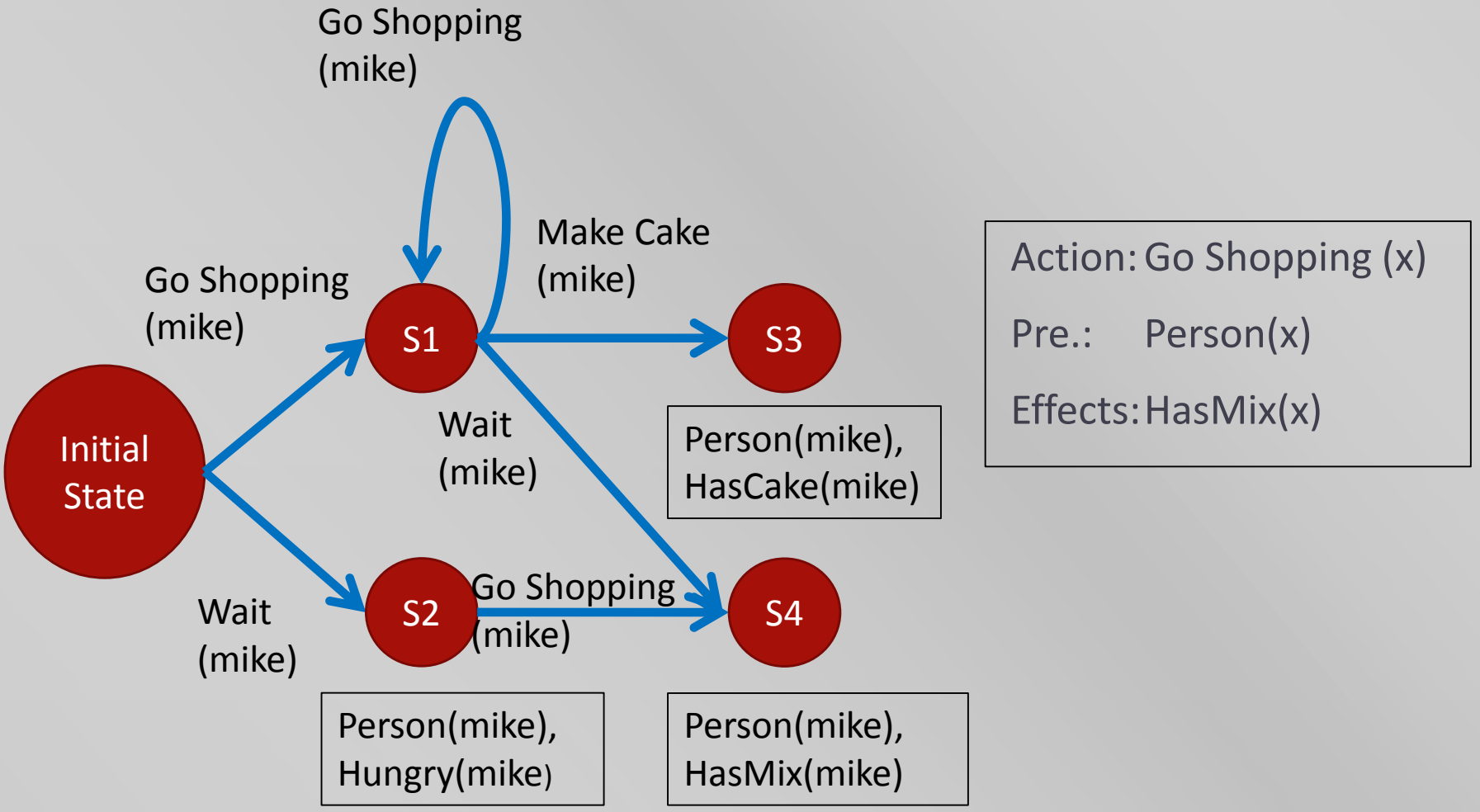
Action: Wait (x)

Pre.: Person(x)

-Hungry(x)

Effects: Hungry(x)

Or we could wait.

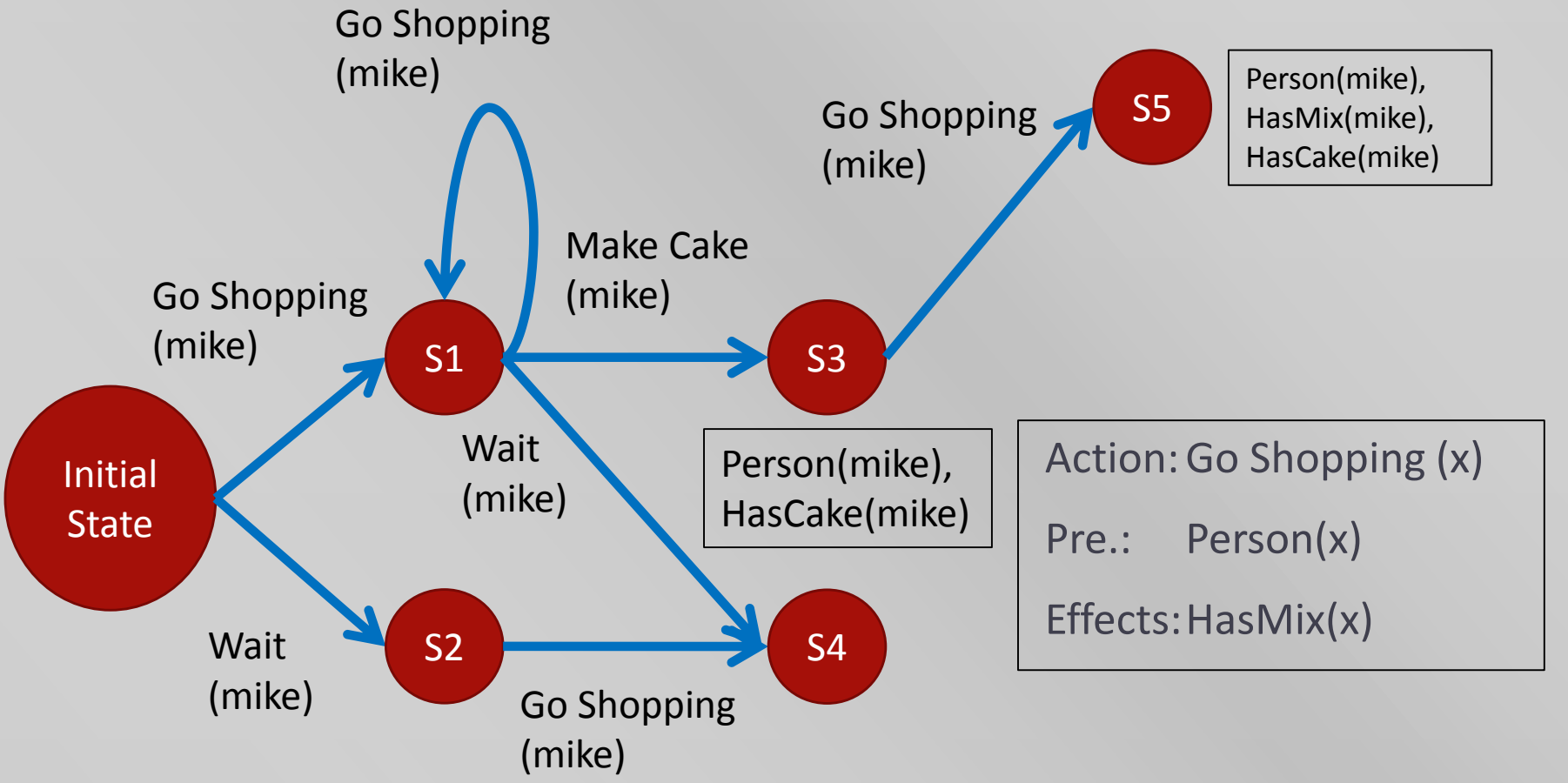


Action: Go Shopping (x)

Pre.: Person(x)

Effects: HasMix(x)

From S2 we can only go shopping. And that will take us to S4 by another route.



From S3 we can go shopping again.

Action: Eat Cake B (x)

Pre.: Person(x)

-Hungry(x)

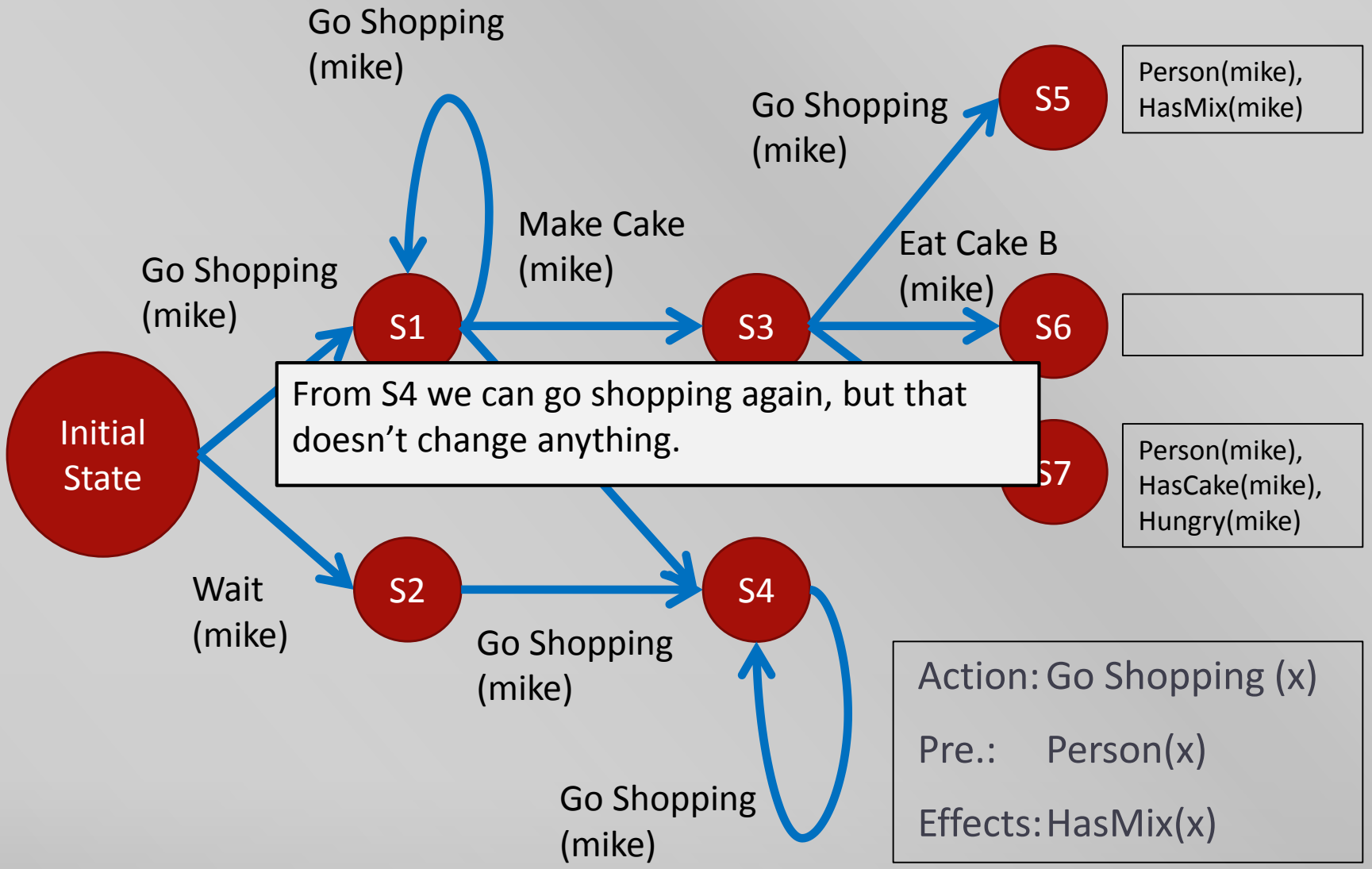
HasCake(x)

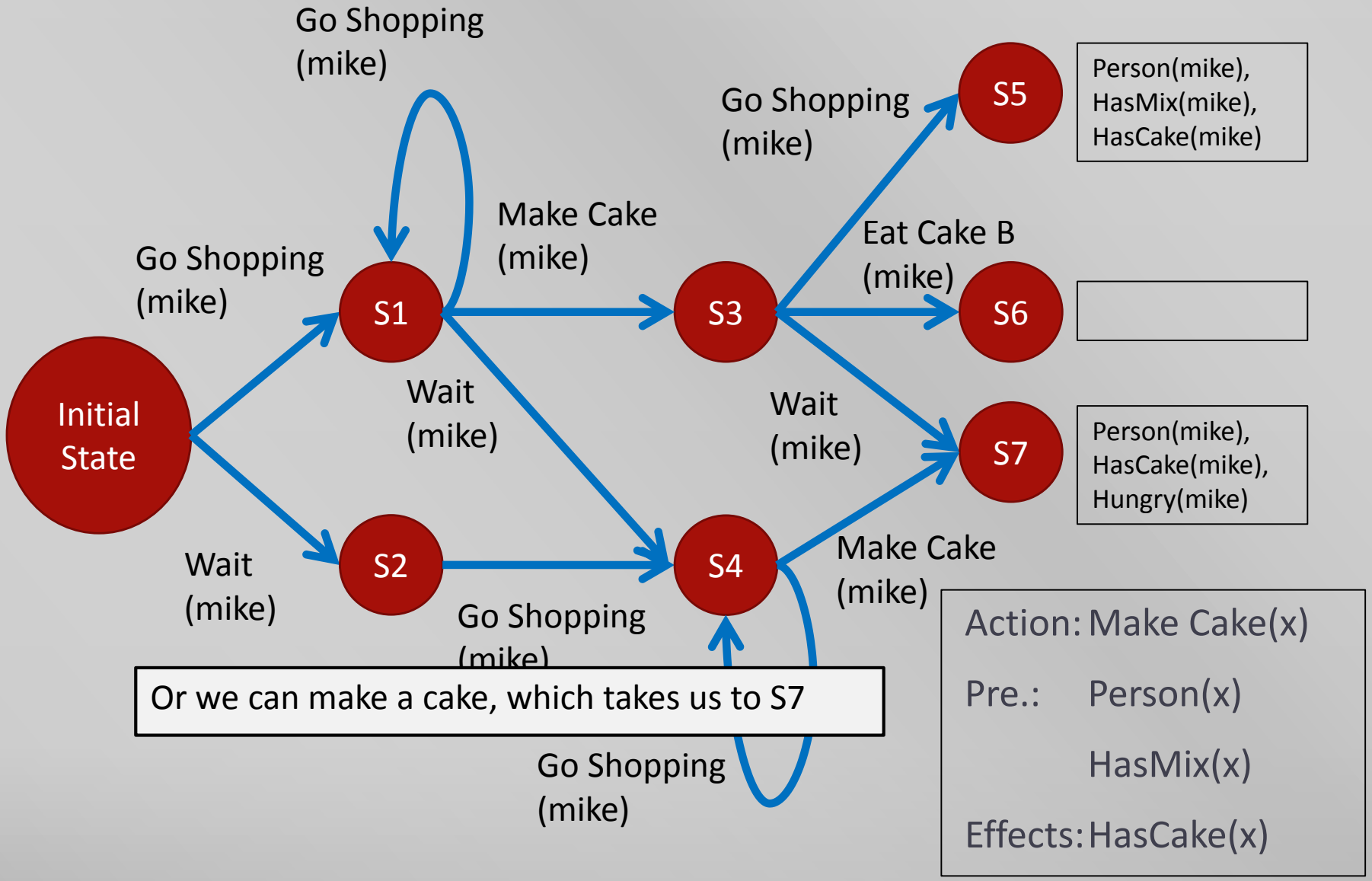
Effects: -Person(x),

-HasCake(x)

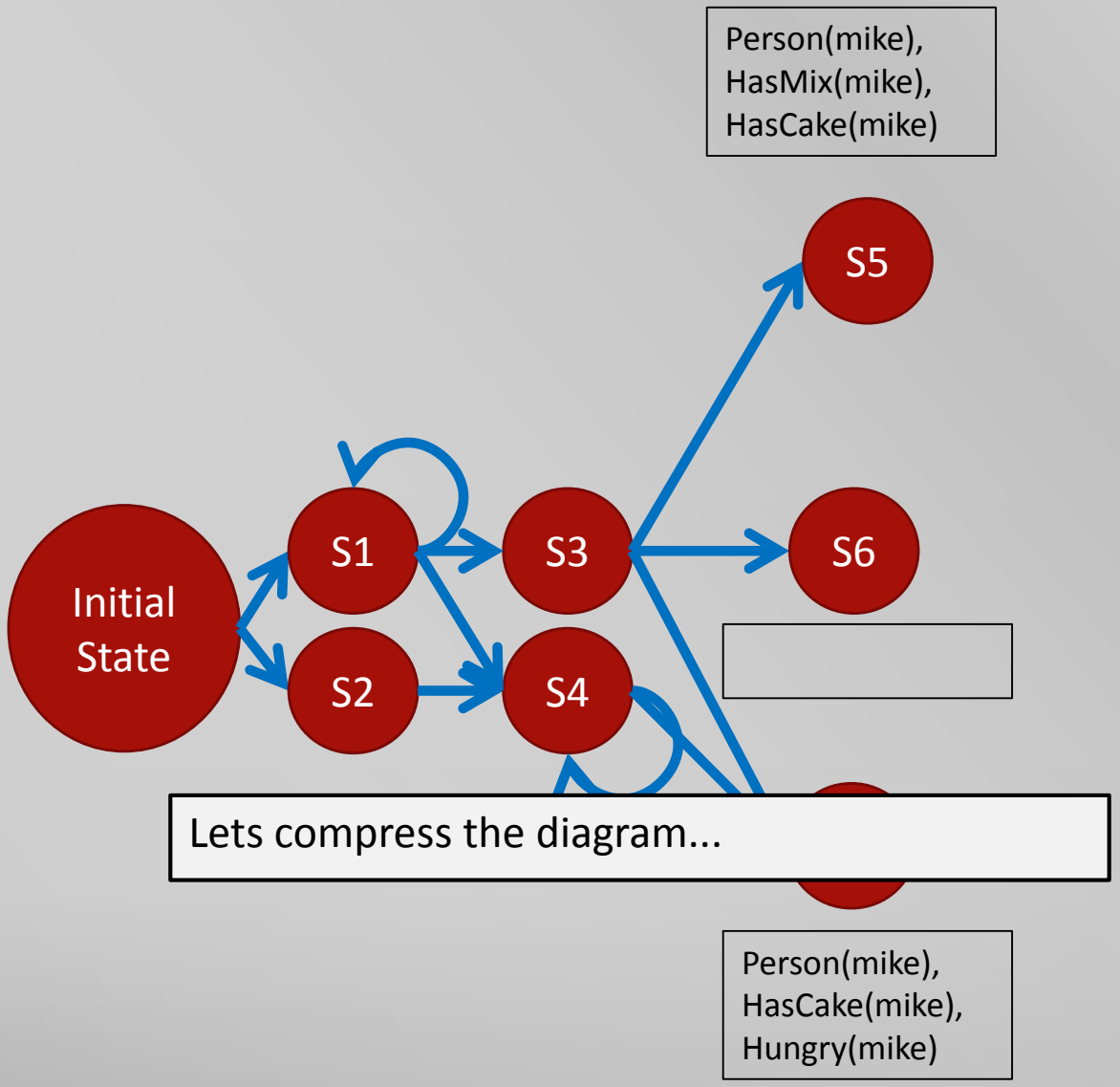
Or we can eat the cake with action 'Eat Cake B' (by omission Mike is not hungry). This leads to a state with no atomic fluents. Everything is negated. Including 'Person(mike)': Mike is dead in this state!

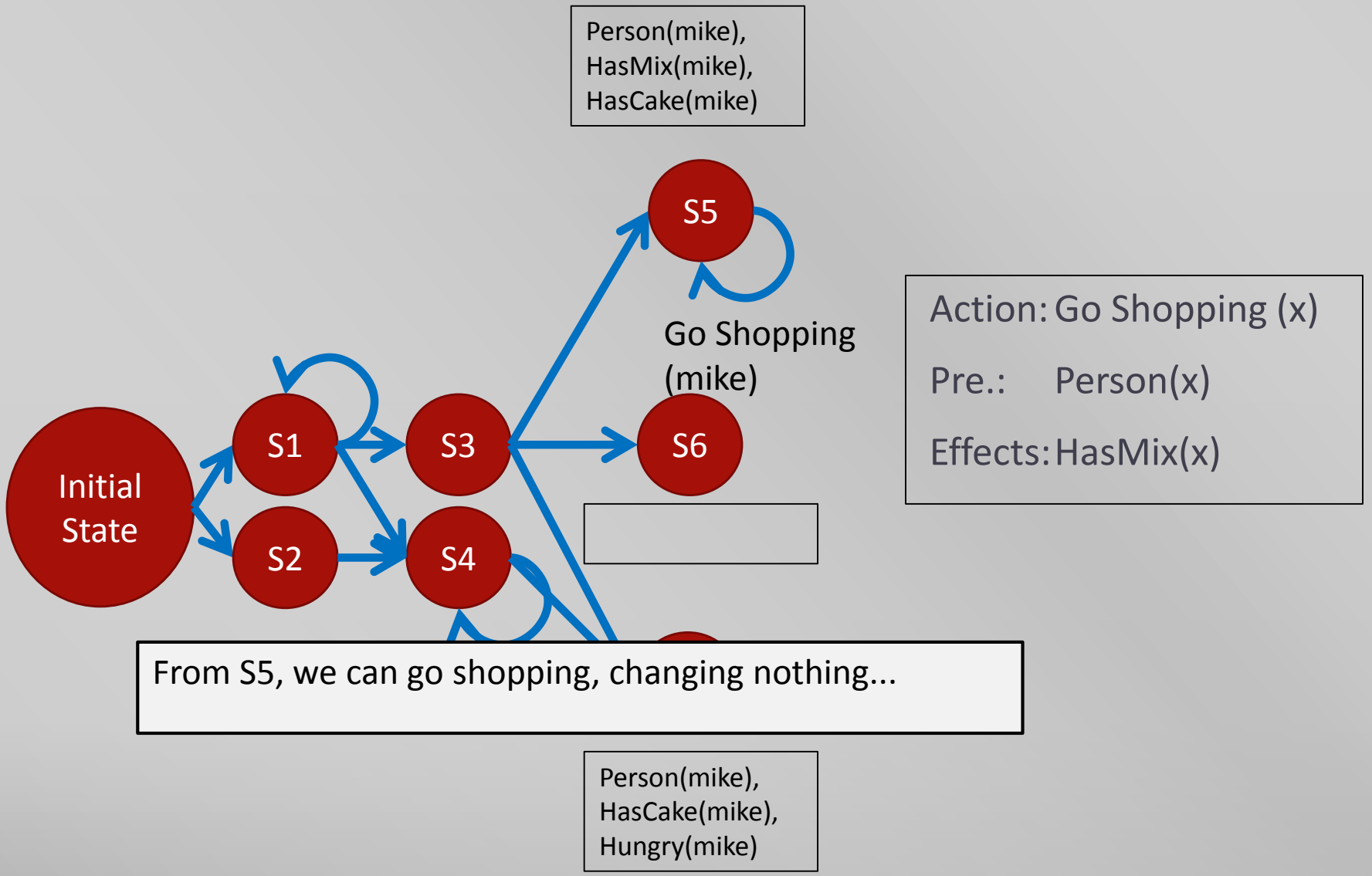

Go Shopping
(mike)
Go Shopping
(mike)
Make Cake
(mike)
Go Shopping
(mike)
S5
Person(mike),
HasMix(mike),
HasCake(mike)
Eat Cake B
(mike)
S1
S3
S6
Initial
State
Wait
(mike)
Person(mike),
HasCake(mike)
S7
Person(mike),
HasCake(mike),
Hungry(mike)
Wait
(mike)
Wait
(mike)
S2
S4
Go Shopping
(mike)
Action: Wait (x)
Pre.: Person(x)
-Hungry(x)
Effects: Hungry(x)
Or we can wait (and Mike lives...)

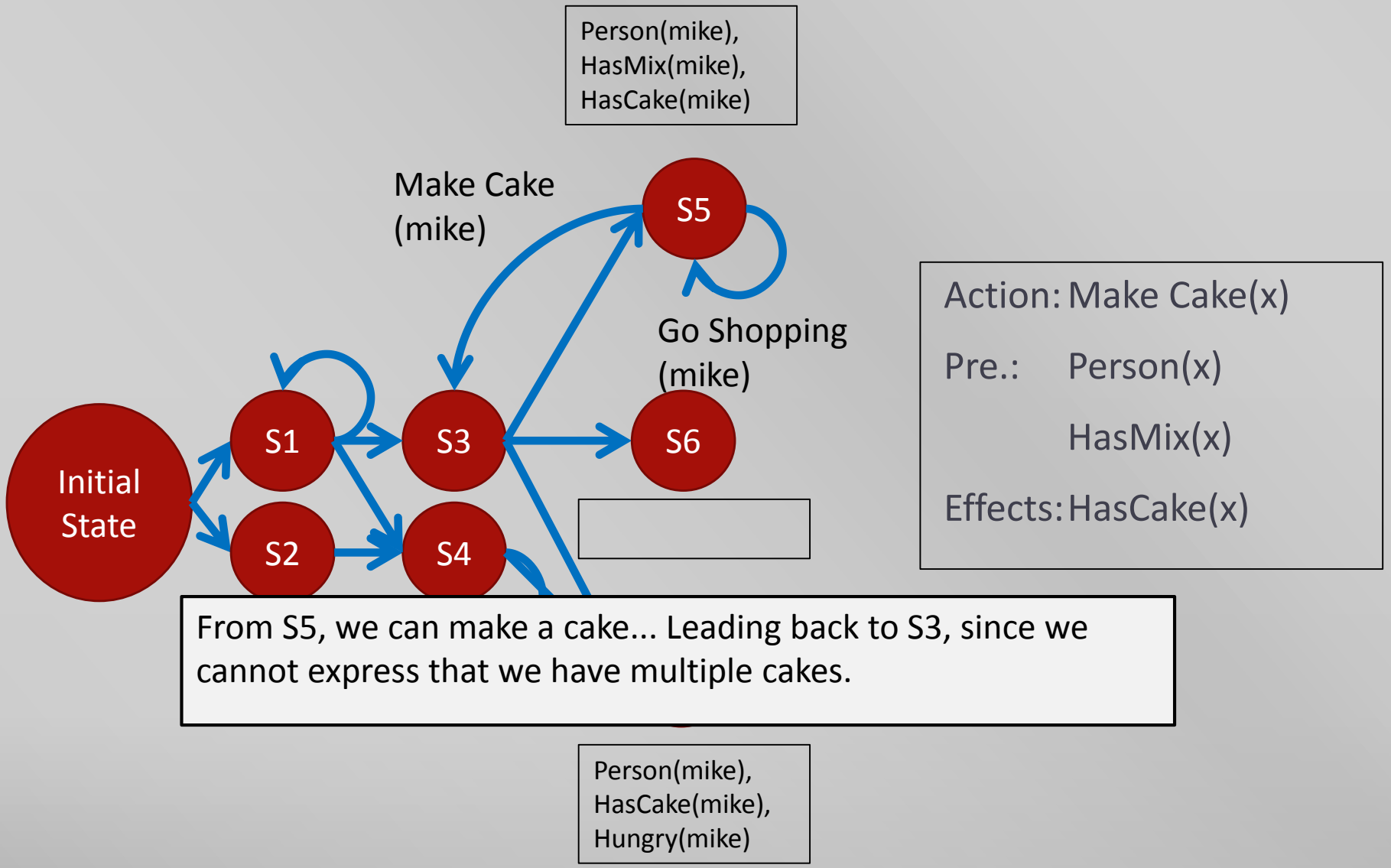

Go Shopping
(mike)
Go Shopping
(mike)
S5
Person(mike),
HasMix(mike)
Make Cake
(mike)
Eat Cake B
(mike)
Go Shopping
(mike)
S1
S3
S6
Initial
State
From S4 we can go shopping again, but that doesn't change anything.
S7
Person(mike),
HasCake(mike),
Hungry(mike)
Wait
(mike)
S2
S4
Go Shopping
(mike)
Go Shopping
(mike)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)

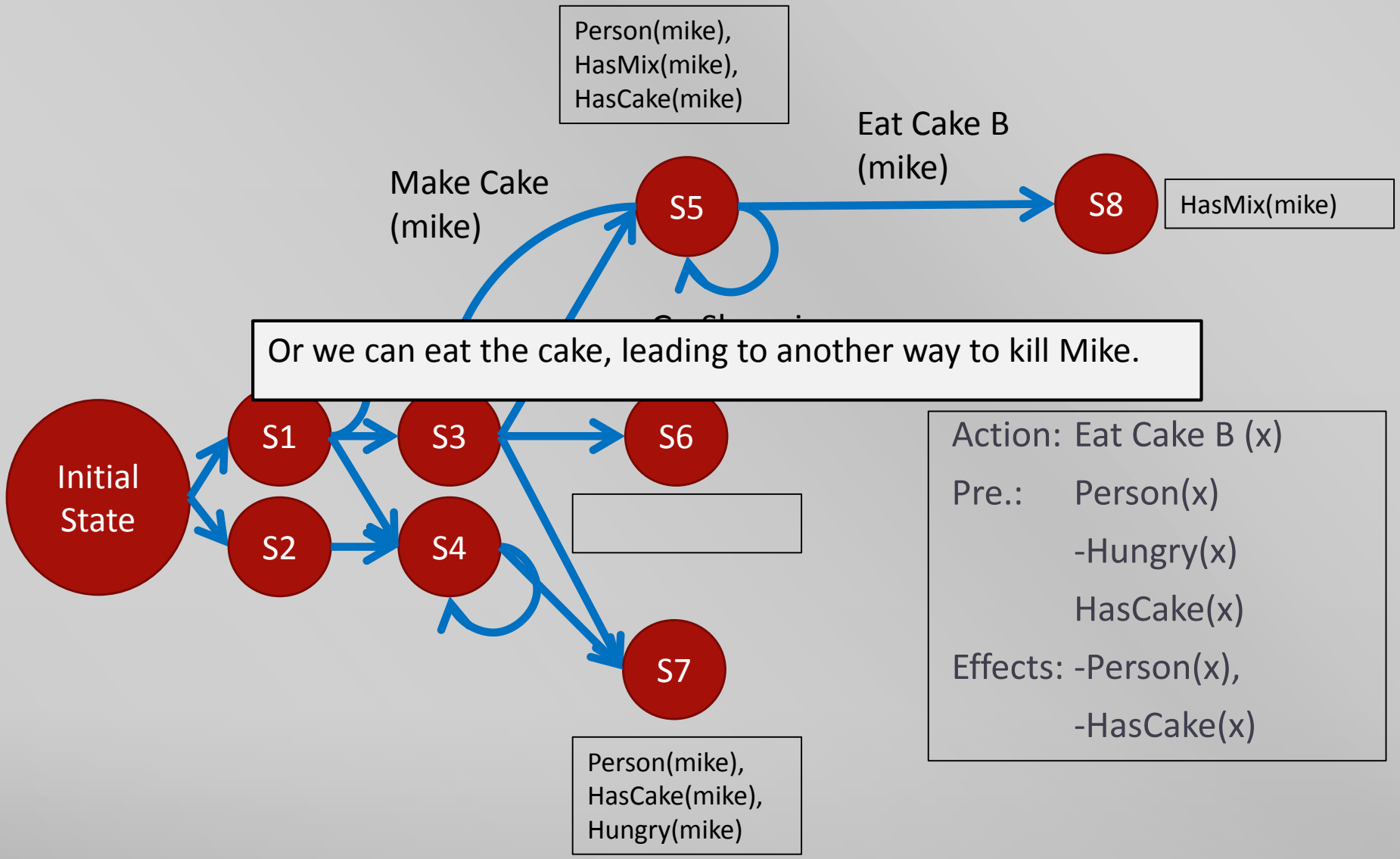

Go Shopping
(mike)
Go Shopping
(mike)
Make Cake
(mike)
Go Shopping
(mike)
Eat Cake B
(mike)
Initial
State
S1
S3
S5
S6
Person(mike),
HasMix(mike),
HasCake(mike)
Wait
(mike)
Wait
(mike)
S7
Person(mike),
HasCake(mike),
Hungry(mike)
Wait
(mike)
S2
S4
Make Cake
(mike)
Go Shopping
(mike)
Or we can make a cake, which takes us to S7
Go Shopping
(mike)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)

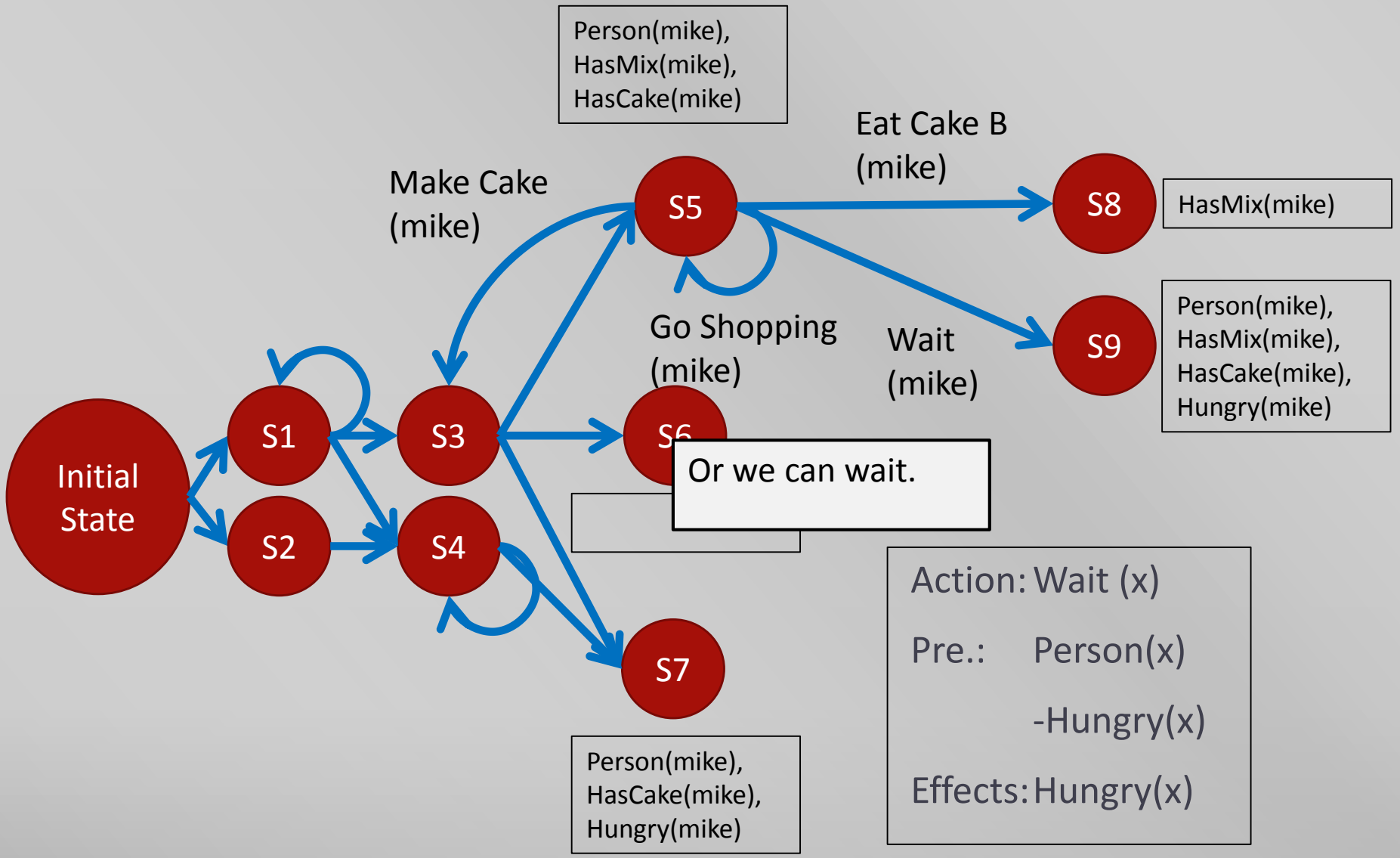

Go Shopping
(mike)
Go Shopping
(mike)
S5
Person(mike),
HasMix(mike),
HasCake(mike)
Make Cake
(mike)
Eat Cake B
(mike)
Go Shopping
(mike)
S1
S3
S6
Initial
State
Wait
(mike)
Wait
(mike)
S7
Person(mike),
HasCake(mike),
Hungry(mike)
Wait
(mike)
S2
S4
Make Cake
(mike)
Go Shopping
(mike)
Lets compress the diagram...
Go Shopping
(mike)
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)

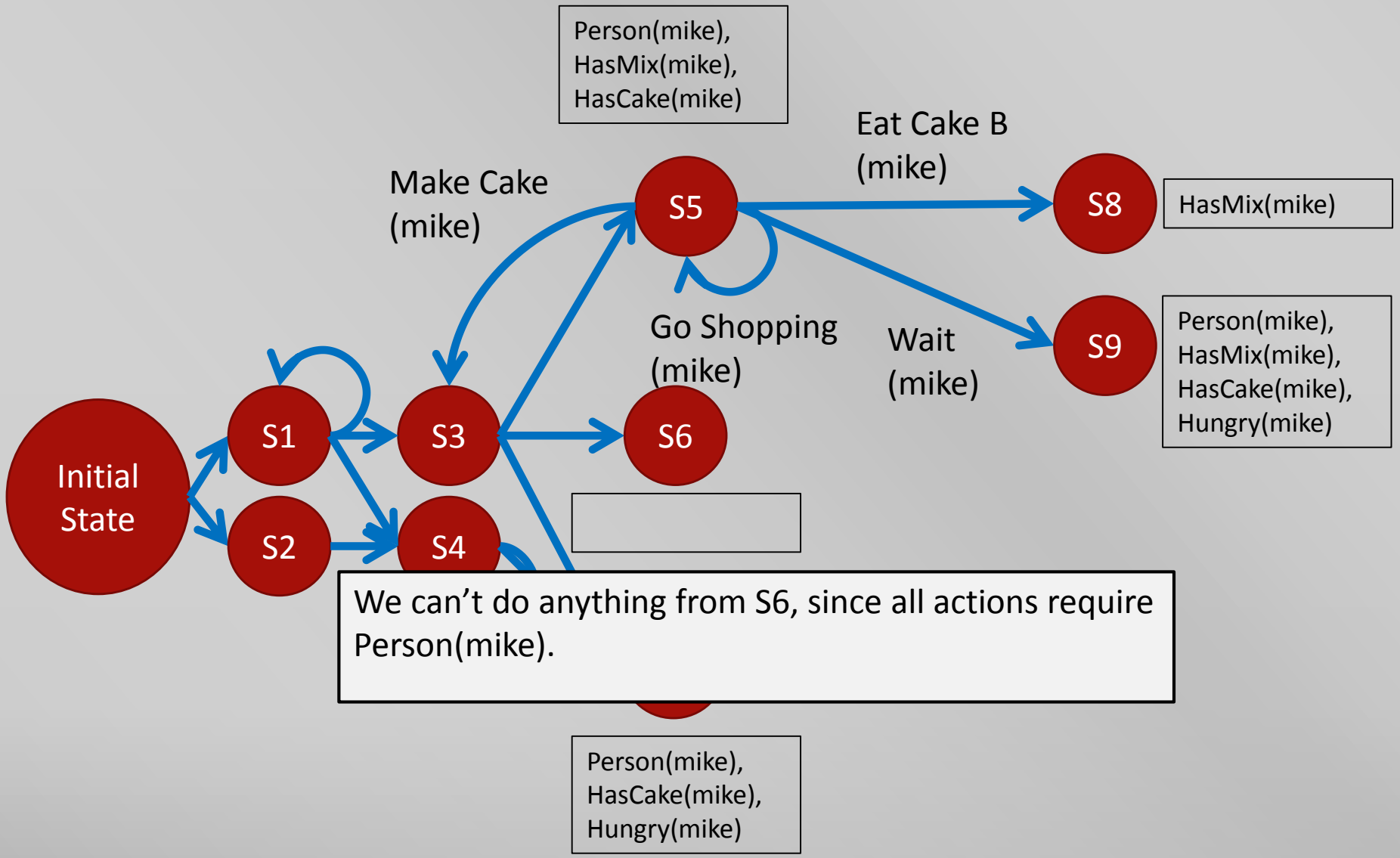

Person(mike),
HasMix(mike),
HasCake(mike)
S5
S1
S3
S6
Initial
State
S2
S4
Lets compress the diagram...
Person(mike),
HasCake(mike),
Hungry(mike)

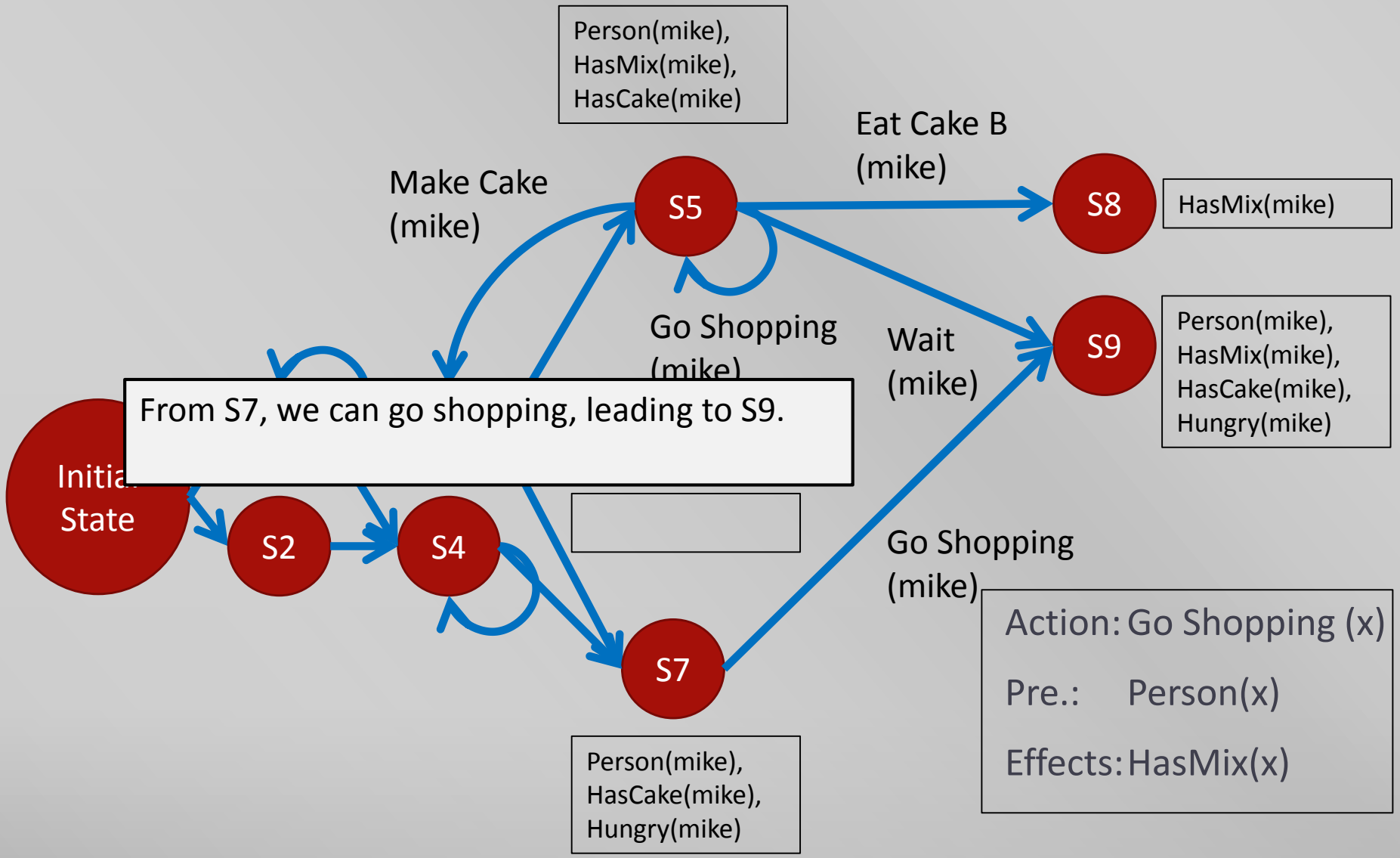

Person(mike),
HasMix(mike),
HasCake(mike)
S5
Go Shopping
(mike)
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
Initial
State
S1
S2
S3
S4
S6
From S5, we can go shopping, changing nothing...
Person(mike),
HasCake(mike),
Hungry(mike)

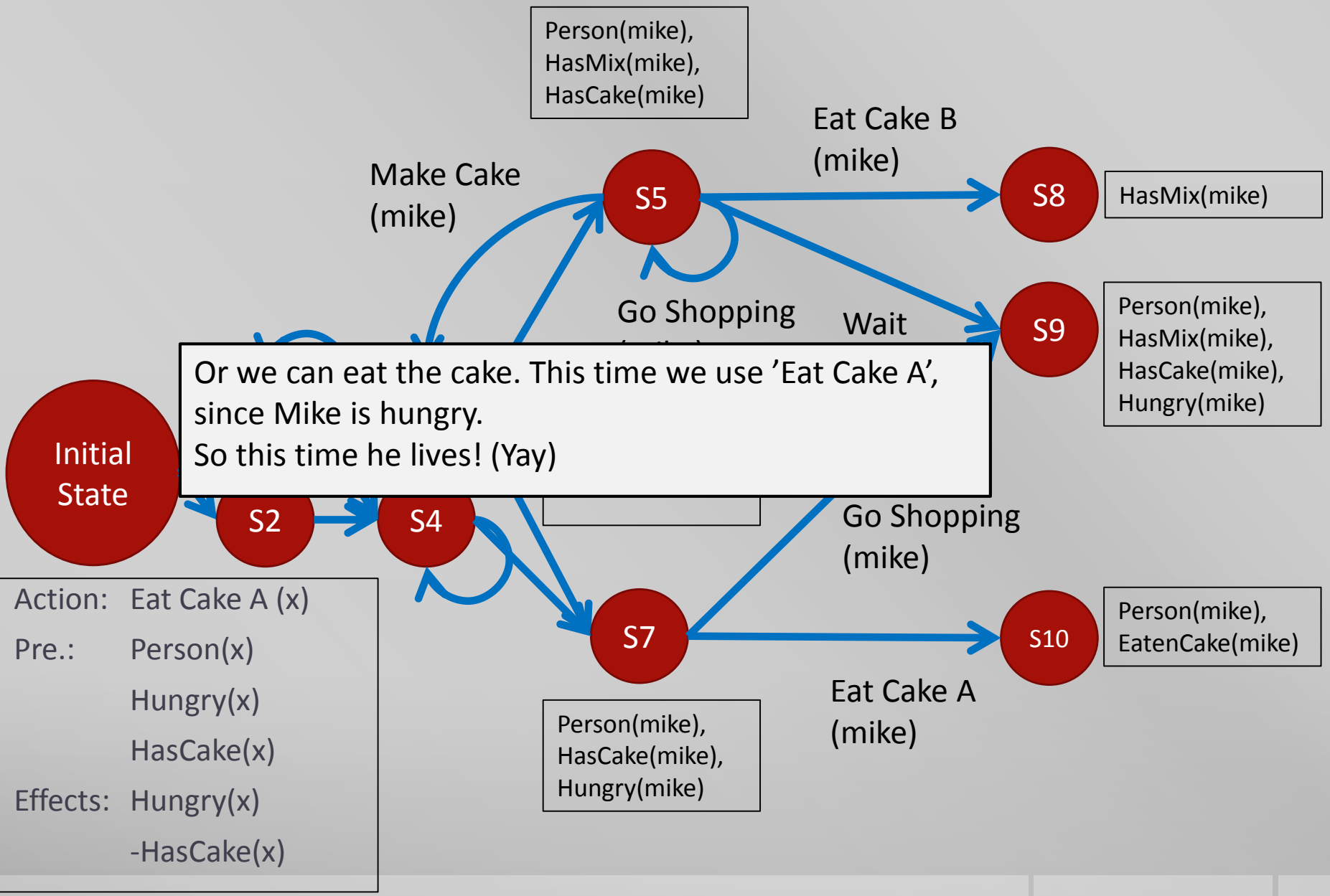

Person(mike),
HasMix(mike),
HasCake(mike)
Make Cake
(mike)
S5
Go Shopping
(mike)
Initial
State
S1
S3
S6
S2
S4
Action: Make Cake(x)
Pre.: Person(x)
HasMix(x)
Effects: HasCake(x)
From S5, we can make a cake... Leading back to S3, since we cannot express that we have multiple cakes.
Person(mike),
HasCake(mike),
Hungry(mike)

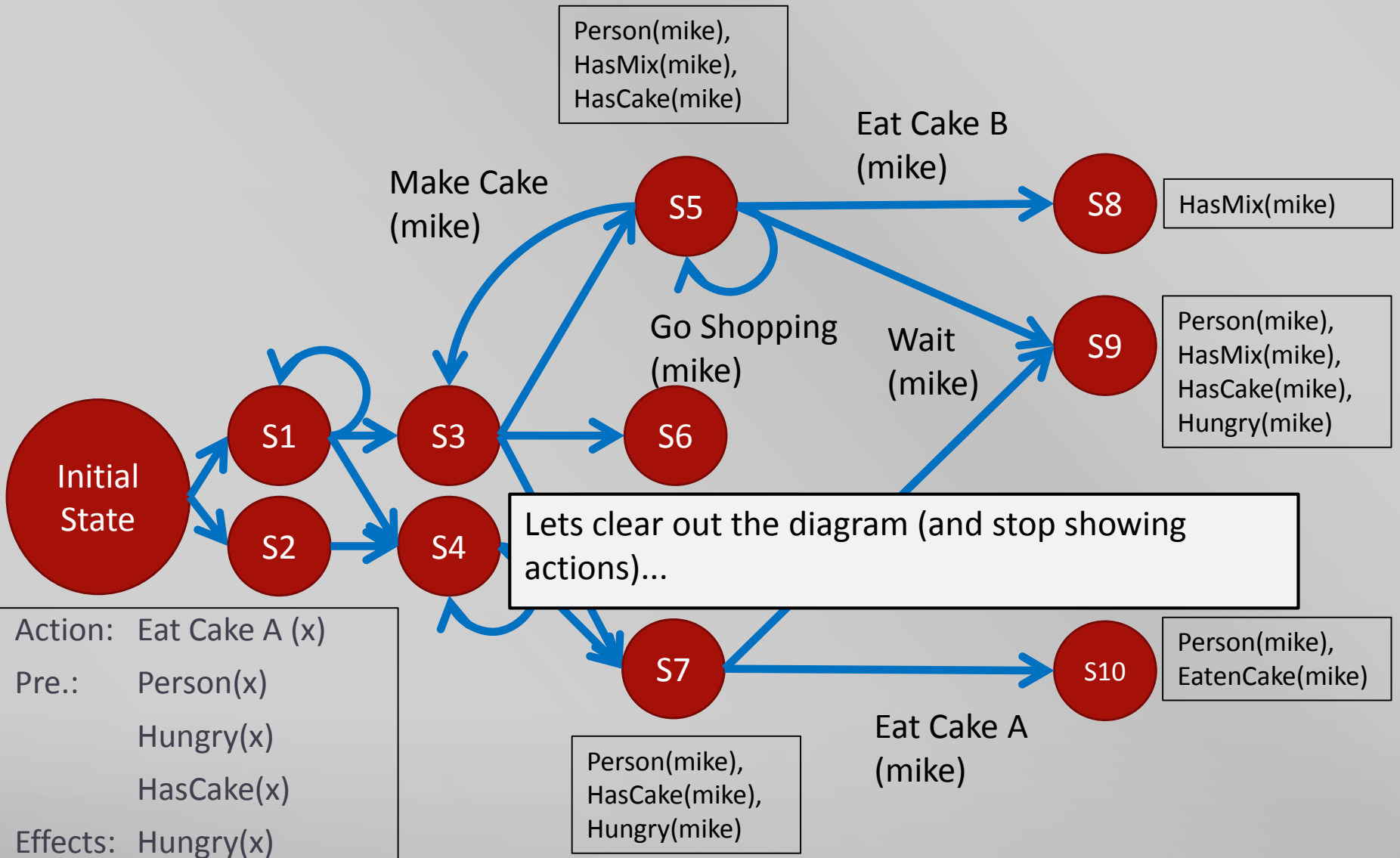

Person(mike), HasMix(mike), HasCake(mike)
Eat Cake B (mike)
Make Cake (mike)
S5
S8
HasMix(mike)
Or we can eat the cake, leading to another way to kill Mike.
Initial State
S1
S2
S3
S4
S6
S7
Person(mike), HasCake(mike), Hungry(mike)
Action: Eat Cake B (x)
Pre.: Person(x)
-Hungry(x)
HasCake(x)
Effects: -Person(x),
-HasCake(x)

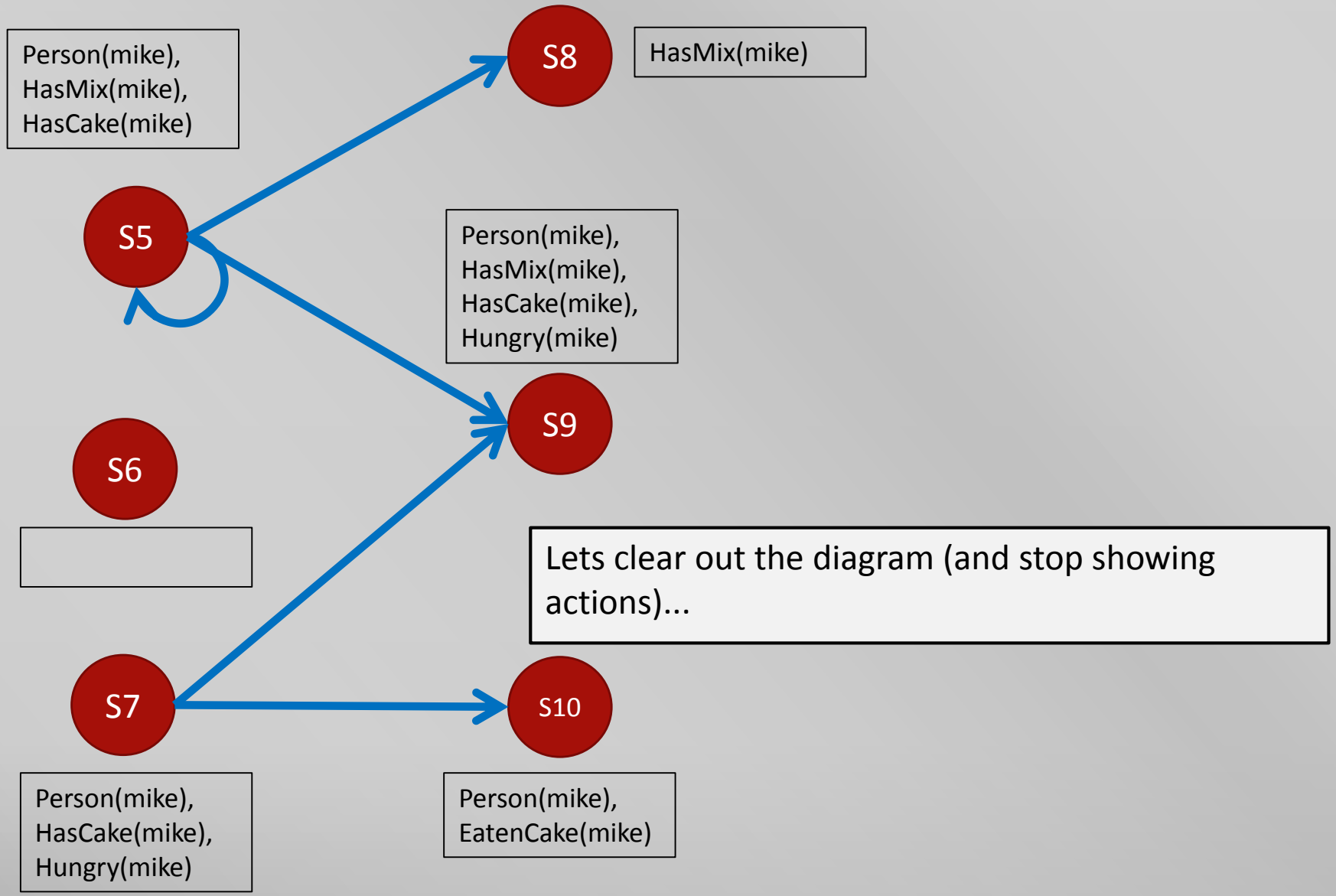

Person(mike),
HasMix(mike),
HasCake(mike)
Eat Cake B
(mike)
Make Cake
(mike)
S5
S8
HasMix(mike)
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
S9
Go Shopping
(mike)
Wait
(mike)
Initial
State
S1
S3
S6
Or we can wait.
S2
S4
S7
Person(mike),
HasCake(mike),
Hungry(mike)
Action: Wait (x)
Pre.: Person(x)
-Hungry(x)
Effects: Hungry(x)

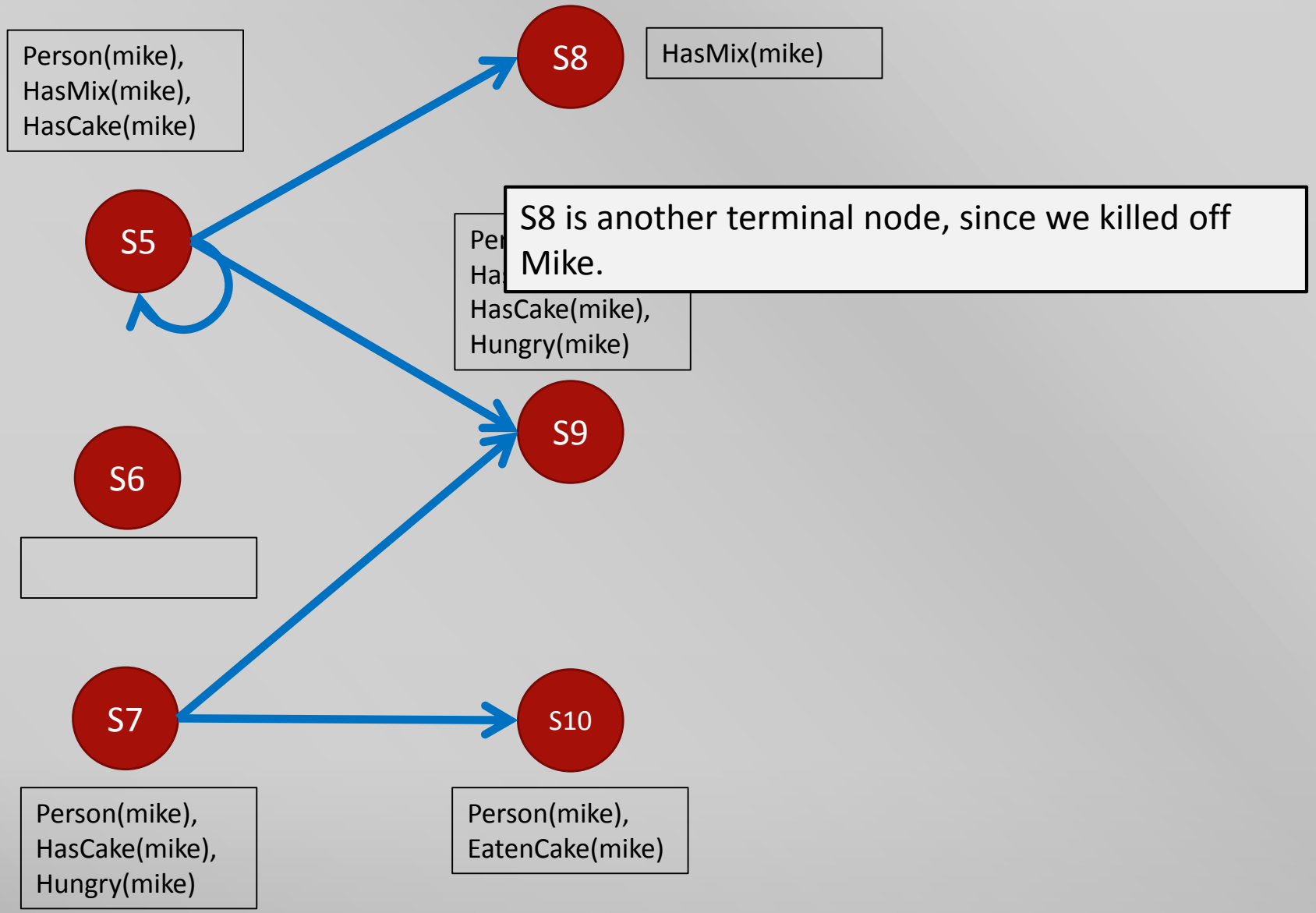

Person(mike),
HasMix(mike),
HasCake(mike)
Eat Cake B
(mike)
Make Cake
(mike)
S5
S8
HasMix(mike)
Go Shopping
(mike)
Wait
(mike)
S9
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
Initial
State
S1
S3
S6
S2
S4
We can’t do anything from S6, since all actions require Person(mike).
Person(mike),
HasCake(mike),
Hungry(mike)

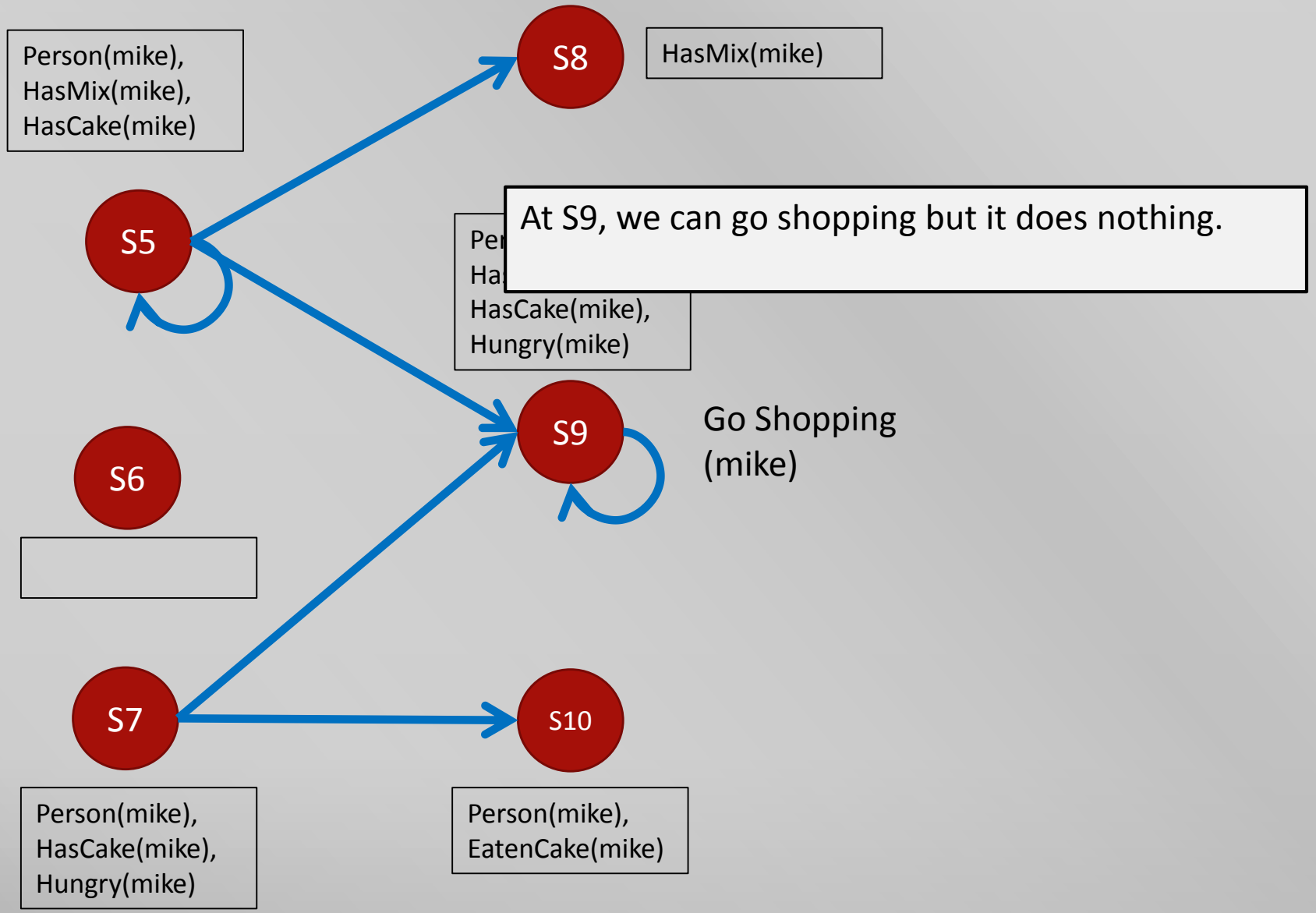

Person(mike),
HasMix(mike),
HasCake(mike)
Eat Cake B
(mike)
Make Cake
(mike)
S5
S8
HasMix(mike)
Go Shopping
(mike)
Wait
(mike)
S9
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
From S7, we can go shopping, leading to S9.
Initia
State
S2
S4
Go Shopping
(mike)
S7
Action: Go Shopping (x)
Pre.: Person(x)
Effects: HasMix(x)
Person(mike),
HasCake(mike),
Hungry(mike)

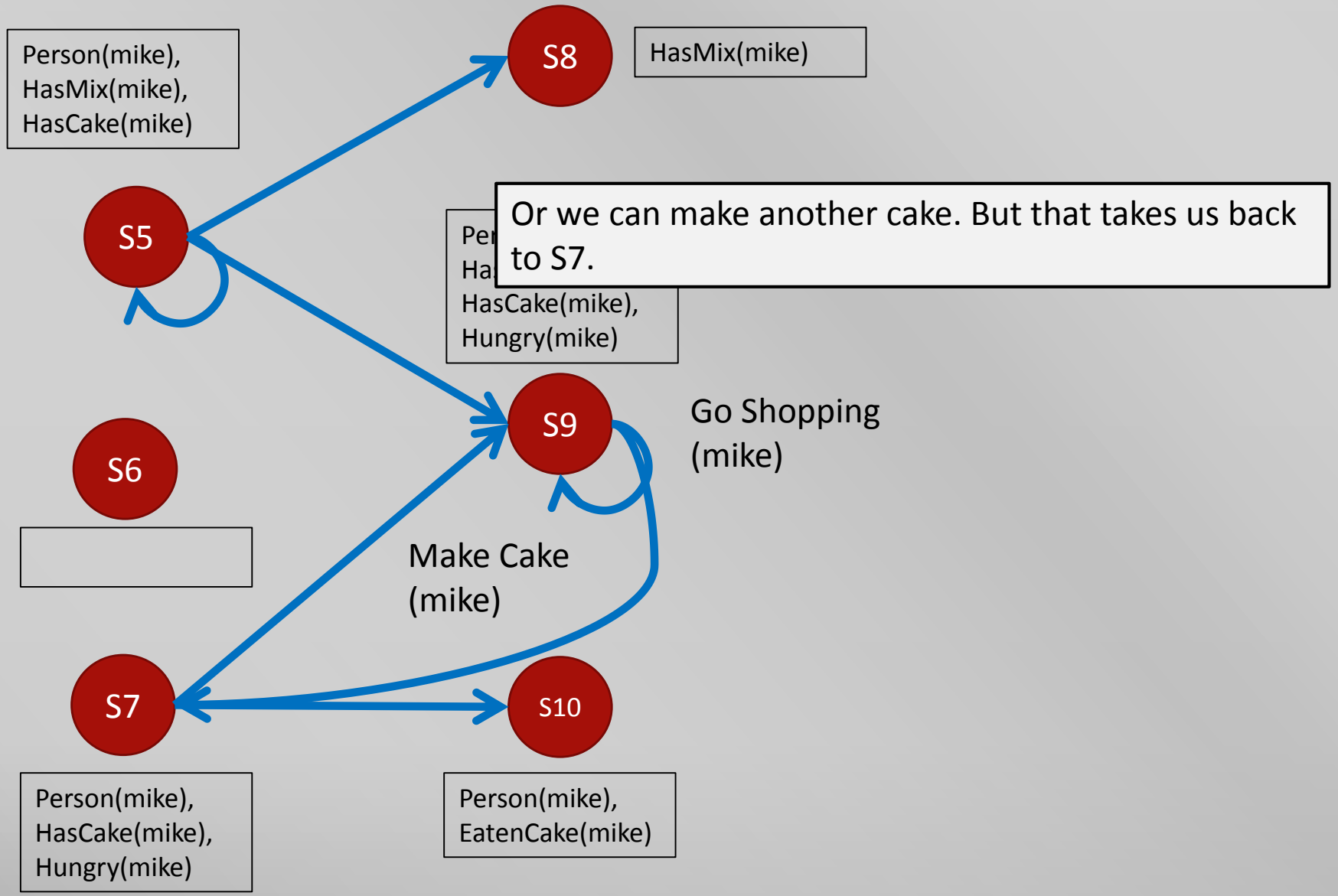

Person(mike), HasMix(mike), HasCake(mike)
Eat Cake B (mike)
Make Cake (mike)
S5
S8
HasMix(mike)
Go Shopping
Wait
S9
Person(mike), HasMix(mike), HasCake(mike), Hungry(mike)
Or we can eat the cake. This time we use 'Eat Cake A', since Mike is hungry.
So this time he lives! (Yay)
Initial State
S2
S4
Go Shopping (mike)
S7
S10
Person(mike), EatenCake(mike)
Eat Cake A (mike)
Person(mike), HasCake(mike), Hungry(mike)
Action: Eat Cake A (x)
Pre.: Person(x)
Hungry(x)
HasCake(x)
Effects: Hungry(x)
-HasCake(x)

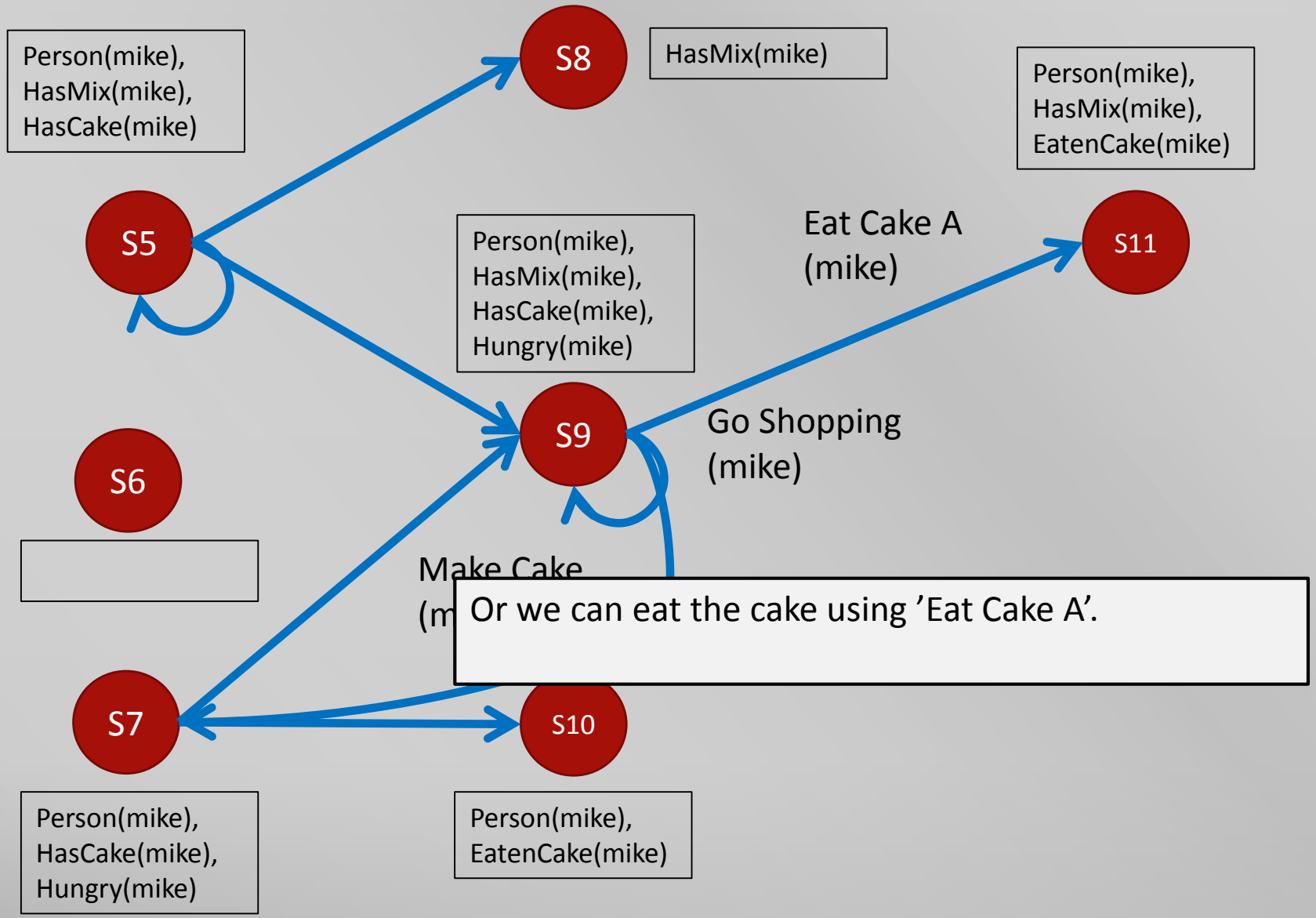

Person(mike), HasMix(mike), HasCake(mike)
Eat Cake B (mike)
Make Cake (mike)
S5
S8
HasMix(mike)
Go Shopping (mike)
Wait (mike)
S9
Person(mike), HasMix(mike), HasCake(mike), Hungry(mike)
Initial State
S1
S3
S6
S2
S4
Lets clear out the diagram (and stop showing actions)...
S7
S10
Person(mike), EatenCake(mike)
Eat Cake A (mike)
Person(mike), HasCake(mike), Hungry(mike)
Action: Eat Cake A (x)
Pre.: Person(x)
Hungry(x)
HasCake(x)
Effects: Hungry(x)
-HasCake(x)

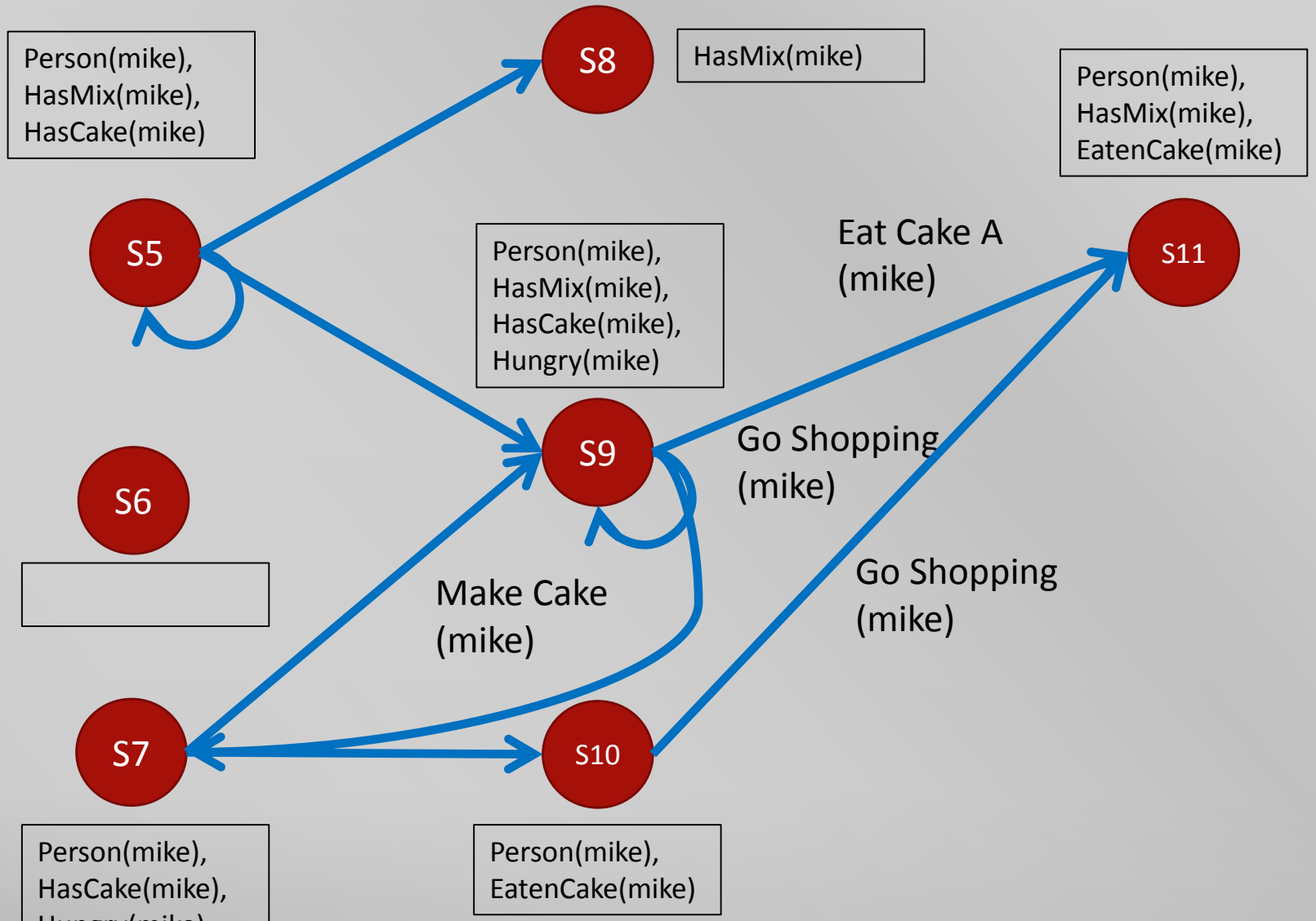

Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
S5
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
S9
S6
Lets clear out the diagram (and stop showing actions)...
S7
S10
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)


Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
S5
S8 is another terminal node, since we killed off Mike.
HasCake(mike),
Hungry(mike)
S9
S6
S7
S10
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)

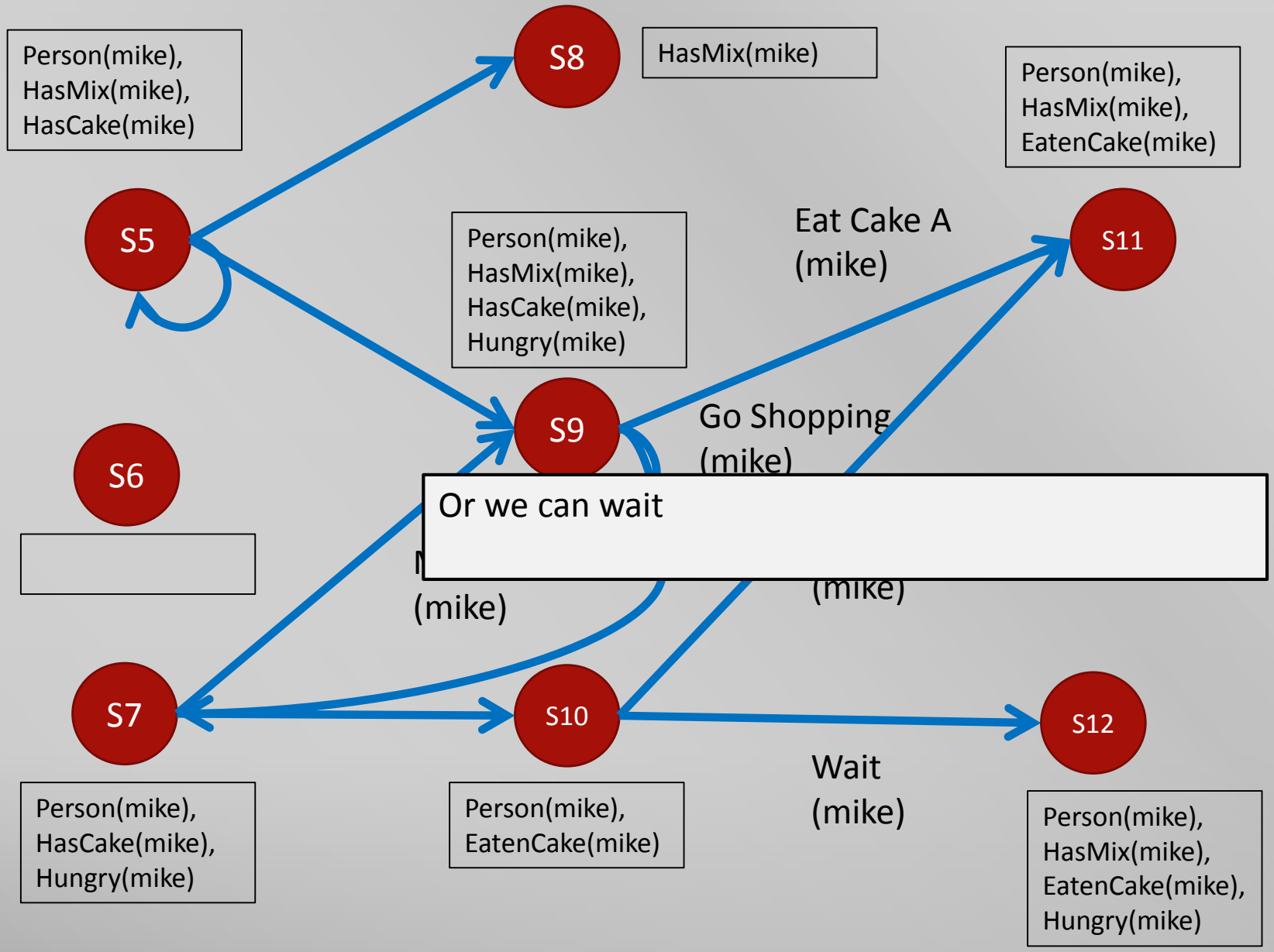

Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
S5
At S9, we can go shopping but it does nothing.
HasCake(mike),
Hungry(mike)
S9
Go Shopping
(mike)
S6
S7
S10
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)

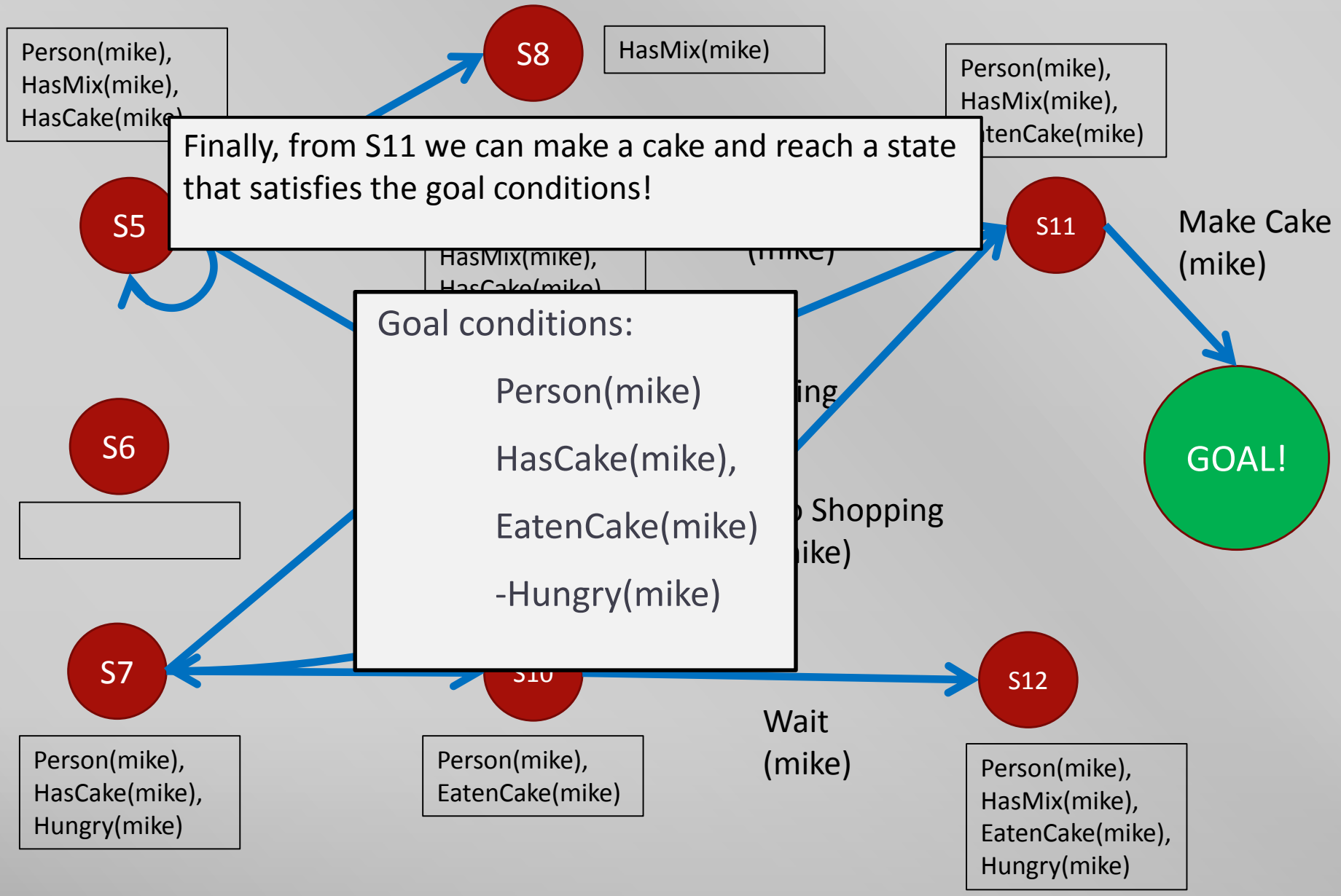

Person(mike), HasMix(mike), HasCake(mike)
S8
HasMix(mike)
S5
Or we can make another cake. But that takes us back to S7.
HasCake(mike), Hungry(mike)
S9
Go Shopping (mike)
S6
Make Cake (mike)
S7
S10
Person(mike), HasCake(mike), Hungry(mike)
Person(mike), EatenCake(mike)

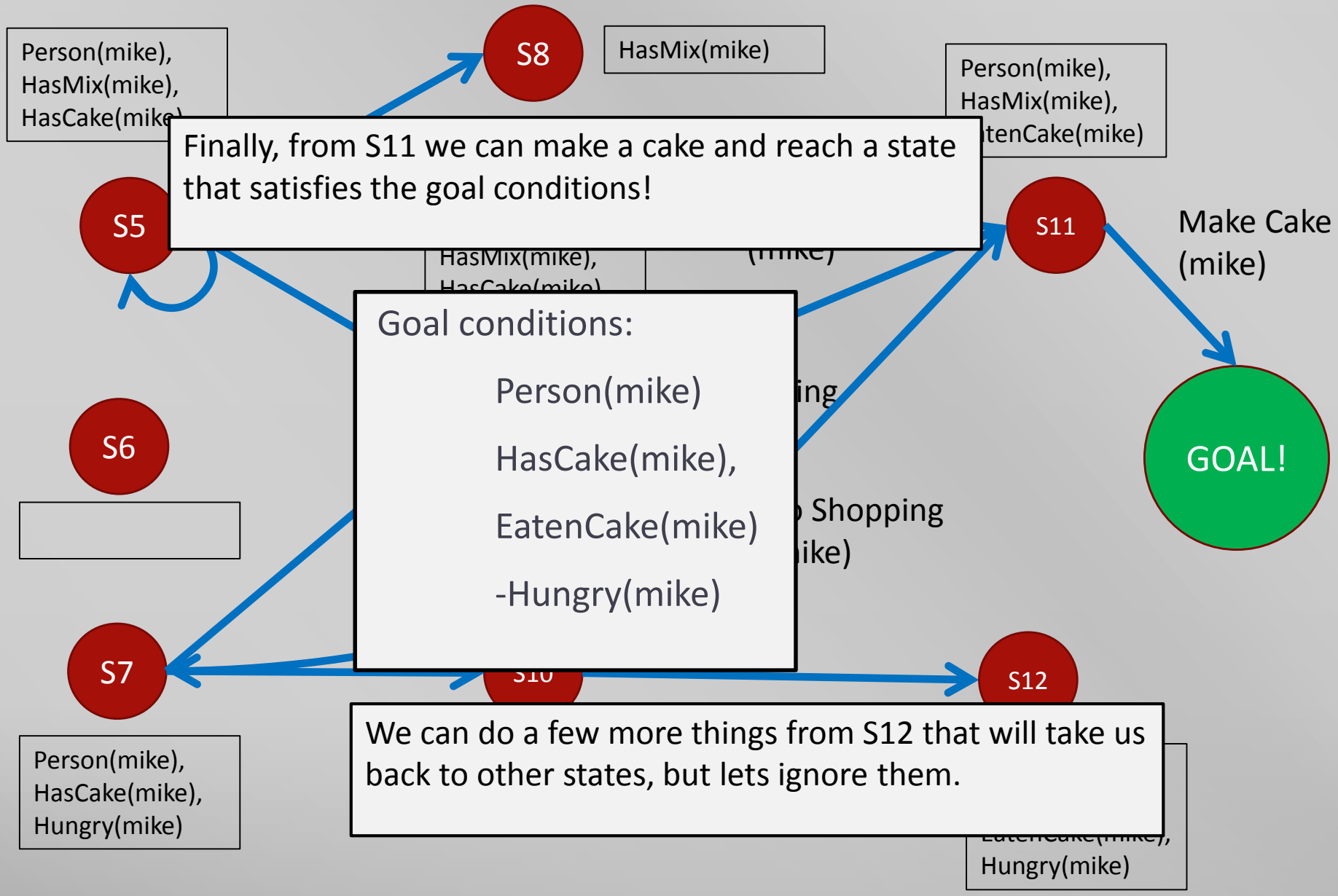

Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
Person(mike),
HasMix(mike),
EatenCake(mike)
S5
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
Eat Cake A
(mike)
S11
S9
Go Shopping
(mike)
S6
Make Cake
(m
Or we can eat the cake using 'Eat Cake A'.
S7
S10
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)



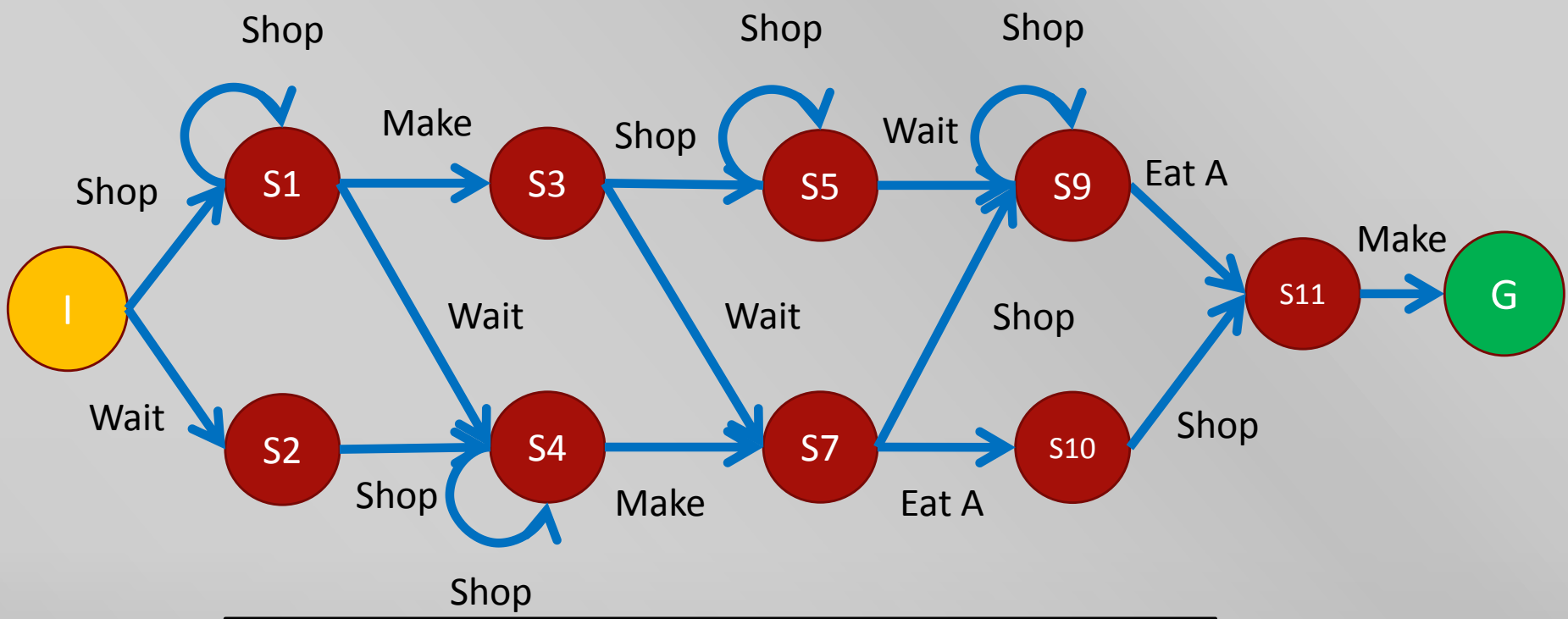
At S10, we can go shopping, which also leads to S11.


Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
Person(mike),
HasMix(mike),
EatenCake(mike)
S5
Person(mike),
HasMix(mike),
HasCake(mike),
Hungry(mike)
Eat Cake A
(mike)
S11
S9
Go Shopping
(mike)
S6
Or we can wait
(mike)
(mike)
S7
S10
S12
Wait
(mike)
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)
Person(mike),
HasMix(mike),
EatenCake(mike),
Hungry(mike)


Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
Person(mike),
HasMix(mike),
tenCake(mike)
Finally, from S11 we can make a cake and reach a state that satisfies the goal conditions!
S5
S11
Make Cake
(mike)
HasMix(mike),
Goal conditions:
Person(mike)
HasCake(mike),
EatenCake(mike)
-Hungry(mike)
ing
S6
GOAL!
Shopping
ike)
S7
S12
Wait
(mike)
Person(mike),
HasCake(mike),
Hungry(mike)
Person(mike),
EatenCake(mike)
Person(mike),
HasMix(mike),
EatenCake(mike),
Hungry(mike)


Person(mike),
HasMix(mike),
HasCake(mike)
S8
HasMix(mike)
Person(mike),
HasMix(mike),
tenCake(mike)
Finally, from S11 we can make a cake and reach a state that satisfies the goal conditions!
S5
S11
Make Cake
(mike)
Goal conditions:
Person(mike)
HasCake(mike),
EatenCake(mike)
-Hungry(mike)
S6
GOAL!
Shopping
ike)
S7
S12
We can do a few more things from S12 that will take us back to other states, but lets ignore them.
Person(mike),
HasCake(mike),
Hungry(mike)
Hungry(mike)

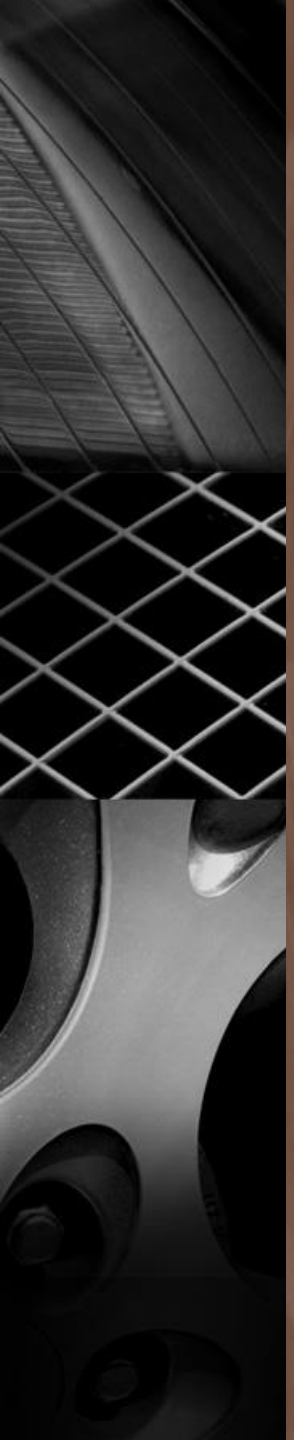
Backtrack to find mutliple paths/plans.



And we can backtrack though the graph (not the search tree, which we didn't show) to find possible paths.

Points to note

- Naive searches are not going to scale.
- Limitations regarding expression
  - Really for deterministic domains.
  - Can have difficulties with disjunctions (numerics a case in point).
- Specifications of actions VERY important.
  - Can overcome expressive limitations
  - Can avoid pointless searching.
- Loops in the search space
- Orphan nodes in the search space
- Potentially multiple goal nodes.
- Potentially multiple paths from the initial state to a goal node.
  - Often due to the fact that we can perform the same steps in multiple orders.



Solving PDDL problems: A* & Planning Graphs

Planning Graphs

- Directed graphs, organized into alternating state and action levels:

  $s_0, a_0, s_1, a_1 \ldots s_n$.

- Each action at a_i is connected to its precondition at s_i and effects at s_{i+1}.
- Trivial 'no-op' actions are given for all literals. No-op action for literal P:

  Preconditions: P

  Effects: P

- Tracks (some) mutual exclusion relations.

Mutual Exclusion

Level a_i contains all actions that can occur at level s_i and records mutual exclusion links between incompatible actions. Two actions are recorded as incompatible if:

- Inconsistent effects – one action's effect negates an effect of the other.
- Interference – one action's effect is the negation of a precondition of the other.
- Competing needs – a precondition for one action is marked as mutually exclusive to a precondition for the other.

Level s_{i+1} contains all literals could result from any subset of compatible actions at level a_i and records mutual exclusion links between incompatible literals. Two literals are recorded as incompatible if:

- Contradictory – One is the negation of the other.
- Inconsistent support – If each possible pair of actions that could produce the literals are mutually exclusive.

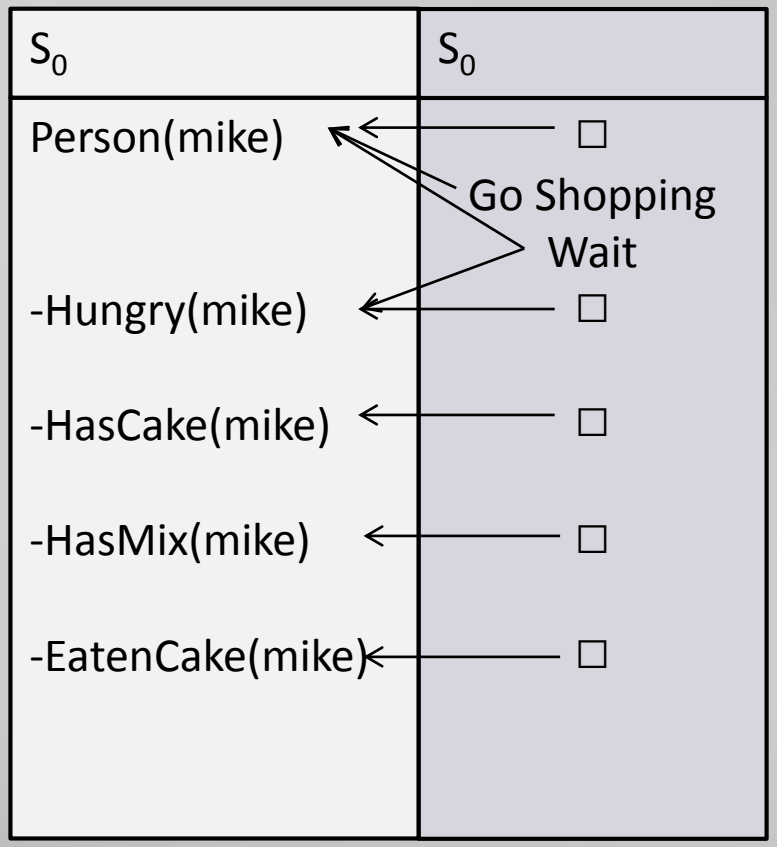
Note: Not all mutual exclusions are tracked. Literals can appear before they are actually possible!

Planning Graph

S_0
Person(mike)
-Hungry(mike)
-HasCake(mike)
-HasMix(mike)
-EatenCake(mike)

The first state layer has all the literals true in the initial state.

Planning Graph

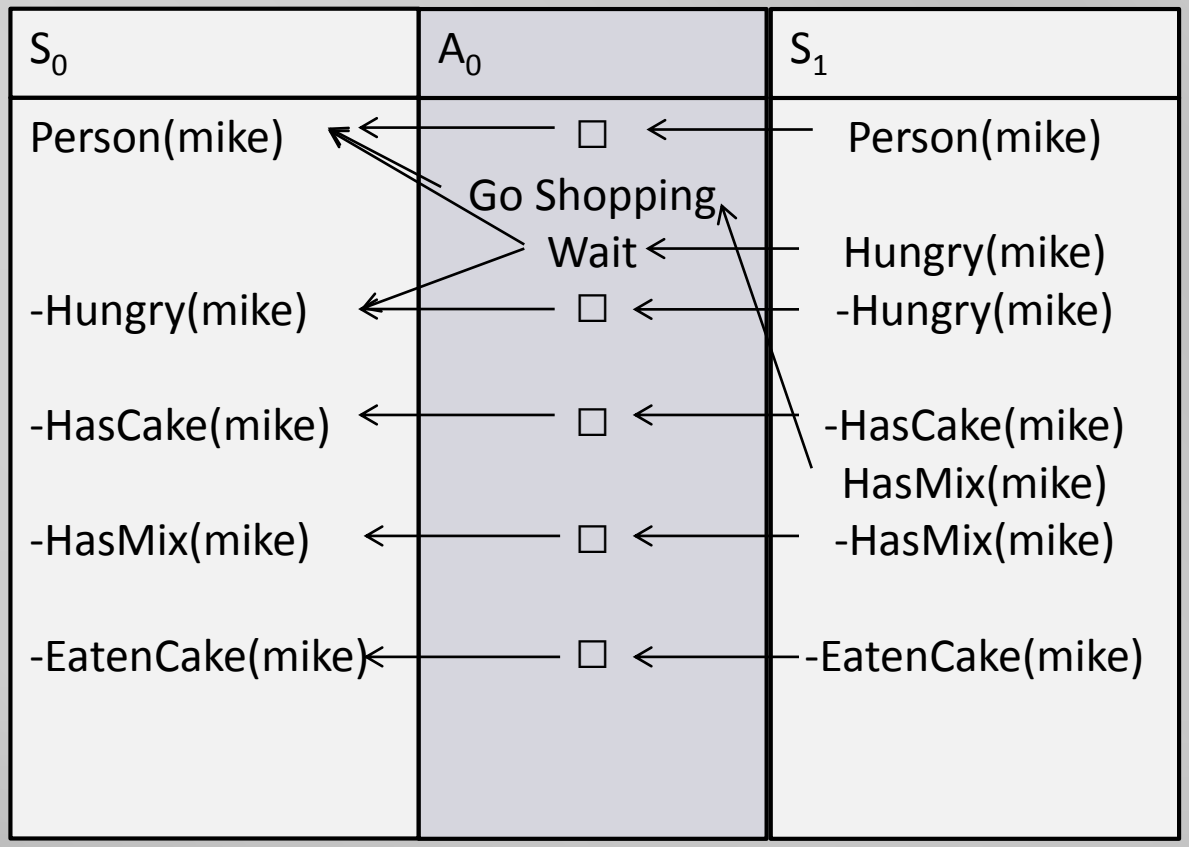


Each action layer has all the actions that could be perform given the first state layer.

Actions point to their preconditions.

Trivial actions are represented by '□'.

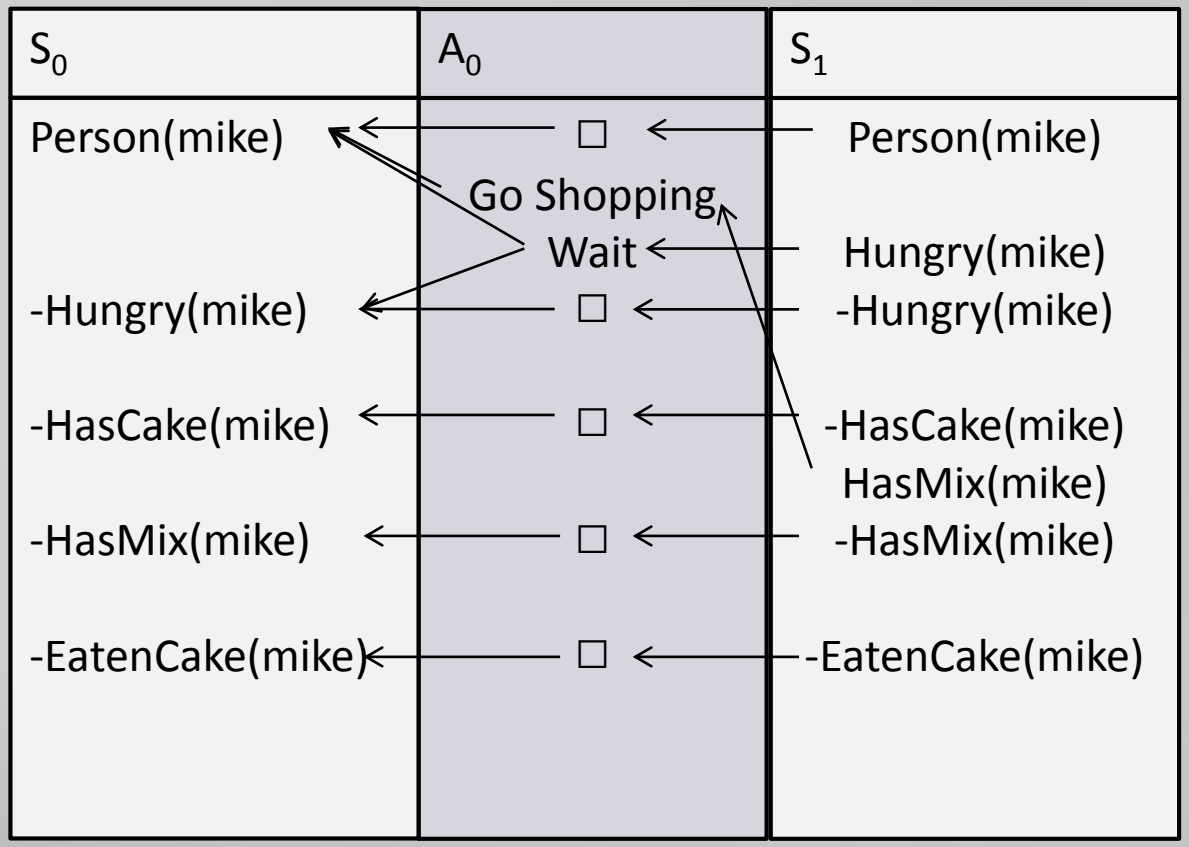
Planning Graph



Each state layer now has all literals that are the result of any action in the previous action later. Each state points to the actions it is an effect of.

Trivial actions ensure that the state layers are monotonically increasing (we never loose states).

Planning Graph

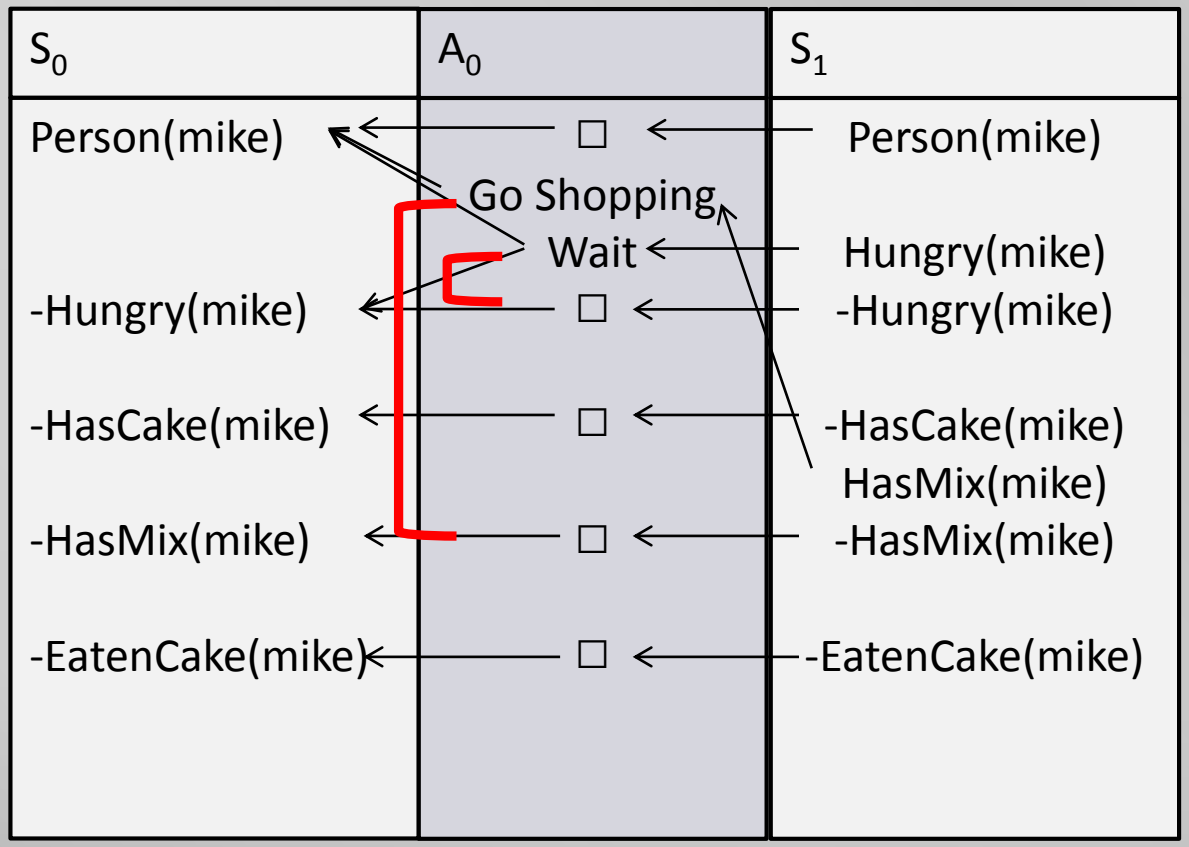


And we enter mutex (mutual exclusion) relations.

Two actions are recorded as incompatible if:

- one action's effect negates an effect of the other.
- one action's effect is the negation of a precondition of the other.
- a precondition for one action is marked as exclusive to a precondition for the other.

Planning Graph

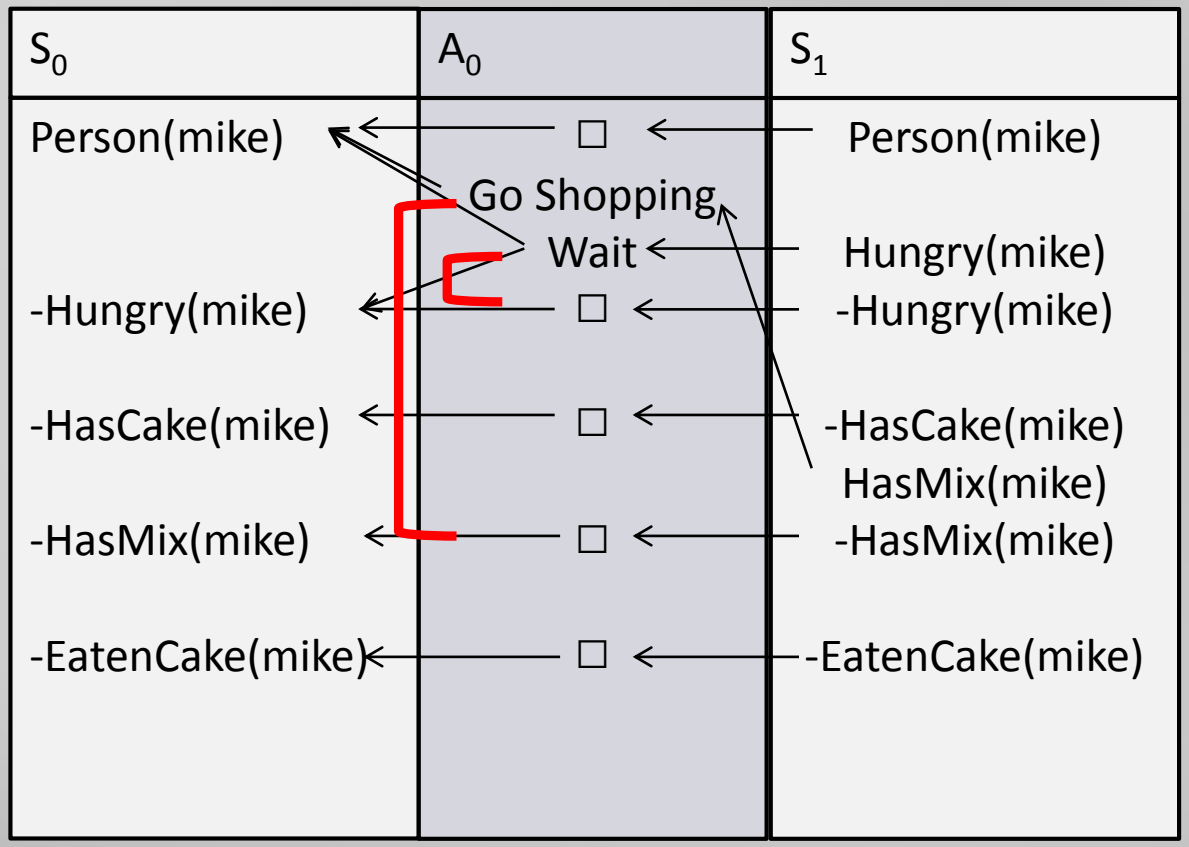


And we enter mutex (mutual exclusion) relations.

Two actions are recorded as incompatible if:

- **one action's effect negates an effect of the other.**
- one action's effect is the negation of a precondition of the other.
- a precondition for one action is marked as exclusive to a precondition for the other.

Planning Graph

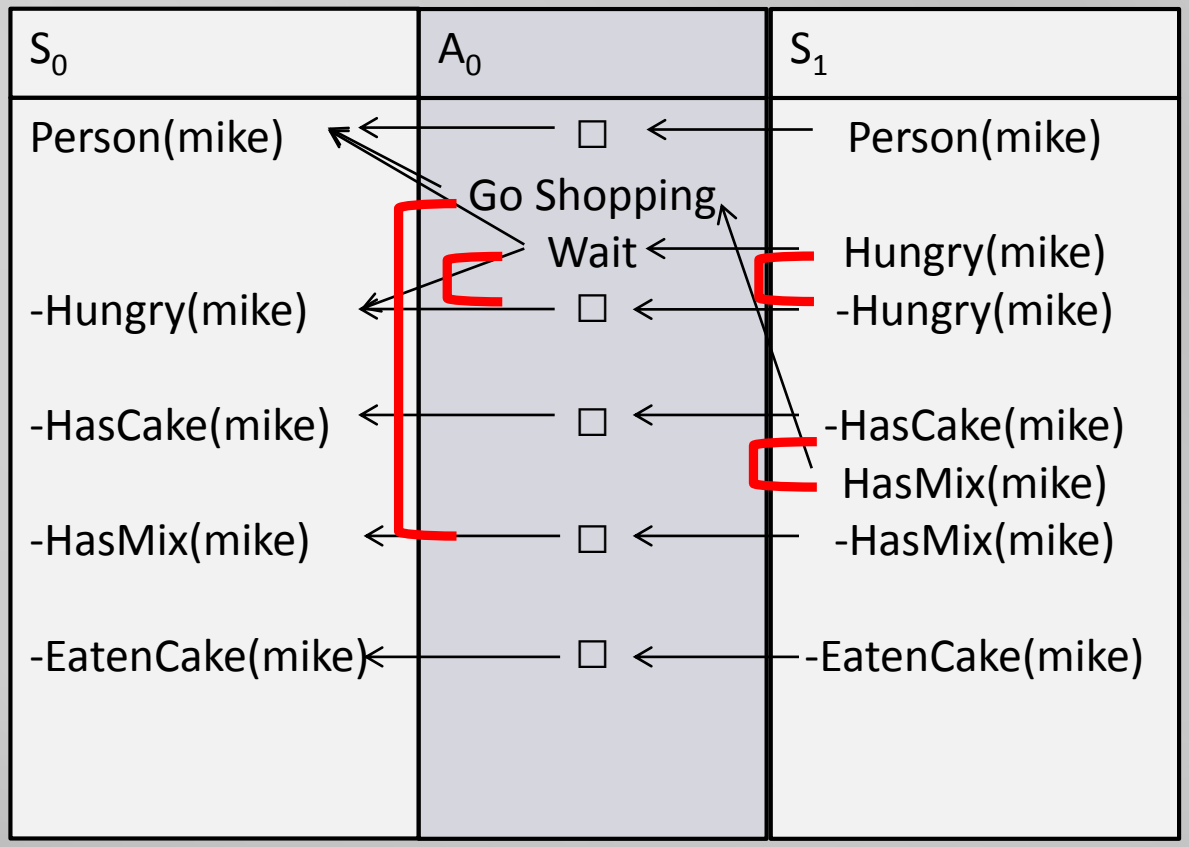


And we enter mutex (mutual exclusion) relations.

Two states are recorded as incompatible if:

- One is the negation of the other.
- If each possible pair of actions that could produce the literals are mutually exclusive.

Planning Graph

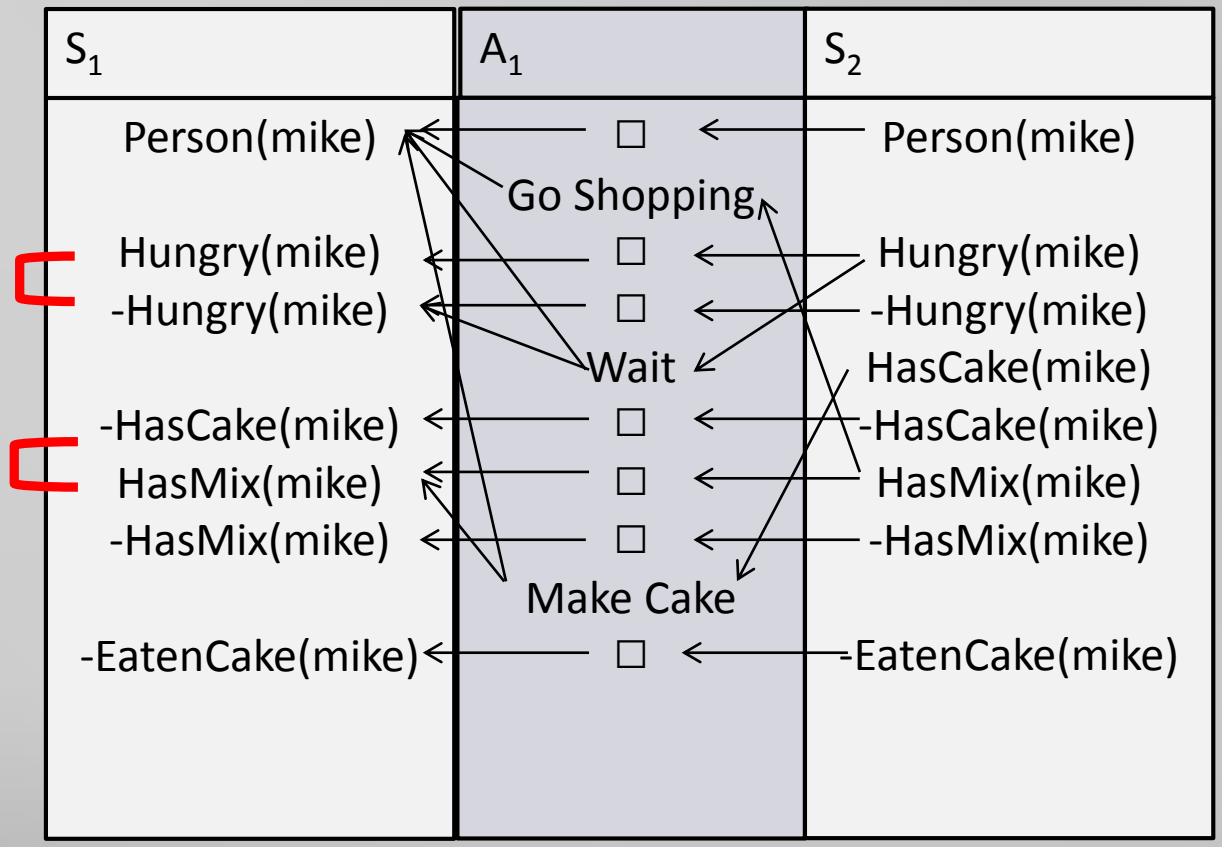


And we enter mutex (mutual exclusion) relations.

Two states are recorded as incompatible if:

- **One is the negation of the other.**
- If each possible pair of actions that could produce the literals are mutually exclusive.

Planning Graph

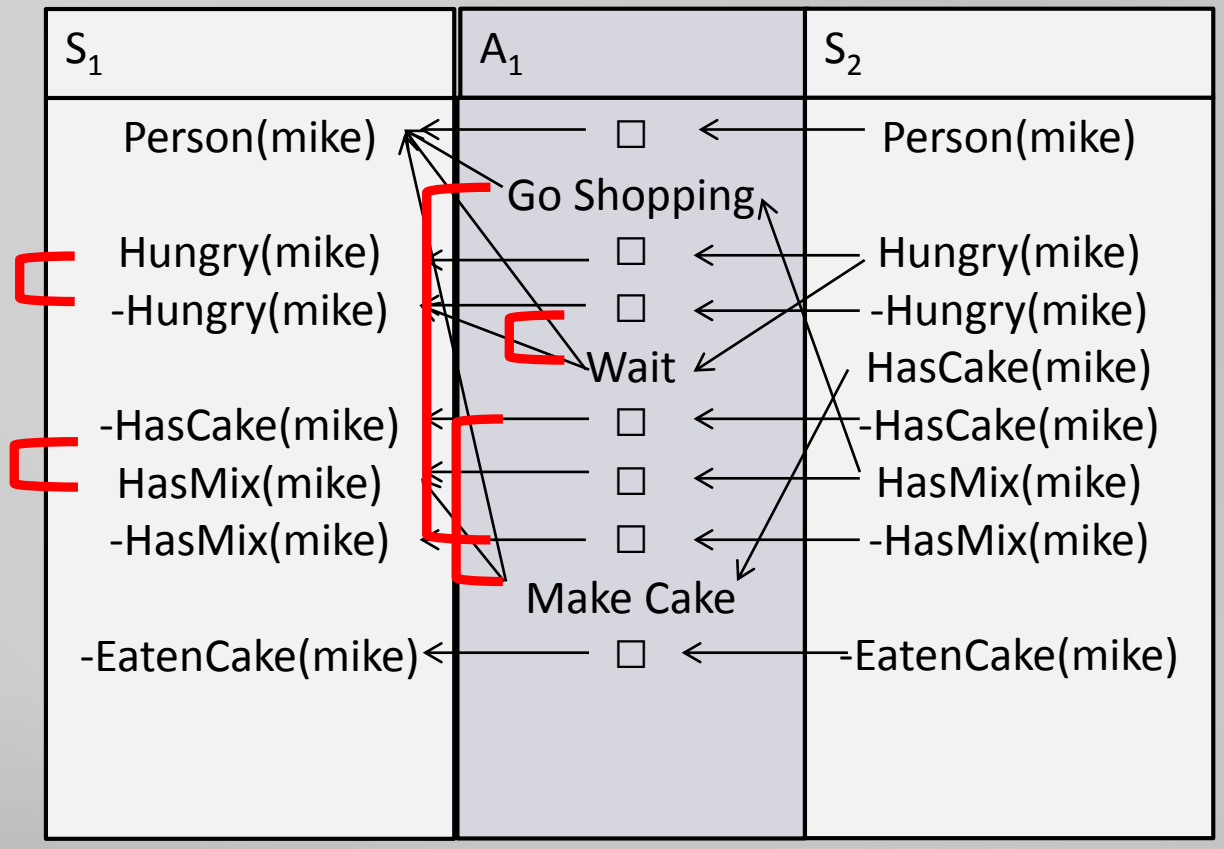


And we proceed.

Note we can create the graph iteratively...

And at each stage the number of actions/literals increases...

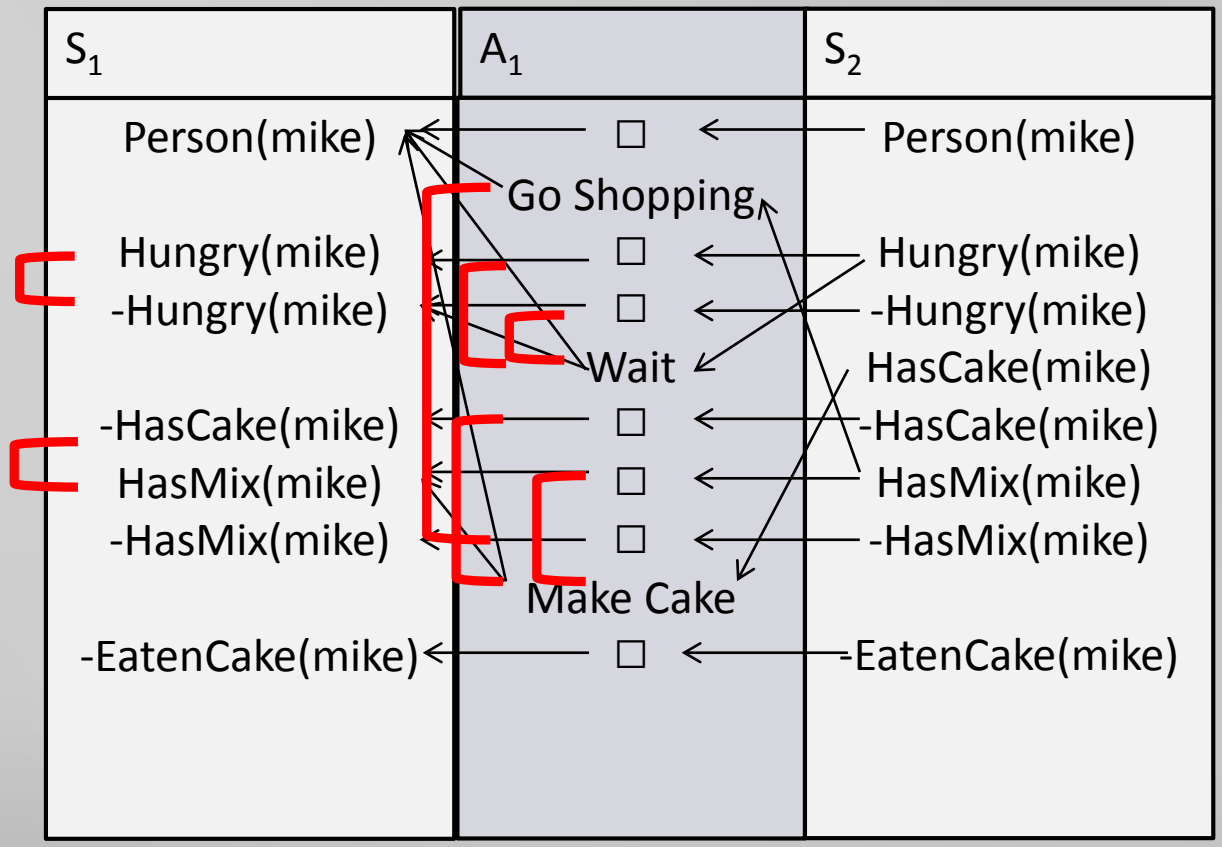
Planning Graph



Two actions are recorded as incompatible if:

- **one action's effect negates an effect of the other.**
- one action's effect is the negation of a precondition of the other.
- a precondition for one action is marked as exclusive to a precondition for the other.

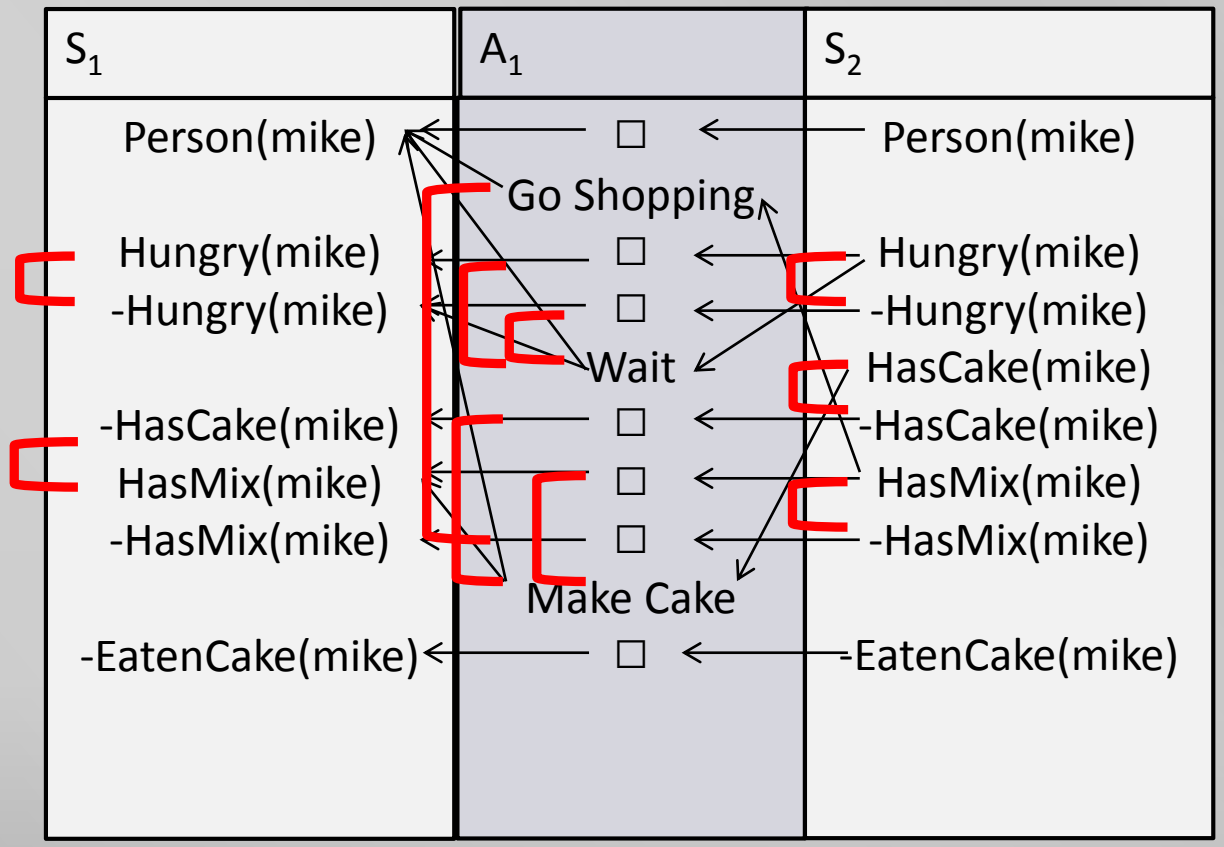
Planning Graph



Two actions are recorded as incompatible if:

- one action's effect negates an effect of the other.
- one action's effect is the negation of a precondition of the other.
- **a precondition for one action is marked as exclusive to a precondition for the other.**

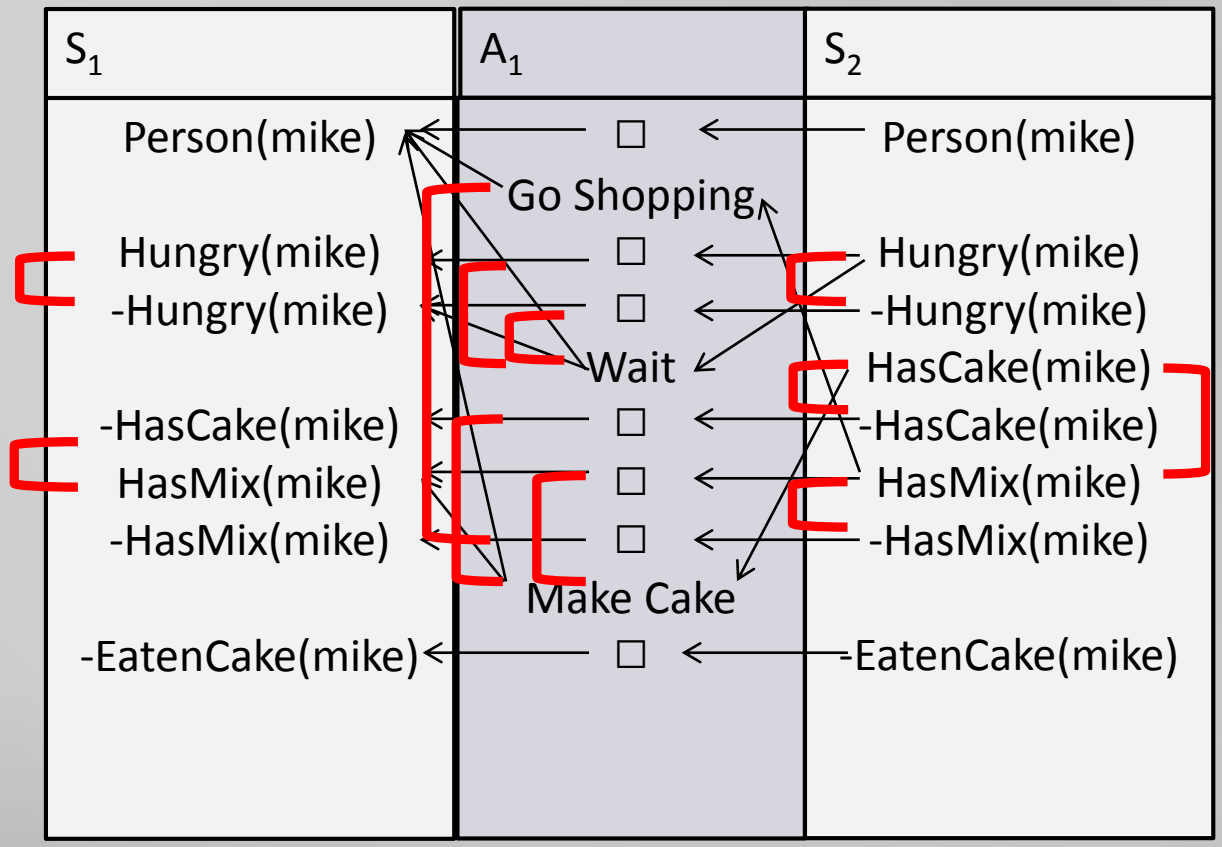
Planning Graph



Two states are recorded as incompatible if:

- **One is the negation of the other.**
- If each possible pair of actions that could produce the literals are mutually exclusive.

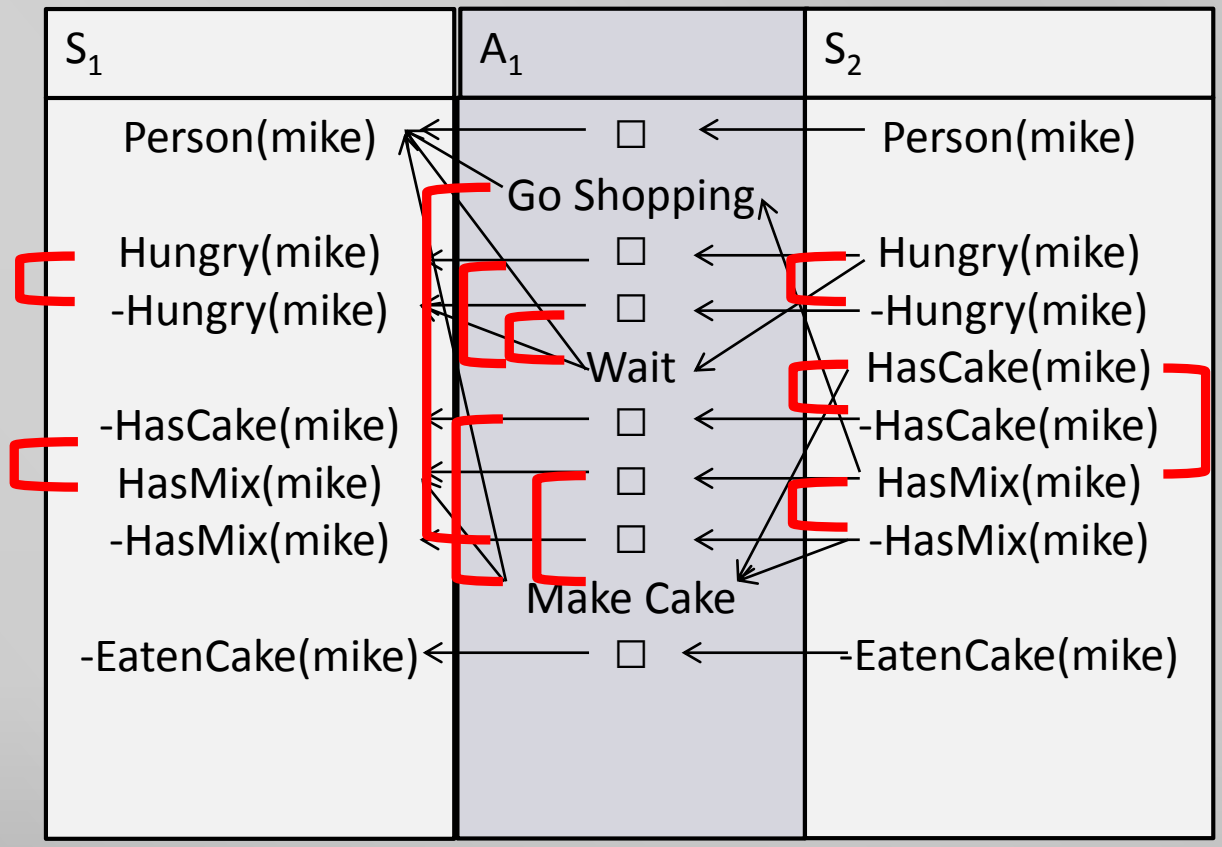
Planning Graph



Two states are recorded as incompatible if:

- One is the negation of the other.
- **If each possible pair of actions that could produce the literals are mutually exclusive.**

Planning Graph



We continue in this iterative fashion until two consecutive levels are identical.

At this point we say the graph has 'leveled off' and terminate.

What use are Planning Graphs?

1. If any goal literal fails to appear in the final state level, then the problem is unsolvable.
2. The number of levels between a state and the presence of all goal literals can be used to give an admissable heuristic (for, e.g. A*).
    1. The max-level heuristic given the number of levels until all goal literals are in the graph.
    2. The set-level heuristic given the number of levels until all goal literals are in the graph AND there are no mutexs between any pair of goal literals.

    This is unsurprising: We have effectively relaxed constraint on the problem, which is a general way of obtaining heuristics.
3. We can obtain a plan directly of the Planning Graph using the Graph Planning Algorithm, but this is generally less efficient.

Graph Planning Algorithm

From current state repeat until termination:

1. While at least one goal literals is absent OR there exists at least one mutex between goal literals:

   Create the next action and state levels, A_c and S_{c+1} and then set current state to S_{c+1}.

2. Otherwise, run the GP-Termination algorithm to see if the algorithm terminated. If it does not, create the next action and state levels, A_c and S_{c+1}.

GP-Termination Algorithm

1. Examine the last level, S_i, of the planning graph and initialize a list of literals, L_i, to be satisfied from the goal state of the planning problem.
2. Transition to layer S_{i-1} through selecting any subset of the actions at layer A_{i-1} where:
    1. These actions are not mutually exclusive
    2. The preconditions of these actions are not mutually exclusive.
    3. These actions cause as effects the list attached to the state. (Remember there are trivial actions.)
3. Set L_{i-1} as the set of preconditions for actions involved in the transition.
4. If a state is found with a goal list that is a subset of the initial state, a solution path has been found and we termination with success.

    REQUIRES SEARCHING ALL SEQUENCES OF VALID SUBSETS OF ACTIONS AT EACH LAYER
5. If no such state exists, no solution is currently possible and we record which goals were unsatisfiable at the final step.
6. If $S_c = S_{c-1}$ AND the goals that were unsatisfiable at S_c are the same as those that were unsatisfiable at S_{c-1} then we terminate with failure.