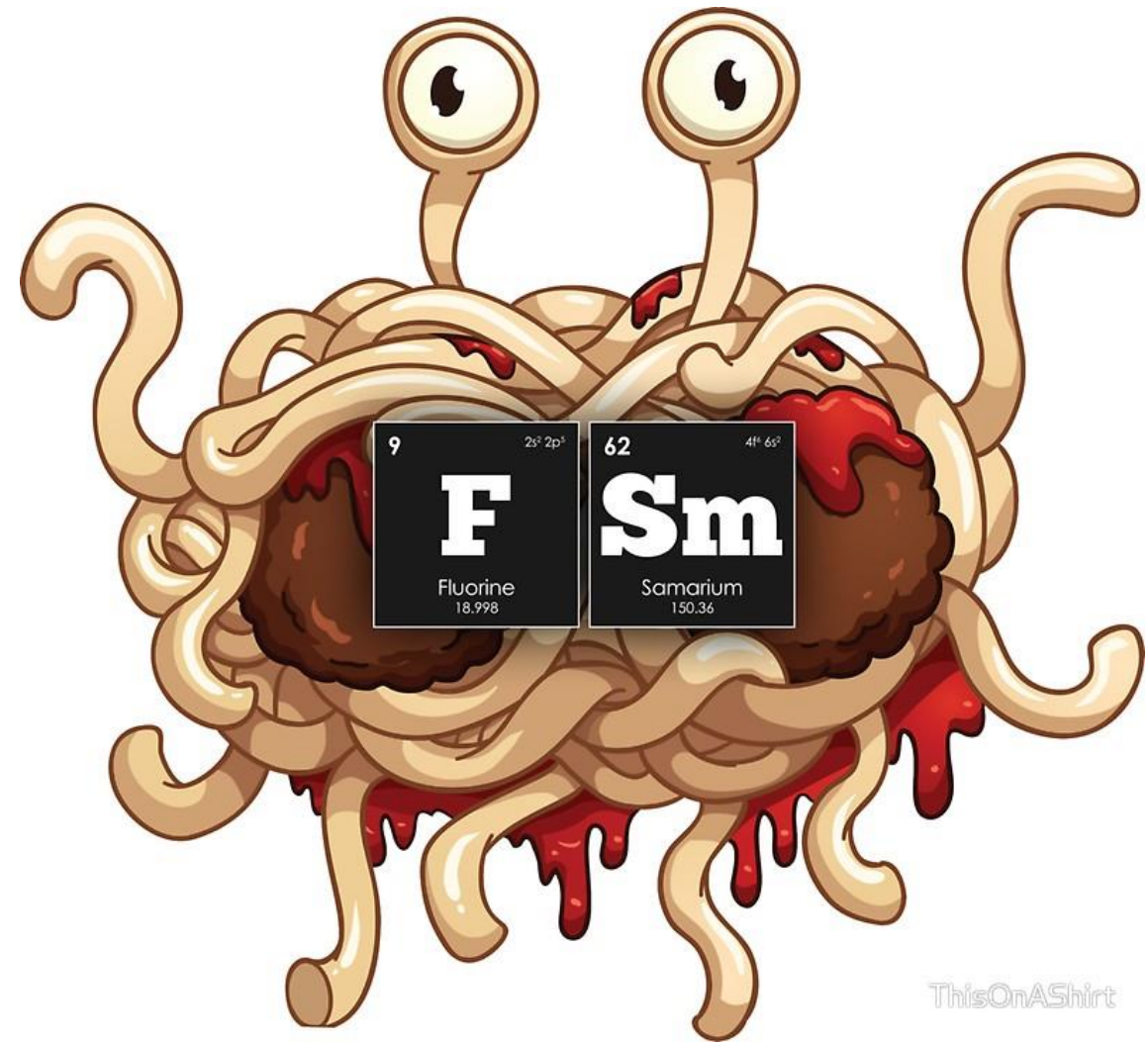


“Once you make a decision, the universe conspires to make it happen.” – Ralph Waldo Emerson



<https://www.redbubble.com/people/thisonashirt/works/28066542-chemistry-periodic-table-elements-fsm-flying-spaghetti-monster?p=sticker>

Announcements

- HW3. <http://osi.gatech.edu/>
- HW4 is posted, due this Sunday
 - Obfuscated solution to hw3

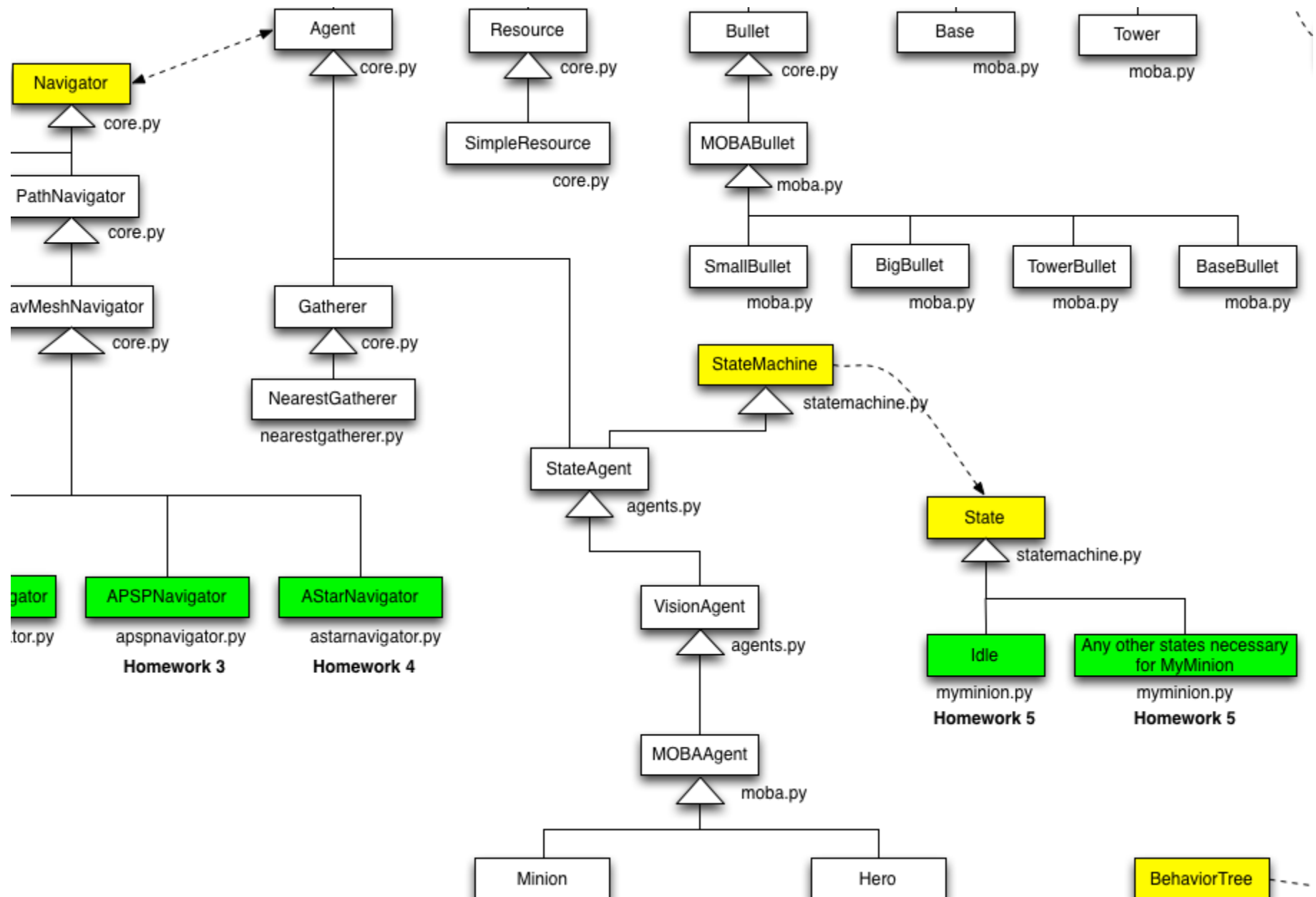


N-2: formations

1. What's the problem with independent shortest distance navigation with groups?
2. What's a simple solution, and its drawbacks? Other workarounds?
3. Is it always sensible to replan paths when problems occur?
4. What are 3 ways of implementing fixed formations?
5. Discuss some problems with entities of different sizes.

N-1: Decision Making, FSMs

1. How can we describe decision making?
2. What makes FSMs so attractive? What is difficult to do with them?
3. Two drawbacks of FSMs and how to fix?
4. What are the performance dimensions we tend to assess?
5. What are two methods we discussed to learn about changes in the world state?
6. FSMs/Btrees: _____ :: Planning : _____



<http://game-ai.gatech.edu/sites/default/files/documents/documentation/object-hierarchy.png>

<http://game-ai.gatech.edu/sites/default/files/documents/assignments/homework5.html>

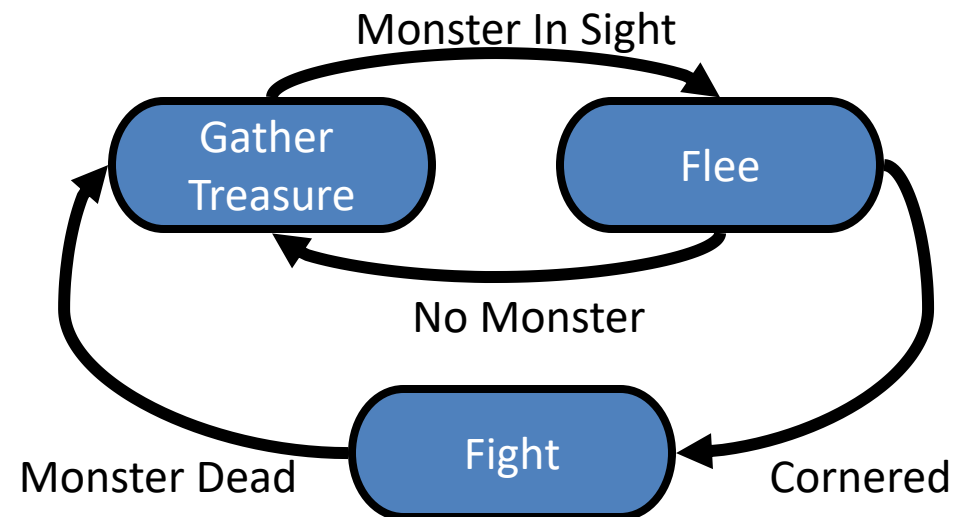
Decision Making – FSMs

2018-02-13

B Ch2, M 5.1,5.3

State Transition Table

Current State	Condition	State Transition
Gather Treasure	Monster	Flee
Flee	Cornered	Fight
Flee	No Monster	Gather Treasure
Fight	Monster Dead	Gather Treasure



FSM Pros & Cons

Pro

- Ubiquitous (not only in digital games)
- Quick and simple to code
- (can be) Easy* to debug
- Very fast: Small computational overhead
- Intuitive
- Flexible
- Easy for designers without coding knowledge
- Non-deterministic FSM can make behavior unpredictable

Con

- When it fails, fails hard:
 - A transition from one state to another requires forethought (get stuck in a state or can't do the "correct" next action)
- Number of states can grow fast
 - Exponentially with number of events in world (multiple ways to react to same event given other variables): $s=2^e$
- Number of transitions/arcs can grow even faster: $a=s^2$
- Doesn't work with sequences of actions/memory

Choices have consequences

FSM IMPLEMENTATIONS

Implementations

- Centralized conditionals
 - If / then statements
 - Throw it all in a switch statement
 - Simple, but not extendable
 - Macros
- Distributed
 - State as class; transition rules within
 - Agent carries reference to current state. Extendable
- State as table (central or distrib)
 - Can be stored separately, easier for designers

```
void RunLogic( int state ) {  
    switch( state ) {  
        case 0: //Wander  
            Wander();  
            if( SeeEnemy() )  
                state = 1;  
            if( Dead() )  
                state = 2;  
  
            break;  
        case 1: //Attack  
            Attack();  
            state = 0;  
            if( Dead() )  
                state = 2;  
  
            break;  
        case 2: //Dead  
            SlowlyRot();  
            break;  
    }  
}
```

Impl: Centralized Conditionals

- Simplest method
- After an action, the state might change.
- Requires a recompile for changes (hard-coded)
- No pluggable AI
- Not accessible to non-programmers
- No set structure
- Can be a bottleneck.

```
void RunLogic( int *state ) {  
    switch( *state ) {  
        case 0: //Wander  
            Wander();  
            if( SeeEnemy() )  
                *state = 1;  
            if( Dead() )  
                *state = 2;  
            break;  
        case 1: //Attack  
            Attack();  
            *state = 0;  
            if( Dead() )  
                *state = 2;  
            break;  
        case 3: //Dead  
            SlowlyRot()  
            break;  
    }  
}
```

... in Game Loop (w/ enum)

```
public enum State {STATE1, STATE2, STATE3};
State state = State.STATE1;
void tick ()
{
    switch (state) {
        case STATE1:
            PlayAnimation(...);
            if (...) state = newstate;
            else if (...) state = newstate;
            else if ...
            else ...
        case STATE2:
            PlayAnimation(...);
            if (...) state = newstate;
            else if...
            else if...
            else ...
    }
}
```

Implementation: Macros

```
...
BeginInitMachine
    State(WANDER)
        Begin:
            Wander();
            if (SeeEnemy()) GotoState(ATTACK);
            if (Incapacitated()) GotoState(INCAPACITATED);
    State(INCAPACITATED)
        Begin:
            ...
        Moan:
            PlaySound(moan);
            goto 'Moan';
EndInitMachine
```

Impl: State Transition Tables

Current State	Condition	State Transition
RunAway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened && StrongerThanEnemy	Attack
Patrol	Threatened && WeakerThanEnemy	RunAway

If Kitty_Hungry AND NOT Kitty_Playful SWITCH_CARTRIDGE eat_fish

Impl: Tables Alt

Event → State ↓	E1	E2	E3
S1	----	A1/S2	A3/S1
S2
S3

S: state, E: event, A: action, ----: illegal transition

Impl: State Transition Tables Alt Alt

Current State	Condition	State Transition	Action
RunAway	Safe	Patrol	
Attack	WeakerThanEnemy	RunAway	
Patrol	Threatened && StrongerThanEnemy	Attack	
Patrol	Threatened && WeakerThanEnemy	RunAway	

Implementation: Virtual FSM

State Name	Conditions	Actions
Current state name	Entry	Outputs...
	Exit	Outputs...
	Condition 1...	Outputs...
	Condition 2...	Outputs...
Next state name	Condition X	Outputs...
Next state name	Condition Y	Outputs...
...

https://en.wikipedia.org/wiki/Virtual_finite-state_machine

Implementation: Virtual FSM

State Name	Conditions	Actions
Patrol	Entry	SwingKeys()
	Exit	DropClipboard()
	Happy()	Whistle()
	NearDog()	PetDog()
Flee	Overwhelmed()	Scream()
Attack	EnemyNear()	TakeOutGun()
...

Impl: Distributed State Design Pattern

- Rules for transition contained within the states themselves
- Good encapsulation
- Can swap in/out states easier
- AKA
 - “State Design Pattern” (Buckland italics)
 - “Embedded rules” (Buckland subheading)

Eat_fish cartridge knows when to switch to Use_litterbox

Impl: Distributed

```
interface Entity
{
    void update () ; ← Where “thinking” happens.

    //void changeState (State newstate);
}
```

```
interface State
{
    void execute (Entity thing);

    void onEnter (Entity thing);

    void onExit (Entity thing);
}
```

Impl: Distributed

```
class Troll implements Entity
{ int liveTime=0;
  State currentstate, previousState;
  @Override
  void update () {
    liveTime++;
    currentstate.execute( this );
  }
  //@Override
  void changeState (State newstate) {
    previousState = currentState;
    currentstate.onExit( this );
    currentstate = newstate;
    currentState.onEnter( this );
  }
}
```

```
Class CoolState implements State
{ @Override
  void execute (Entity thing) {}
  void execute (Troll thing) {
    if ( thing.liveTime = 0 ) {
      thing.playAnimation(ani1);
      thing.changeState(new st);
    }
    else thing.doSomething();
  }

  @Override
  void onEnter (Entity thing) {...}
  @Override
  void onExit (Entity thing) {...}
}
```

See Buckland for further details (e.g. singleton use)

Impl: Consolidated, Distributed

```
class StateMachine //implements Entity?
{ State currSt, prevSt, globalSt;
  Entity owner;

  StateMachine( Entity e ){ owner = e; }

  void update () {
    if( globalSt != null)
      globalSt.execute( owner);
    currentstate.execute( owner );
  }
  void changeState (State newstate) {
    previousState = currentState;
    currentstate.onExit( owner);
    currentstate = newstate;
    currentState.onEnter( owner );
  }
  void revertToPrev(){ changeState( prevSt ); }
  boolean isInState( State st ) { ...}
}
```

```
class Troll implements Entity
{ StateMachine fsm;
  Troll(){
    fsm = new StateMachine( this );
    fsm.setGlobalState(
      TrollGlobalState.singleton() );
    fsm.setLocalState(
      TrollSleepInCave.singleton() );
  }

  void update(){
    liveTime++;
    fsm.update();
  }

  StateMachine getFSM(){ return fsm; }
}
```

Global States

- May have multiple states that could happen at any time
- Want to avoid authoring many transitions from every other state to these
- Create a global state that is called every update cycle
- State “blips” (return to previous after global)

Impl: Python-like

class StateMachine:

states #list of states

initST

curST = initST

def update():

 triggeredT = None

 for t in curST.transitions():

 if t.isTriggered():

 triggeredT = t

 break

 if triggeredT:

 targetST = triggeredT.getTargetState()

 actions = curST.getExitAction()

 actions += triggeredT.getAction()

 actions += targetST.getEntryAction()

 curST = targetST

 return actions

 else: return curST.getAction()

class State:

actions

def getAction(): return actions

entryActs

def getEntryAction(): return entryActs

exitActs

def getExitAction(): return exitActs

transitions

def getTransitions(): return transitions

class Transition:

condition

def isTriggered(): return condition.test()

targetState

def getTargetState(): return targetState

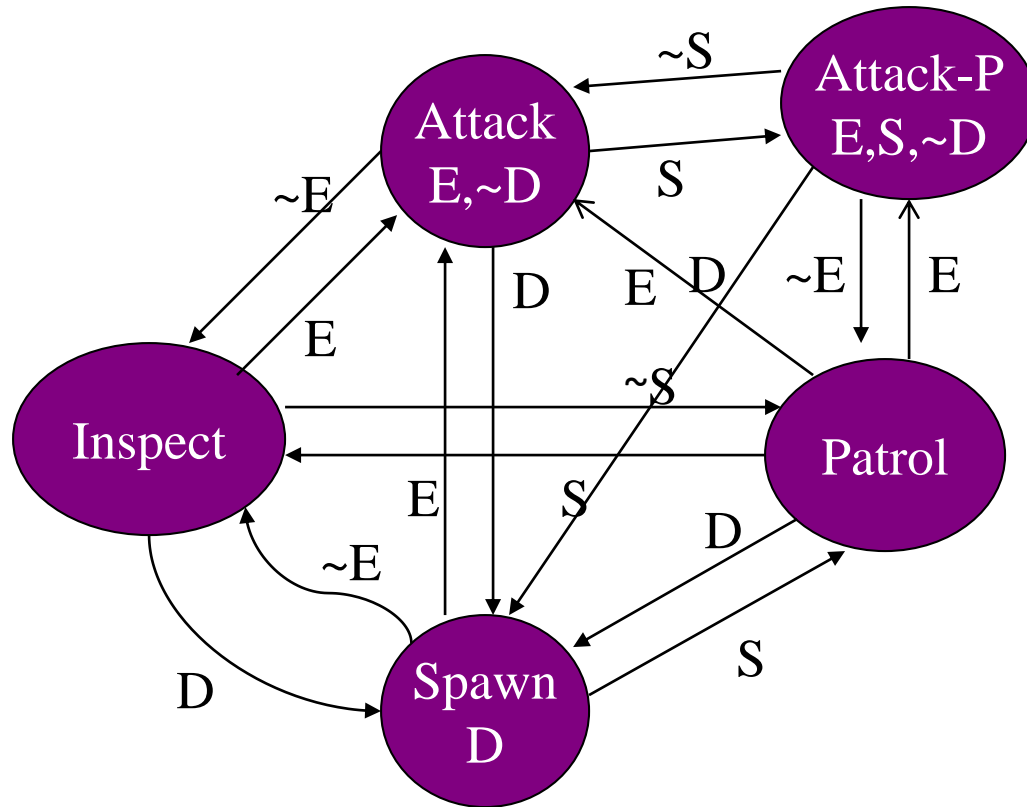
actions

def getAction(): return actions

FSM Extensions

- Extending States
 - Adding `onEnter()` and `onExit()` states can help handle state changes gracefully.
- Stack Based FSM's
 - Push new state onto stack, when it's done pop stack for next state
 - Allows an AI to switch states, then return to a previous state.
 - Gives the AI 'memory'
 - More realistic behavior
 - Subtype: Hierarchical FSM's

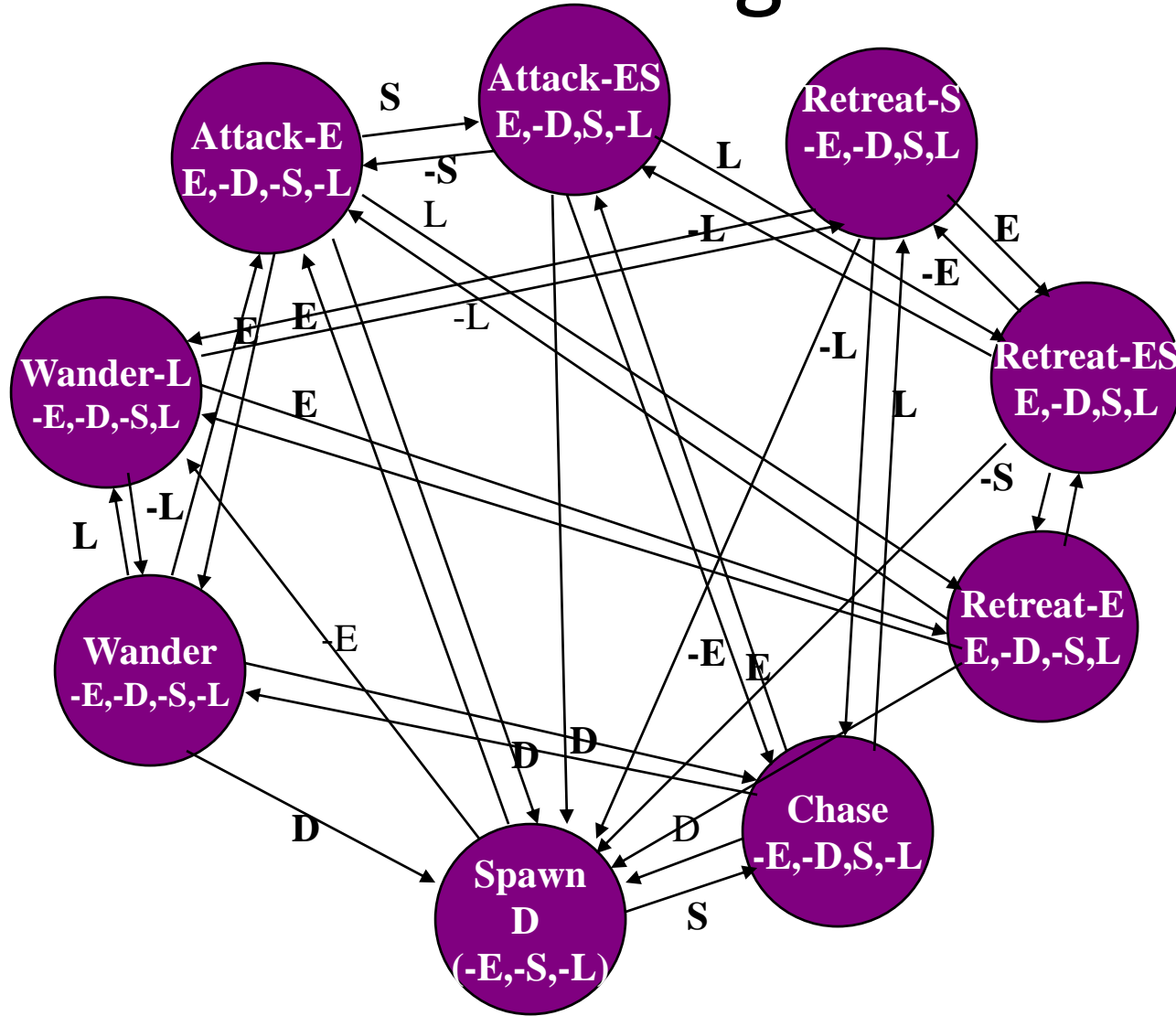
Motivating FSM Stacks



- Original version doesn't remember what the previous state was.
- One solution is to add another state to remember if you heard a sound before attacking.

E: Enemy in sight; S: hear a sound; D: dead

Motivating FSM Stacks (2)



Worst case:
Each extra state
variable can add 2^n
extra states
 n = number of
existing states

Using a stack would
allow much of this
behavior without the
extra states.

E: Enemy in sight; S: hear a sound; D: dead

Stack FSM – Thief 3



Stack allows AI to move back and forth between states.

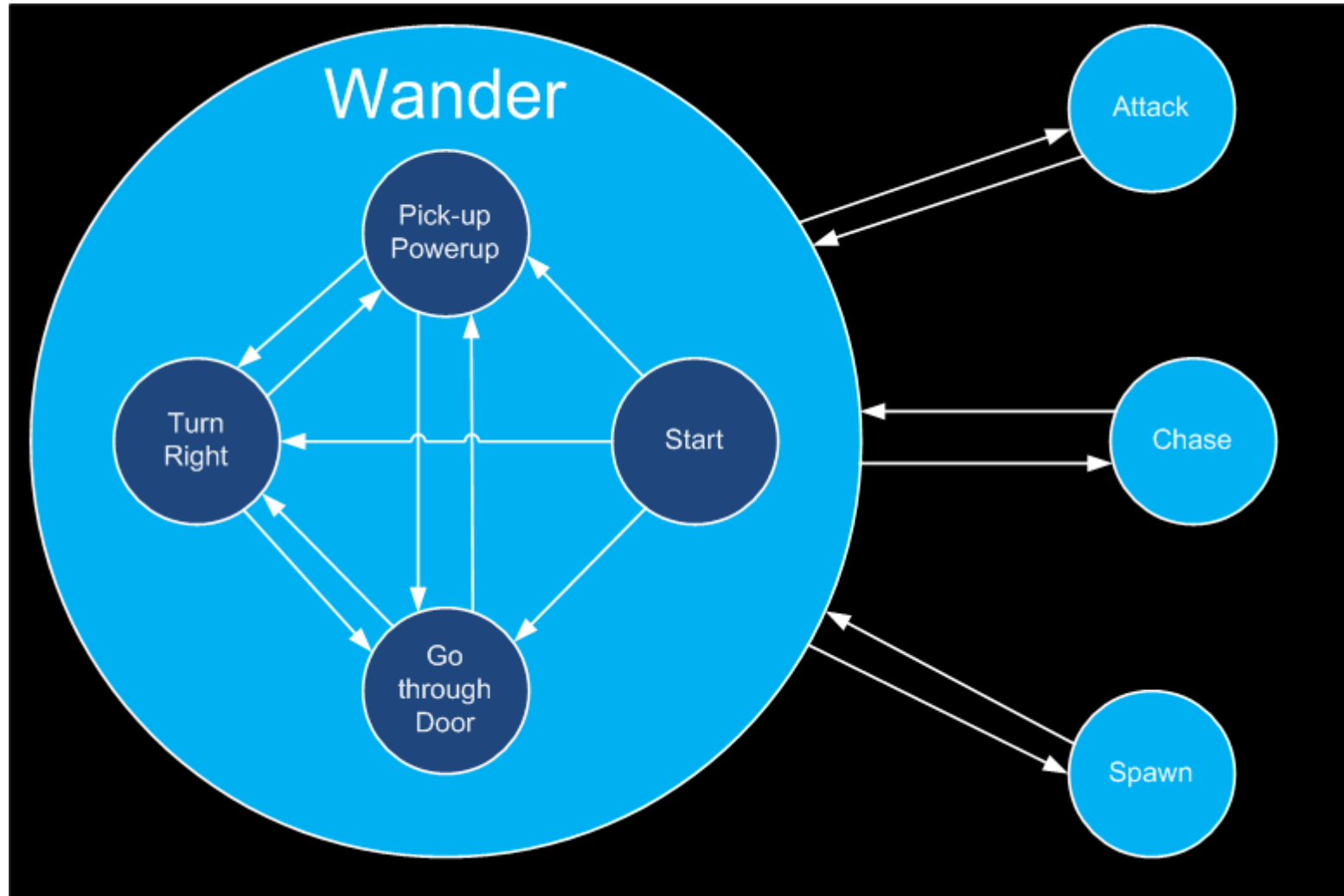


Leads to more realistic behavior without increasing FSM complexity.

Hierarchical FSMs

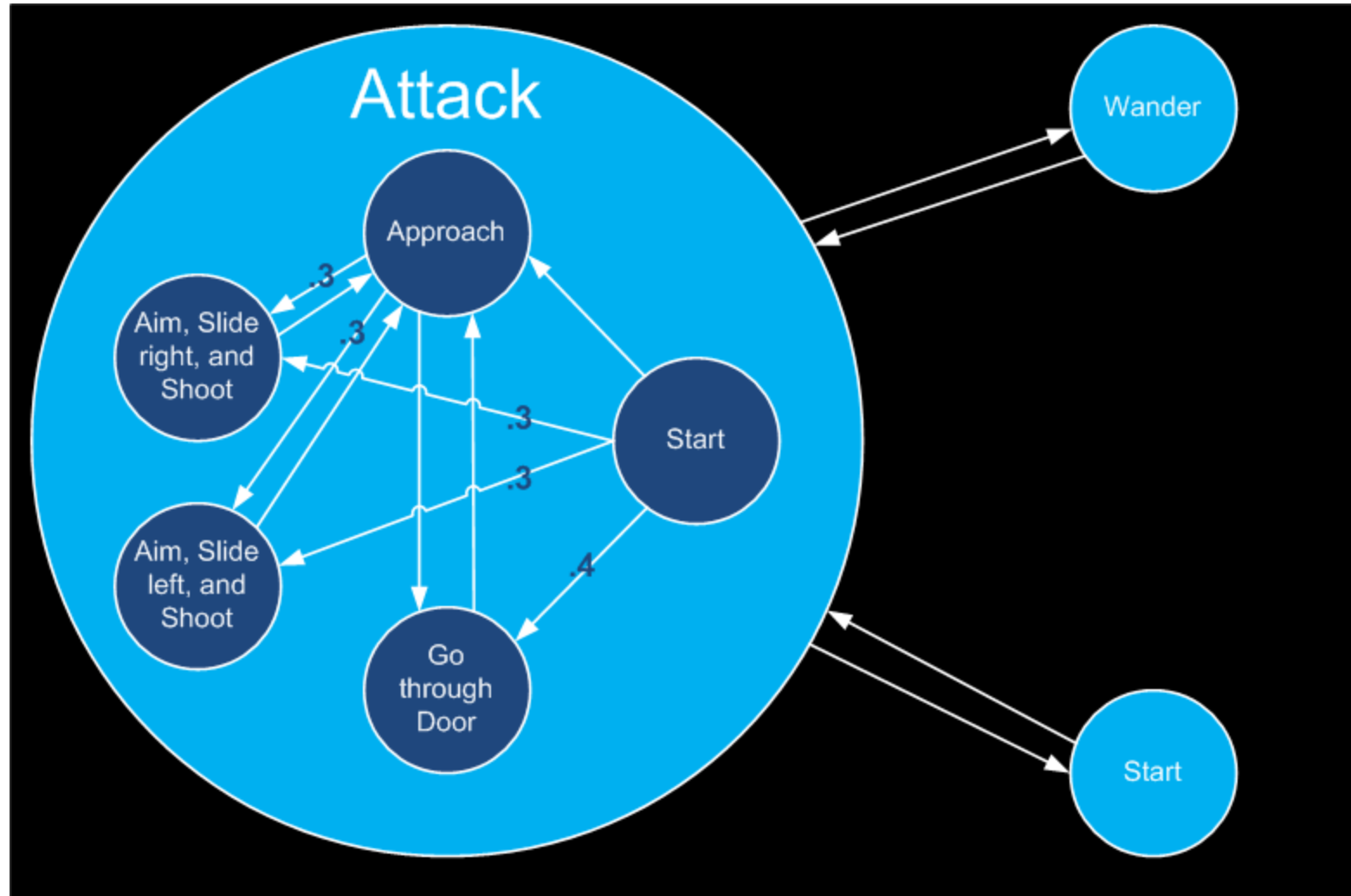
- Expand a state into its own sub-FSM
- Some events move you around the same level in the hierarchy, some move you up a level
- When entering a state, have to choose a state for it's child in the hierarchy
 - Set a default, and always go to that
 - Random choice
 - Depends on the nature of the behavior

Hierarchical FSM Example



E: Enemy in sight; S: hear a sound; D: dead

Non-Deterministic Hierarchical FSM



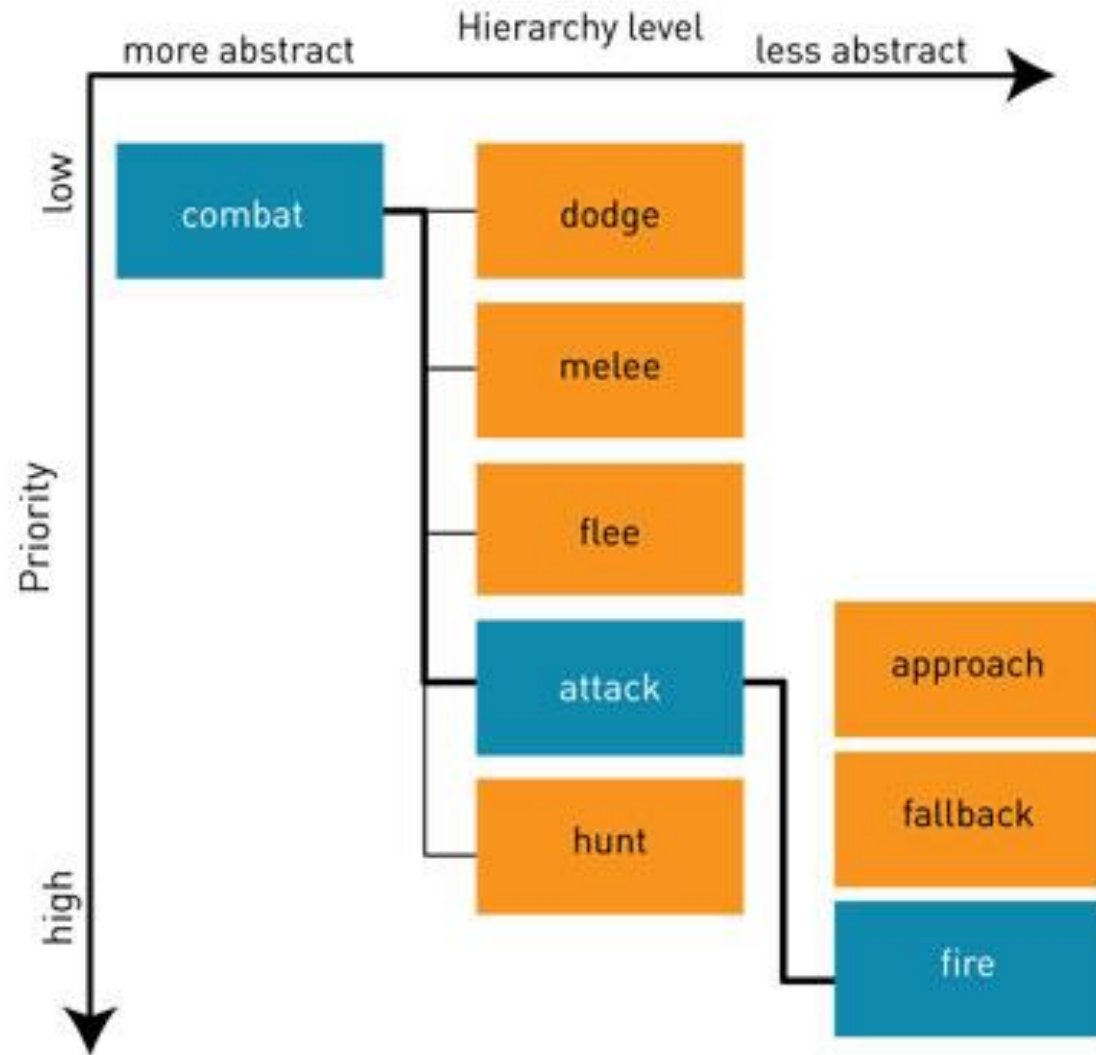
Hierarchical FSMs in *Destroy All Humans 2*



http://www.gamasutra.com/view/feature/130279/creating_all_humans_a_datadriven_.php

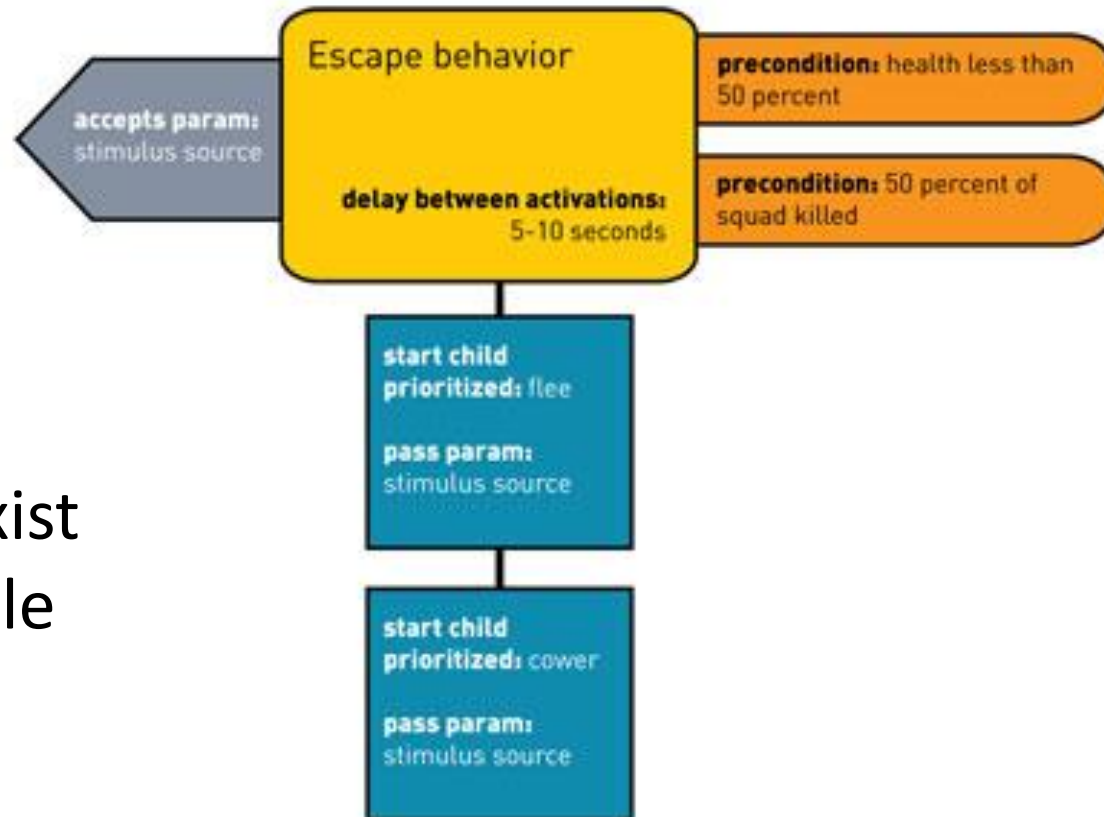
Hierarchical FSMs in *Destroy All Humans 2*

- Active (blue), pending (orange)
- Only active behaviors update
- Only active behaviors have children
- If * children startable, rank
- States can be marked as non-interruptable or non-blocking



Hierarchical FSMs in *Destroy All Humans 2*

- Self-contained behaviors
 - When to activate
 - What activates it, interrupts it
 - What to do on start, exit
 - What children it starts
- Code-supported behaviors exist for complex, non-generalizable cases



Hierarchical FSMs in *Destroy All Humans 2*

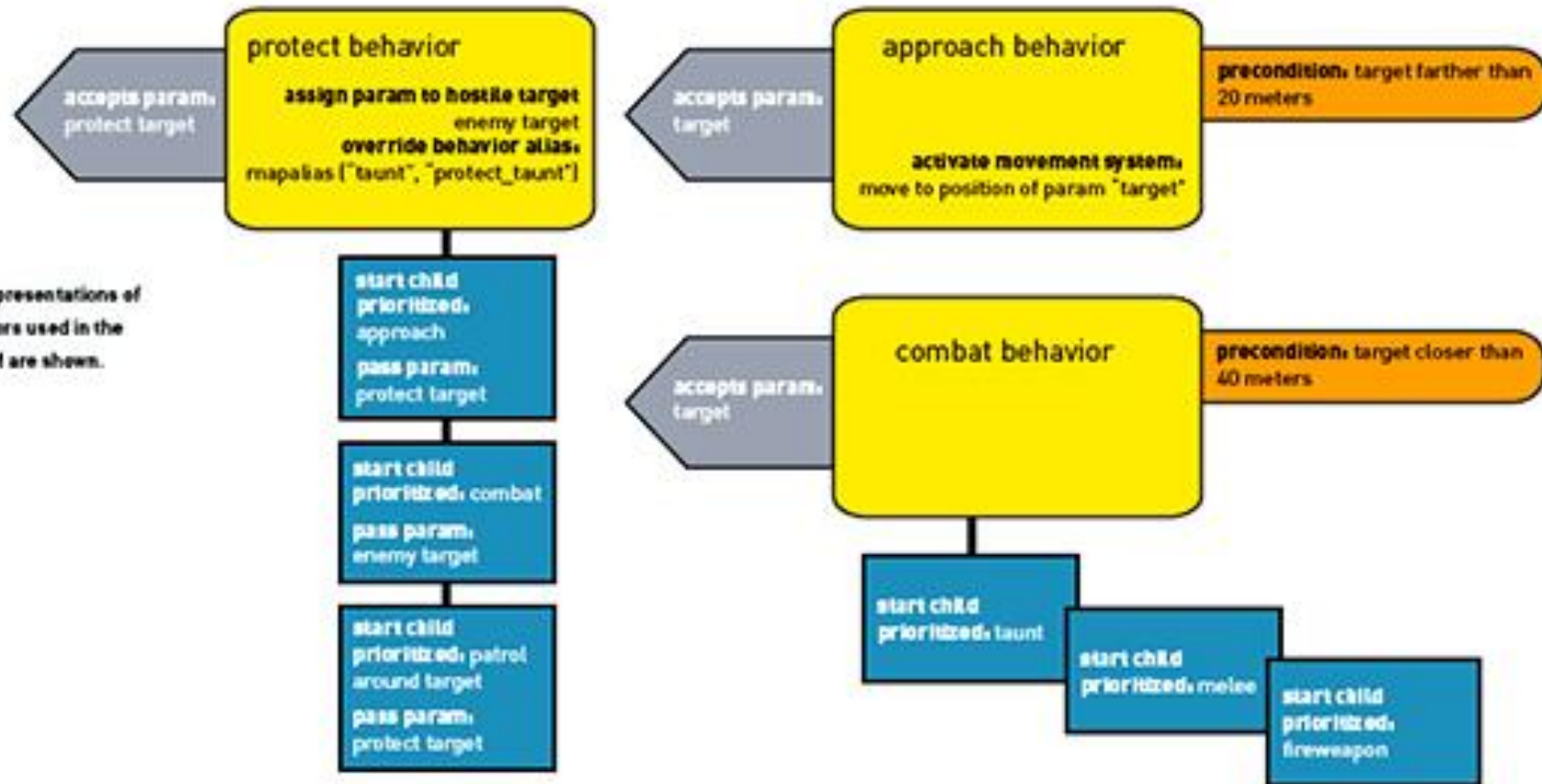


FIGURE 3 Representations of three behaviors used in the protect HFSM are shown.

More FSM Extensions

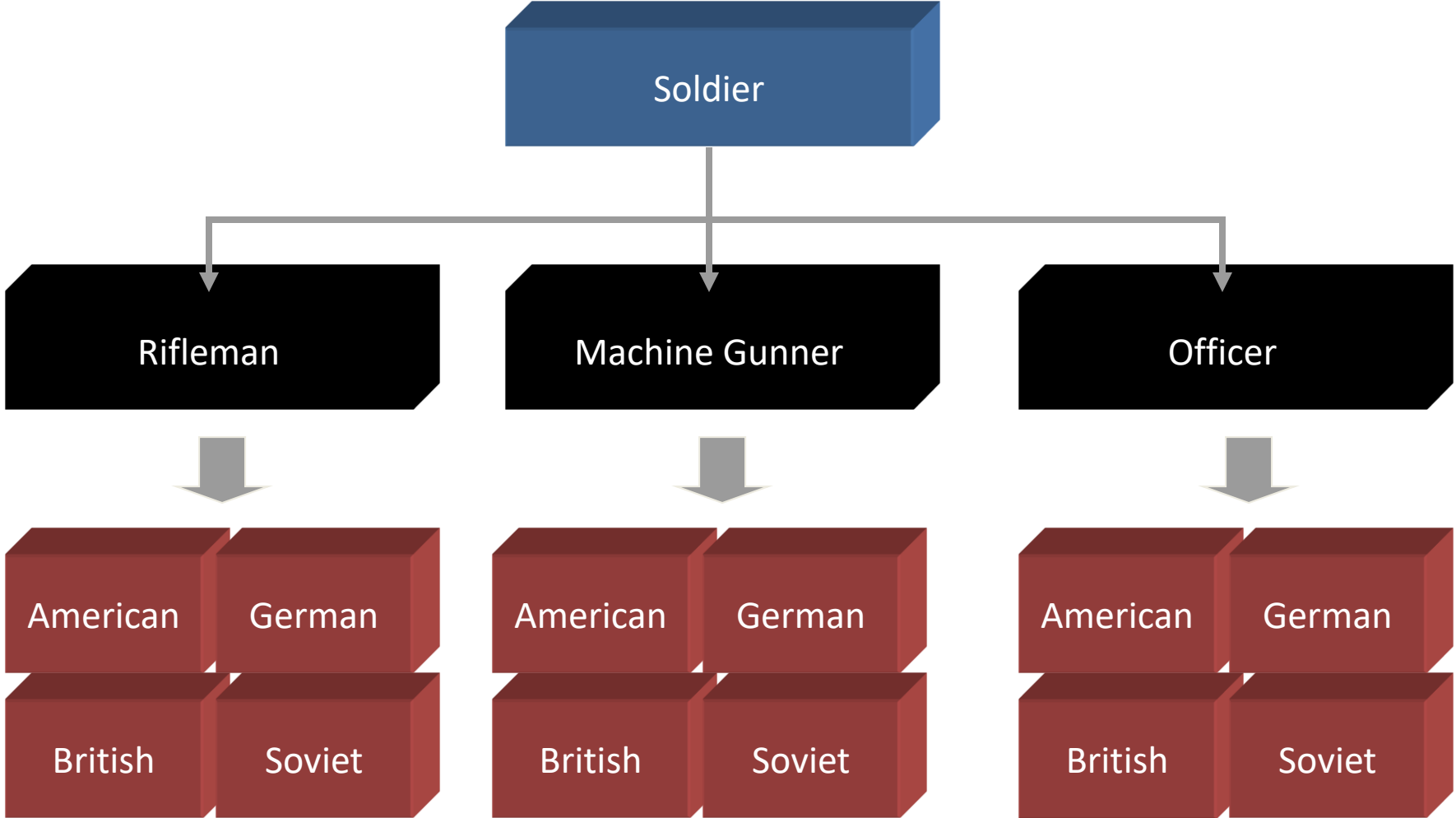
- Fuzzy State Machines
 - Degrees of truth allow multiple FSM's to contribute to character actions.
- Multiple FSM's
 - High level FSM coordinates several smaller FSM's.
- Polymorphic FSM's
 - Allows common behavior to be shared.
 - Soldier -> German -> Machine Gunner



Polymorphic FSMs

- Small changes to low level behaviors may be needed for different types of entities
- Polymorphism allows multiple versions of a single FSM to be executed on NPC state

Polymorphic FSM Example



Other FSM extensions

- Inter-character concurrent FSM
 - Coordination of multiple characters
- Intra-character concurrent FSM
 - Coordination of multiple behaviors within one NPC
- Levels of detail (LODs)
 - Analogous to LOD in graphs
 - E.g. crowd simulation
 - Close NPCs use fully elaborated FSM
 - Faraway NPCs use simpler FSMs or worse

Impl: Data Driven

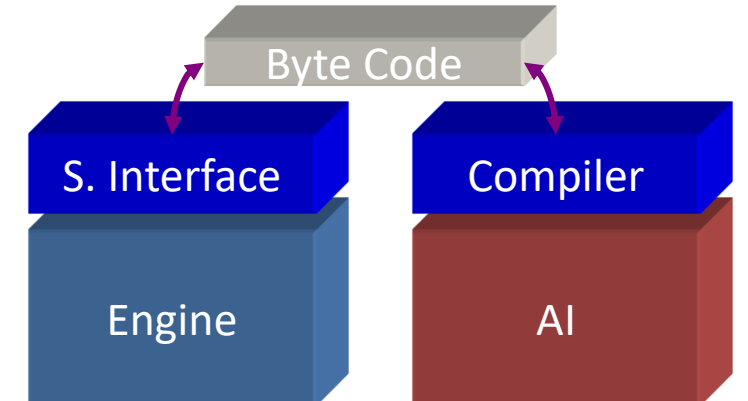
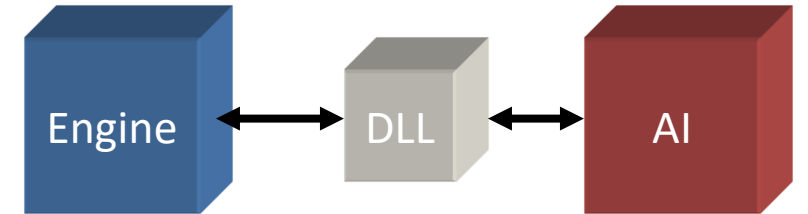
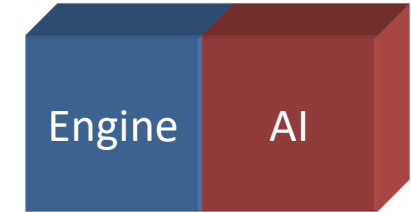
- Developer creates scripting language to control AI.
- Script is translated to C++ or bytecode.
- Requires a vocabulary for interacting with the game engine.
- A 'glue layer' must connect scripting vocabulary to game engine internals.
- Allows pluggable AI modules, even after the game has been released.

Scripted AI

- Many game engines are virtual machines
- Script is a program written in a programming language that makes calls into the game engine
- AI is the script
- Examples: Lua, Ruby, UnrealScript
- Powerful when paired with trigger systems

Game Engine Interfacing

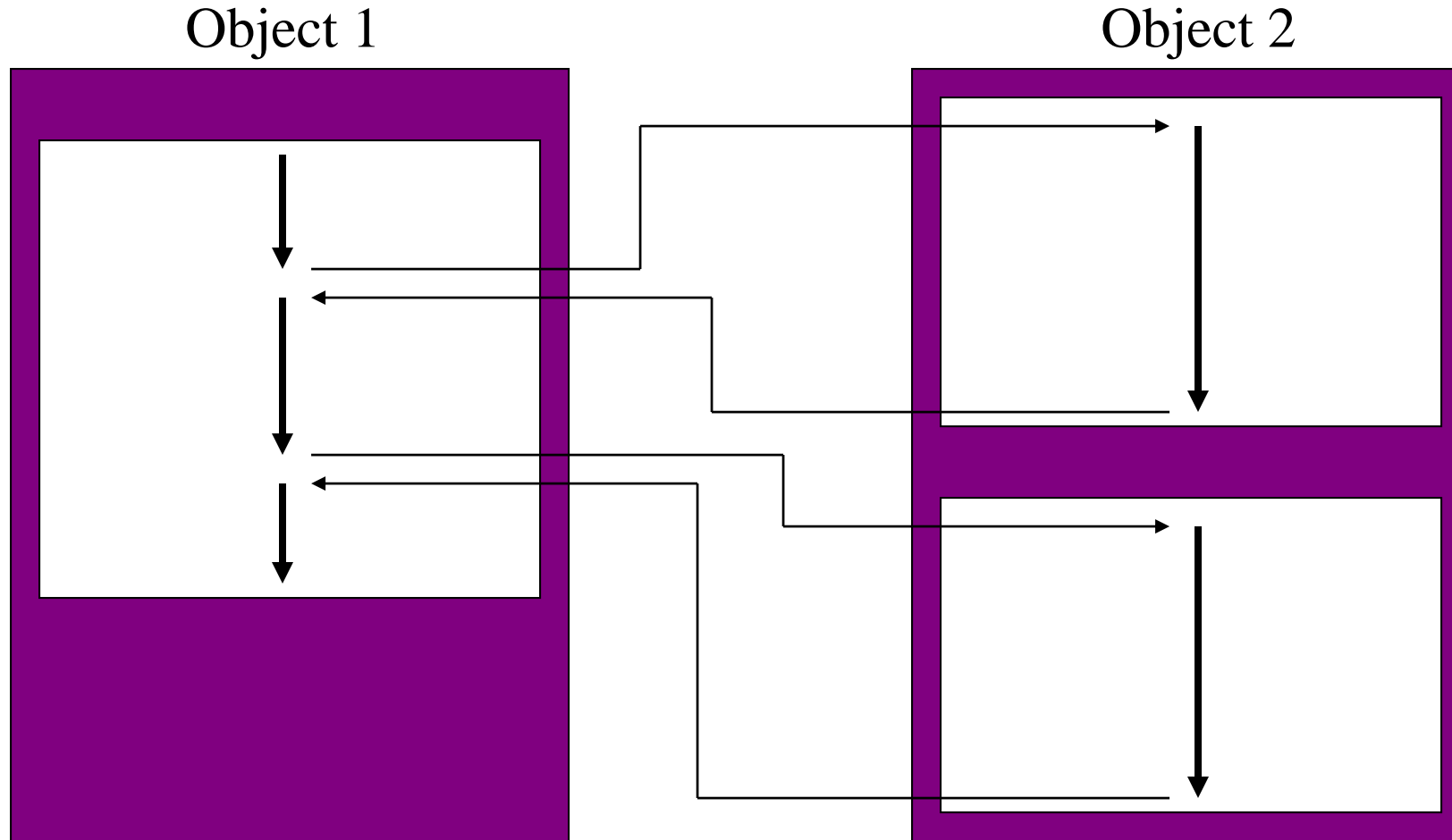
- Simple hard coded approach
 - Allows arbitrary parameterization
 - Requires full recompile
- Function pointers
 - Pointers are stored in a singleton or global
 - Implementation in DLL
 - Allows for pluggable AI.
- Data Driven
 - An interface must provide glue from engine to script engine.



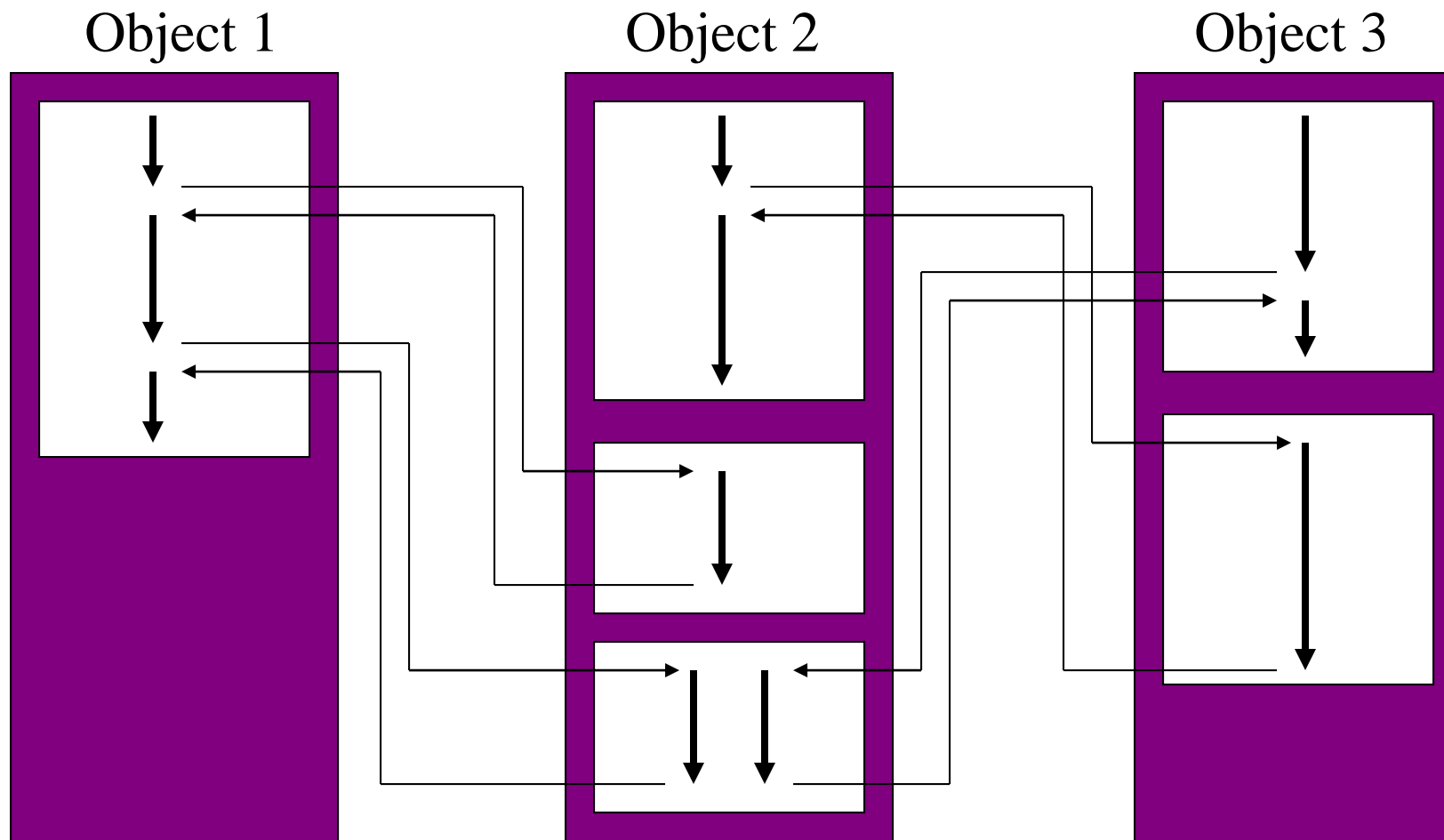
Processing Paradigms

- Polling
 - Simple and easy to debug.
 - Inefficient since FSM's are always evaluated.
- Event Driven Model
 - FSM registers which events it is interested in.
 - Requires Observer model in engine.
 - Hard to balance granularity of event model.
- Multithreaded
 - Each FSM assigned its own thread.
 - Requires thread-safe communication.
 - Conceptually elegant.
 - Difficult to debug.
 - Can be made more efficient using microthreads.

Single-threaded execution



Multi-threaded execution



Messaging/Triggers vs Polling

- Well-designed games tend to be event driven
- Examples (broadcast to relevant objs)
 - Wizard throws fireball at orc
 - Football player passes to teammate
 - Character lights a match (delayed dispatch match)
- Events / callbacks, publish / subscribe, Observers (GoF)
 - See Buckland Ch 2: Adding Messaging (pp69)

Time Management

- Helps manage time spent in processing FSM's.
- Scheduled Processing
 - Assigns a priority that decides how often that particular FSM is evaluated.
 - Results in uneven (unpredictable) CPU usage by the AI subsystem.
 - Can be mitigated using a load balancing algorithm.
- Time Bounded
 - Places a hard time bound on CPU usage.
 - More complex: interruptible FSM's

FSM Pros & Cons

Pro

- Ubiquitous (not only in digital games)
- Quick and simple to code
- (can be) Easy* to debug
- Very fast: Small computational overhead
- Intuitive
- Flexible
- Easy for designers without coding knowledge
- Non-deterministic FSM can make behavior unpredictable

Con

- When it fails, fails hard:
 - A transition from one state to another requires forethought (get stuck in a state or can't do the "correct" next action)
- Number of states can grow fast
 - Exponentially with number of events in world (multiple ways to react to same event given other variables): $s=2^e$
- Number of transitions/arcs can grow even faster: $a=s^2$
- Doesn't work with sequences of actions/memory

References / See Also

- AI Game Programming Wisdom 2
- Web
 - <http://ai-depot.com/FiniteStateMachines>
 - http://www.gamasutra.com/view/feature/130279/creating_all_humans_a_datadriven_.php
 - https://en.wikipedia.org/wiki/Virtual_finite-state_machine
- Buckland Ch 2
 - http://www.ai-junkie.com/architecture/state_driven/tut_state1.html
- Millington Ch 5
- Jarret Raim's slides (Dr. Munoz-Avila's GAI class 2005)
 - http://www.cse.lehigh.edu/~munoz/CSE497/classes/FSM_In_Games.ppt
- Mark Riedl, Brian O'Neill, and Brian Magerko

Trajectory Update

- HW4: A*
- To come: More decision making
 - Decision trees
 - Behavior trees
 - Rule based systems
 - Planning
 - Fuzzy Logic
 - Markov Systems