COMP 4303:

Artificial Intelligence in Games

Hierarchical Finite State Machines

Jay Henderson (jayhend@mun.ca)

Problems with State Machines

On its own, a state machine is a powerful tool, but as the complexity of agent behavior increases, the state machine can grow uncontrollably

N states:

N x N possible Transitions

N can already be very big, N x N is even bigger

Problems with State Machines

Maintainability:

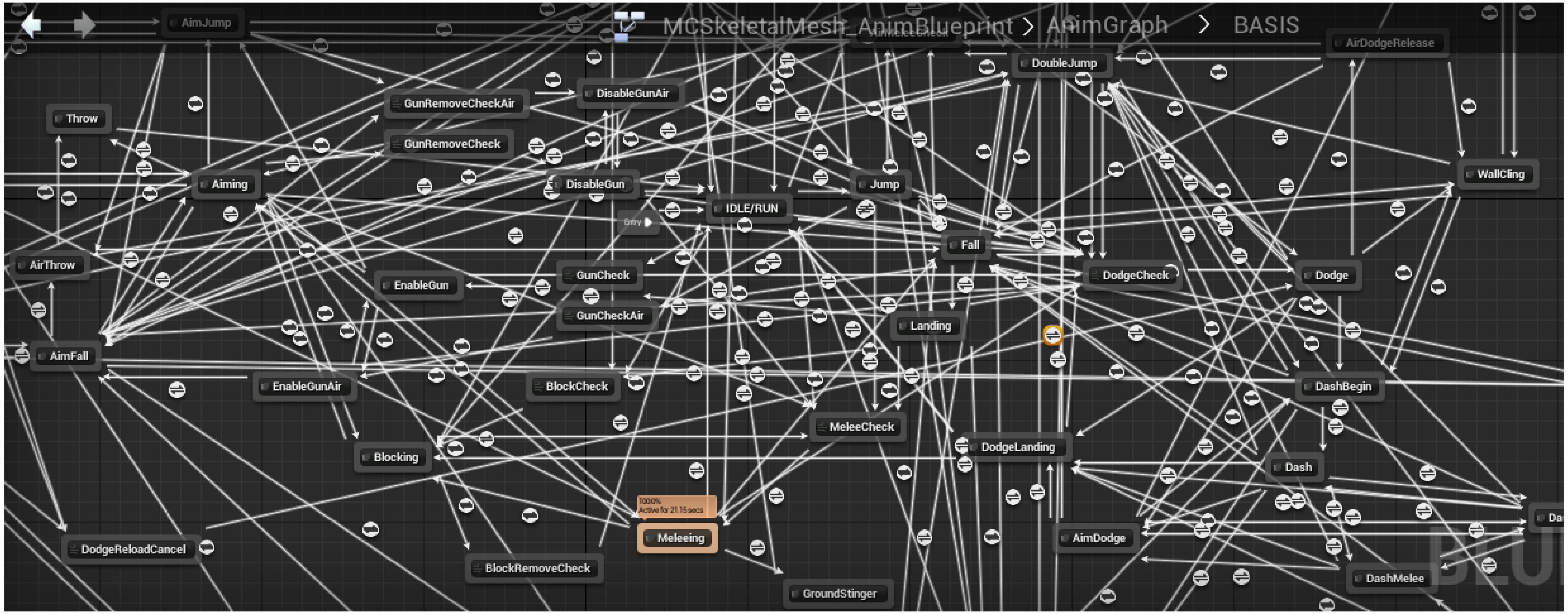
When adding or removing a state, it is necessary to change the conditions of all other states that have transition to the new or old one

Big changes are more susceptible to errors that may pass unnoticed

Reusability:

Conditions are inside the states, so coupling between states is strong, making it difficult to use the same behavior in multiple projects.

Problems with State Machines: Readability



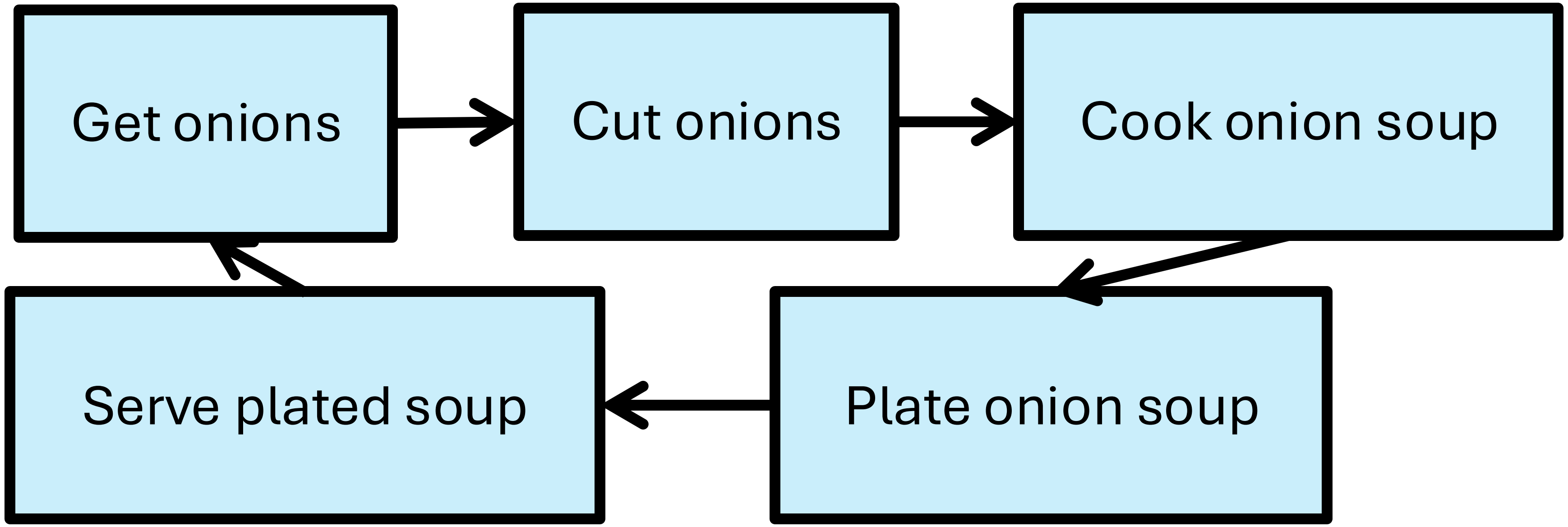
State Machine Spaghetti

Problems with State Machines

Difficult to express certain behaviours

For instance, a mechanism that requires an immediate state switch, followed by the character returning to the original state/behaviour afterward

Making Onion Soup in Overcooked



Get onions

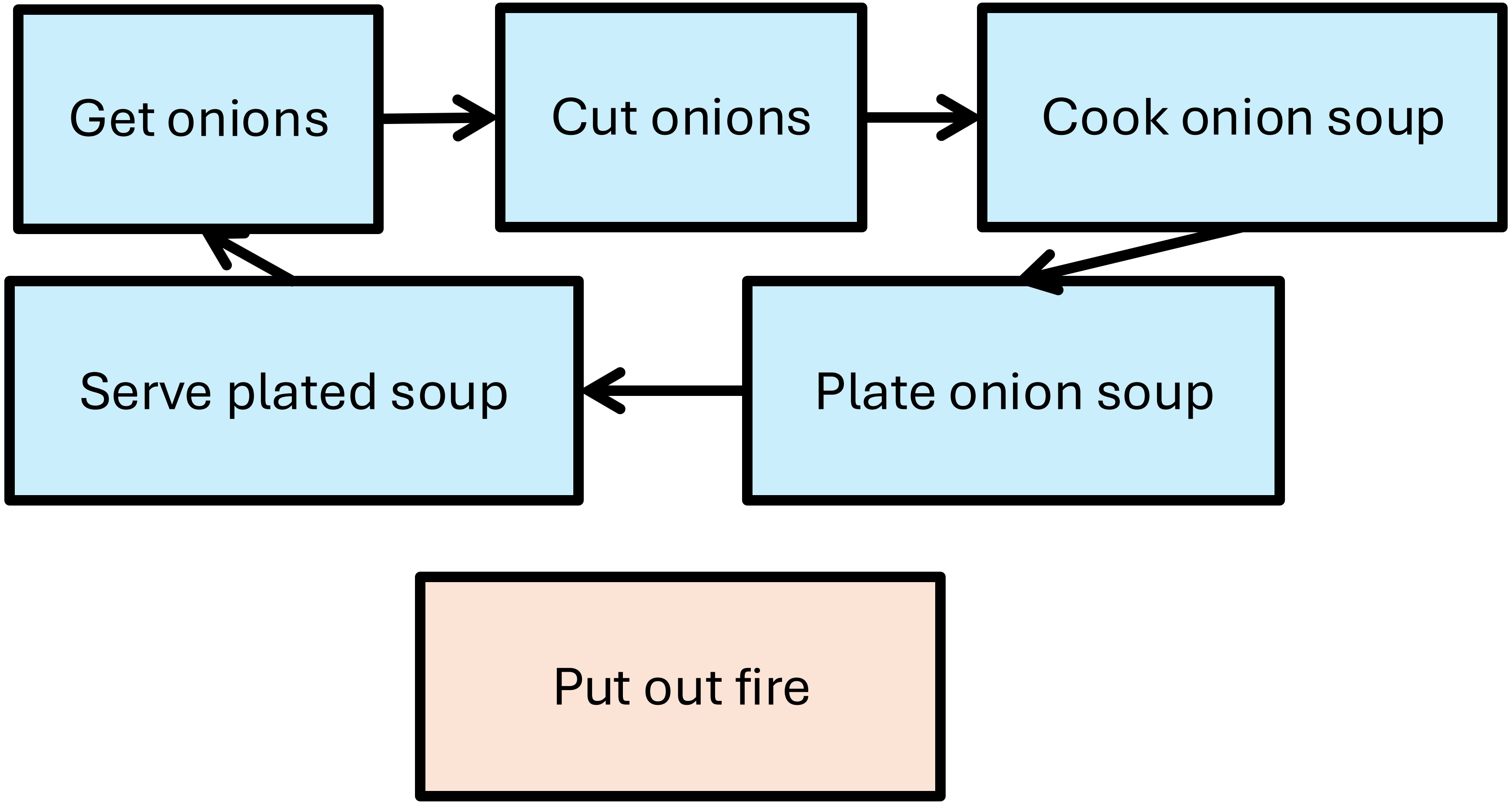
Cut onions

Cook onion soup

Plate onion soup

Serve plated soup

Making Onion Soup in Overcooked



Get onions

Cut onions

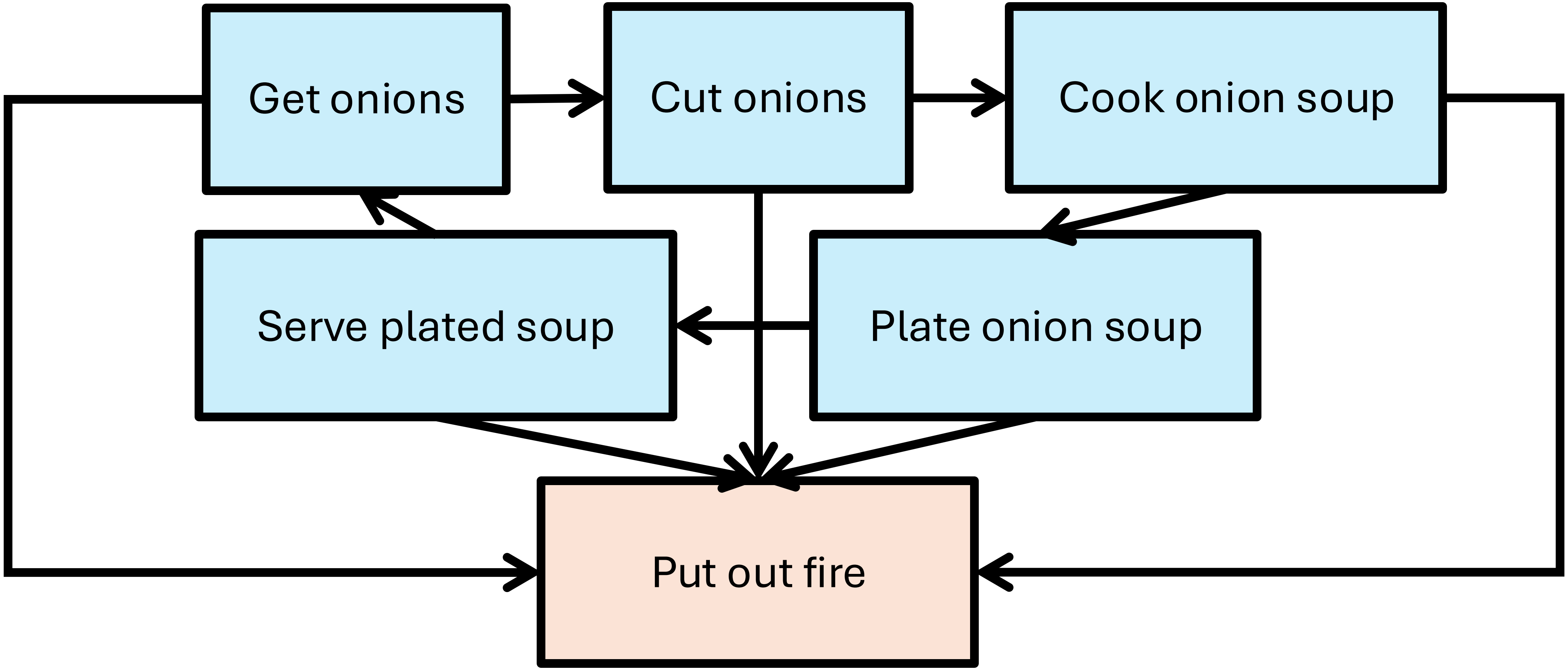
Cook onion soup

Plate onion soup

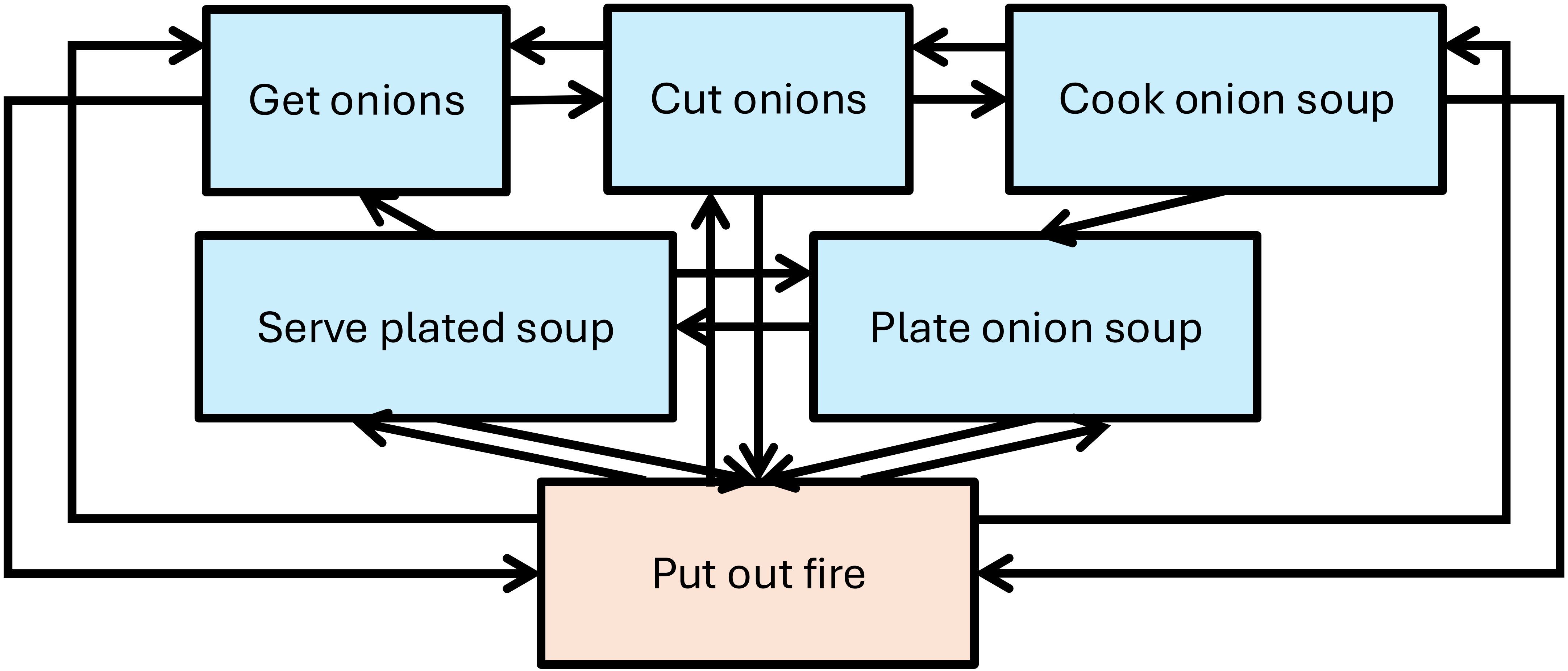
Serve plated soup

Put out fire

Making Onion Soup in Overcooked



Making Onion Soup in Overcooked



Hierarchical Finite State Machines

Hierarchical Finite State Machines

Also known as “StateCharts” (David Harel)
https://www.inf.ed.ac.uk/teaching/courses/seoc/2005_2006/resources/statecharts.pdf

Rather than combining all the logic into a single state machine,
we can separate it into several

In a hierarchical state machine, each state can be a complete state machine

Can use recursive algorithms to process the whole hierarchy

States

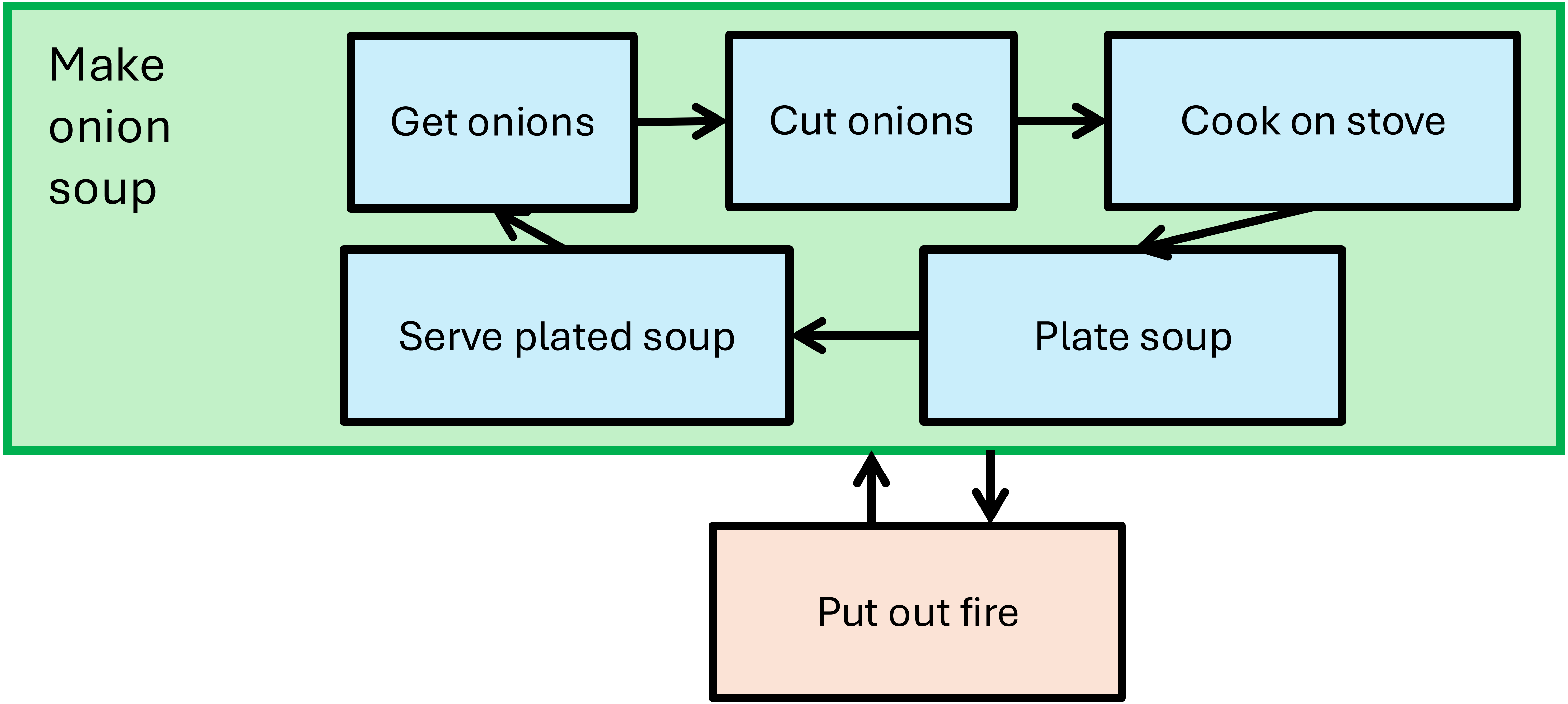
Superstates: groups (clusters) of states

Allows you to prevent redundant transitions by applying them only once to the Superstates, rather than each state individually

Generalized transitions: transitions between Superstates

Substate: our original states

HFSM Example (Overcooked)



HFSM

Each state will hold reference to a **superstate** and a **substate**

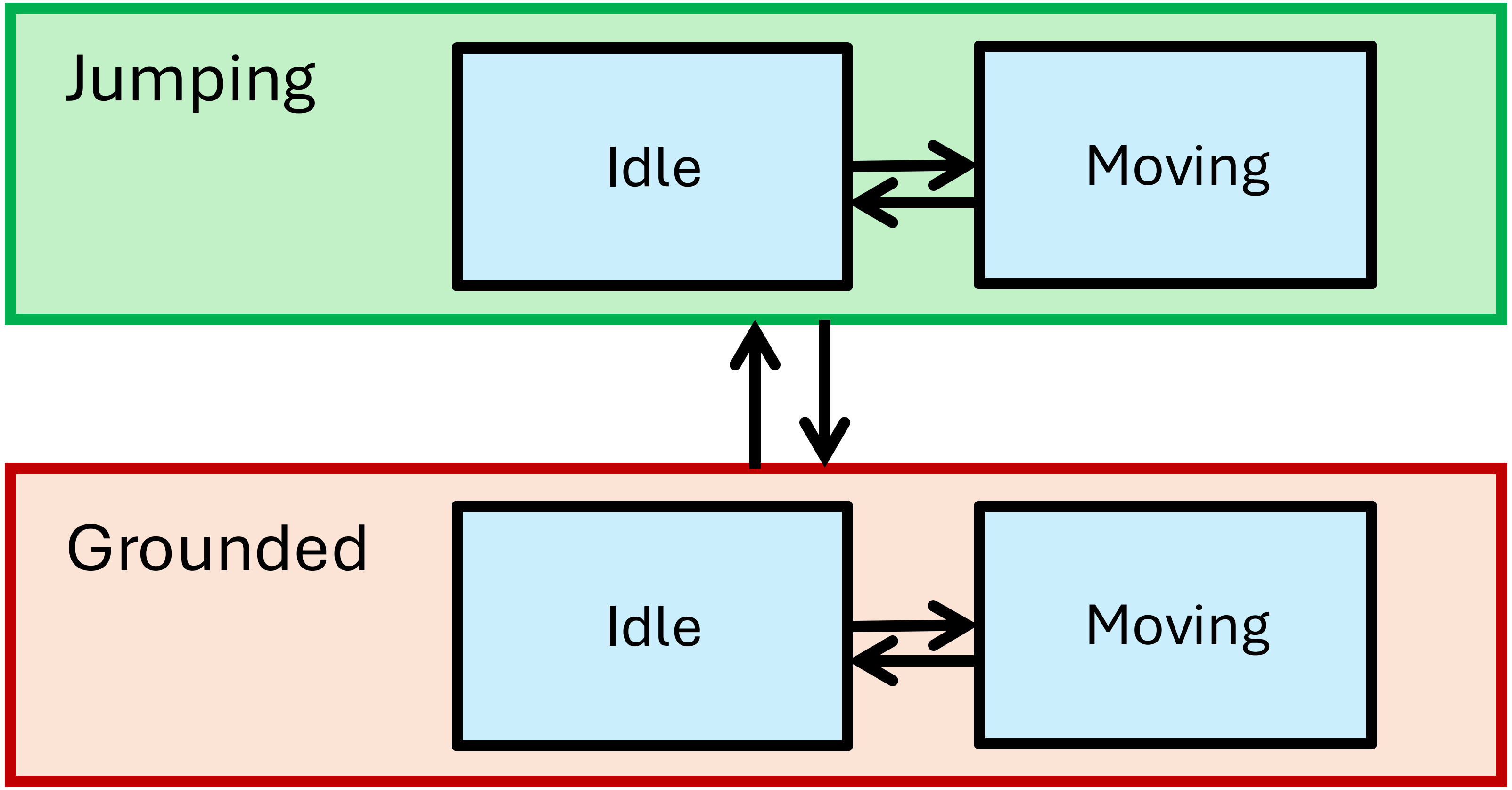
Our **updateState** method will recursively call **updateState** on each substate

Instead of switching states in our Character/NPC/Player,
the **state switching** logic will be in our state itself

When we change state,
we will also change reference to our superstate and substate

HFSM Example (e.g. Jumping)

Note: in this example we want to preserve substates across superstates



Adding a controller jump

Add a new jump boolean to our Controller class

Set jump if a spacebar event is triggered:

keydown => jump = true

keyup => jump = false

Add a jumping() method that returns whether jump is true or false

HierarchicalState class

Instance variables:

substate

superstate

Implement State class methods:

enterState – calls enterState on substate

updateState – cals updateState on substate

HierarchicalState class cont.

New methods for HierarchicalState:

setSubstate(newState) – sets the substate and the superstate of the substate

switchState(player, newState) – switches the current state of the player while maintaining all references to superstates and substates

2 cases:

- If the state does not have a superstate, we are switching the top level state
- If the state has a superstate, we switch the substate of the superstate then enter that substate

Implemented States (IdleState, MovingState)

Instead of switching state via the player,

we will now switch state via the state

e.g. `this.switchState(player, new IdleState());`

In our enterState and updateState,

we will call enterState and updateState from our superclass (HierarchicalState) to recursively update all substates

e.g. `super.updateState(player, controller);`

Jumping State

When jumping:

We want to apply a very strong one time force on the y axis on our character

enterState()

- Apply the large force to our character
- Call super.enterState()

updateState()

- Check to see if we are back on the ground,
  if so we will switch to grounded state
- Call super.updateState()

Grounded State

When grounded:

We don't really need to do anything

enterState()

- Call super.enterState()

updateState()

- Check to see if our controller indicates we are jumping if so, we will switch to jumping state
- Call super.updateState()