Design of Human Interface Game Software

– AI

What is AI?

- AI is the control of every non-human entity in a game
  - The other cars in a car game
  - The opponents and monsters in a shooter
  - Your units, your enemy's units and your enemy in a RTS game
- What AI is not
  - By physics: path of a cannon ball
  - Directly by game logic or user input
  - Purely random: which block falls next in Tetris

AI in the Game Loop

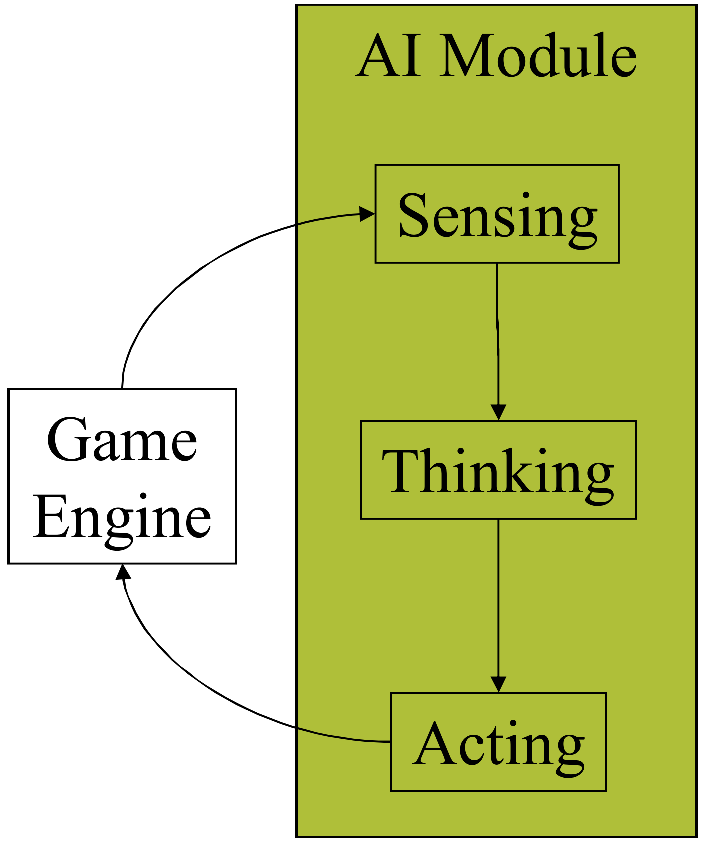
- AI is updated as part of the game loop, after user input, and before rendering
- There are issues here:
  - Which AI goes first?
  - Does the AI run on every frame?
  - Is the AI synchronized?

AI and Animation

- AI determines what to do and the animation does it
  - AI drives animation, deciding what action the animation system should be animating
  - Scenario 1: The AI issues orders like “move from A to B”, and it’s up to the animation system to do the rest
  - Scenario 2: The AI controls everything down to the animation clip to play
- Which scenario is best depends on the nature of the AI system and the nature of the animation system
  - Is the animation system based on move trees (motion capture), or physics, or something else
  - Does the AI perform collision avoidance? Does it do detailed planning?

AI Update Step

- The sensing phase determines the state of the world
  - May be very simple – state changes all come by message
  - Or complex – figure out what is visible, where your team is, etc
- The thinking phase decides what to do given the world
  - The core of AI
- The acting phase tells the animation what to do
  - Generally not interesting



AI by Polling

- The AI gets called at a fixed rate
- Senses: It looks to see what has changed in the world. For instance:
  - Queries what it can see
  - Checks to see if its animation has finished running
- And then acts on it

Event Driven AI

- Event driven AI does everything in response to events in the world
  - Events sent by message (basically, a function gets called when a message arrives, just like a user interface)
- Example messages:
  - A certain amount of time has passed, so update yourself
  - You have heard a sound
  - Someone has entered your field of view
- Real system are a mix – something changes, so you do some sensing

AI Techniques in Games

- Basic problem: Given the state of the world, what should I do?
- A wide range of solutions in games:
  - Finite state machines, Decision trees, Rule based systems, Neural networks, Fuzzy logic
- A wider range of solutions in the academic world:
  - Complex planning systems, logic programming, genetic algorithms, Bayes-nets
  - Typically, too slow for games

Goals of Game AI

- Several goals:
  - Goal driven – the AI decides what it should do, and then figures out how to do it
  - Reactive – the AI responds immediately to changes in the world
  - Knowledge intensive – the AI knows a lot about the world and how it behaves, and embodies knowledge in its own behavior
  - Consistent – Embodies a believable, consistent character
  - Fast and easy development
  - Low CPU and memory usage
- These conflict in almost every way

Two Measures of Complexity

- Complexity of Execution
  - How fast does it run as more knowledge is added?
  - How much memory is required as more knowledge is added?
  - Determines the run-time cost of the AI
- Complexity of Specification
  - How hard is it to write the code?
  - As more “knowledge” is added, how much more code needs to be added?
  - Determines the development cost, and risk

Expressiveness

- What behaviors can easily be defined, or defined at all?
- Propositional logic:
  - Statements about specific objects in the world – no variables
  - Jim is in room7, Jim has the rocket launcher, the rocket launcher does splash damage
  - Go to room8 if you are in room7 through door14
- Predicate Logic:
  - Allows general statement – using variables
  - All rooms have doors
  - All splash damage weapons can be used around corners
  - All rocket launchers do splash damage
  - Go to a room connected to the current room

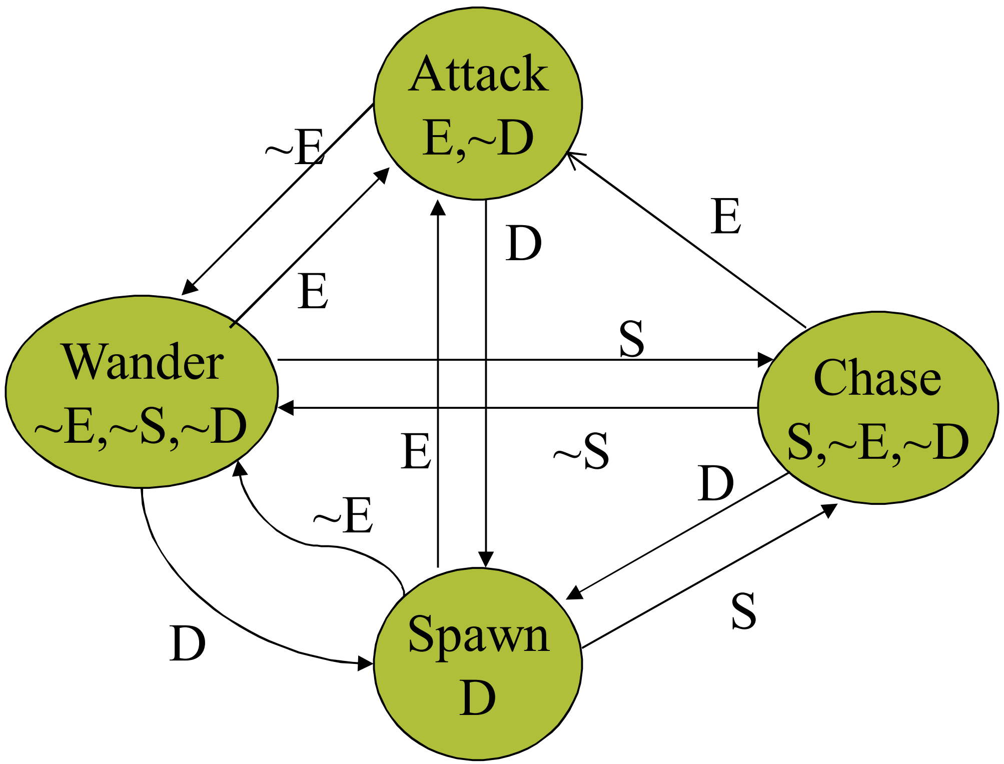
Finite State Machines (FSMs)

- A set of *states* that the agent can be in
- Connected by *transitions* that are triggered by a change in the world
- Normally represented as a directed graph, with the edges labeled with the transition event
- Ubiquitous in computer game AI
- You might have seen them, in formal language theory (or compilers)

Quake Bot Example

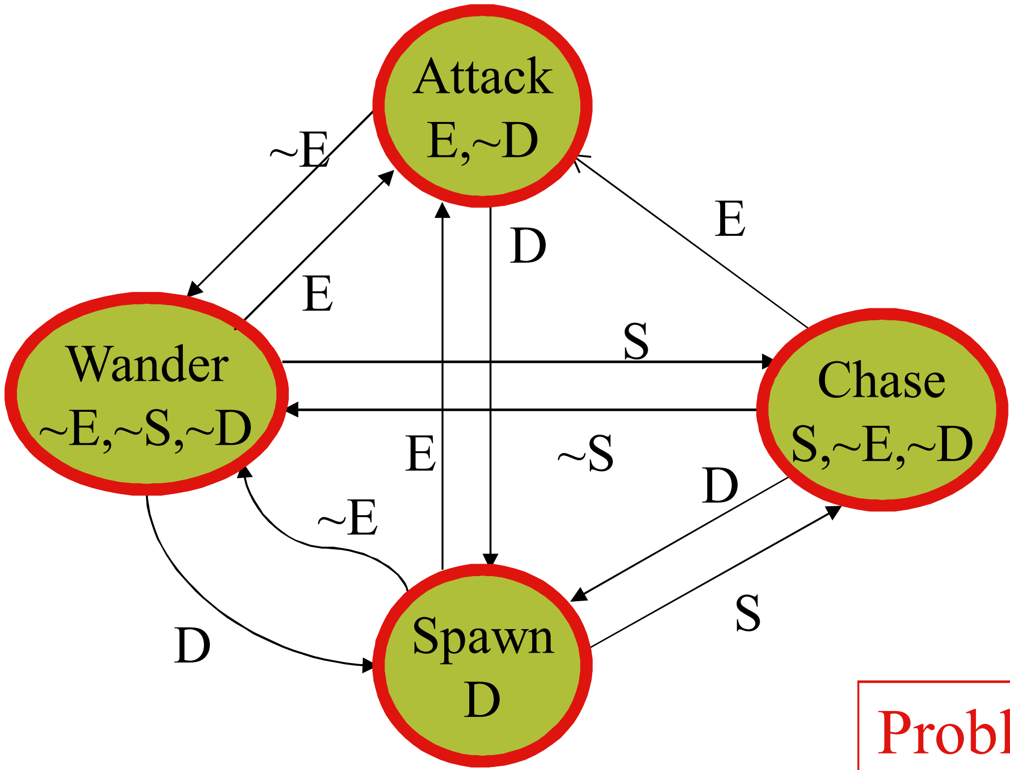
- Types of behavior to capture:
  - **Wander** randomly if don't see or hear an enemy
  - When see enemy, **attack**
  - When hear an enemy, **chase** enemy
  - When die, **respawn**
  - Extras: If health is low and see an enemy, **retreat**
- Extensions:
  - When see power-ups during wandering, collect them

Example FSM



- States:
  - E: enemy in sight
  - S: sound audible
  - D: dead
- Events:
  - E: see an enemy
  - S: hear a sound
  - D: die
- Action performed:
  - On each transition
  - On each update in some states (e.g. attack)

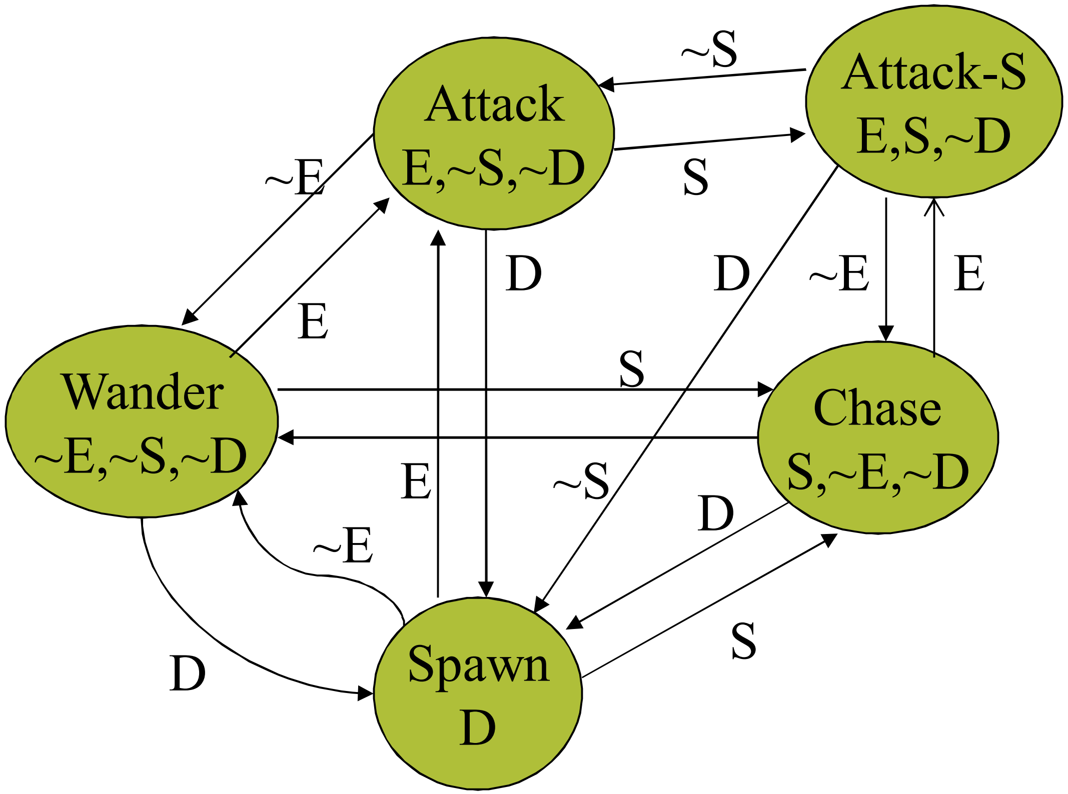
Example FSM Problem



- States:
  - E: enemy in sight
  - S: sound audible
  - D: dead
- Events:
  - E: see an enemy
  - S: hear a sound
  - D: die

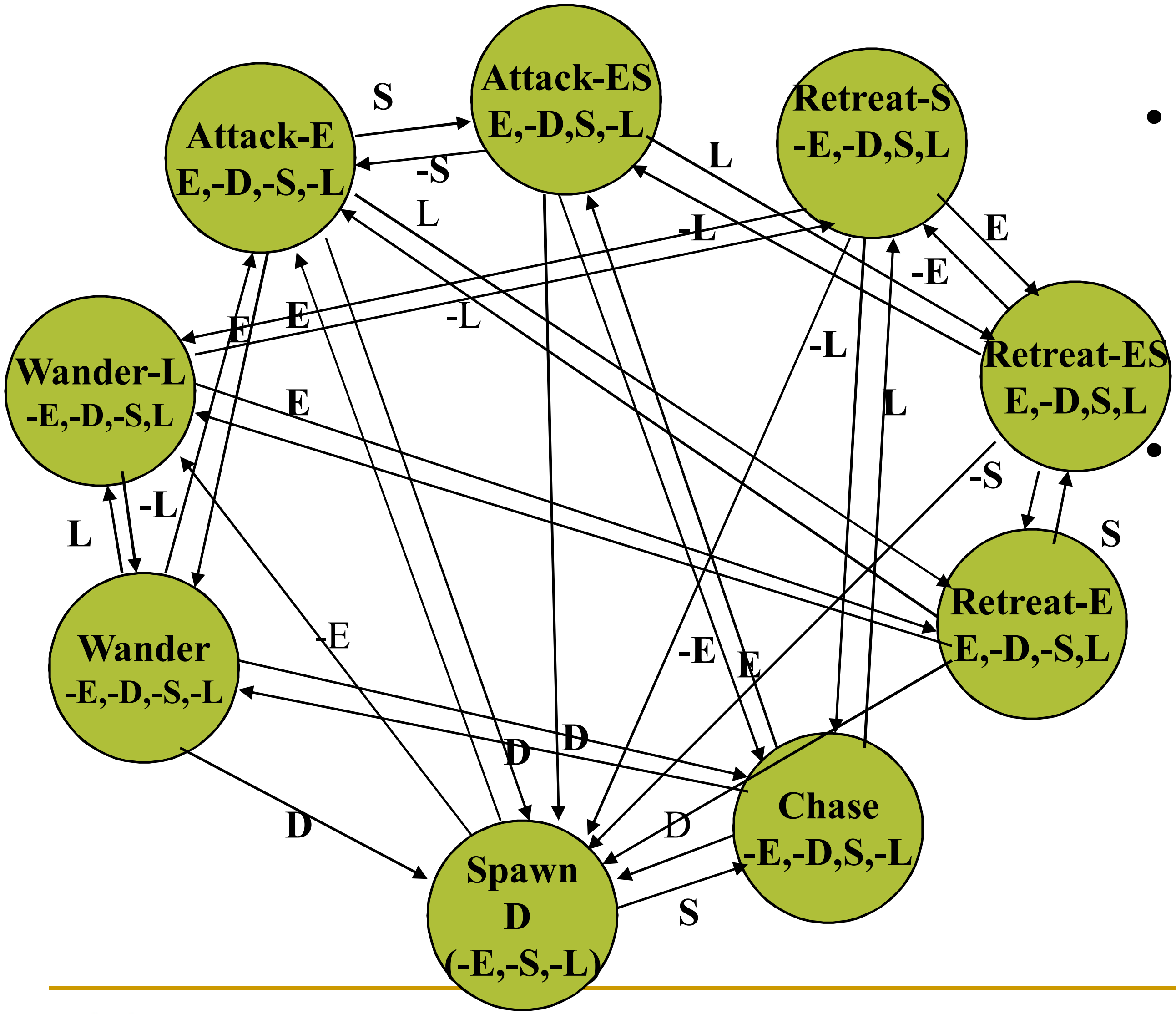
Problem: Can't go directly from attack to chase. Why not?

Better Example FSM



- States:
  - E: enemy in sight
  - S: sound audible
  - D: dead
- Events:
  - E: see an enemy
  - S: hear a sound
  - D: die
- Extra state to recall whether or not heard a sound while attacking

Example FSM with Retreat

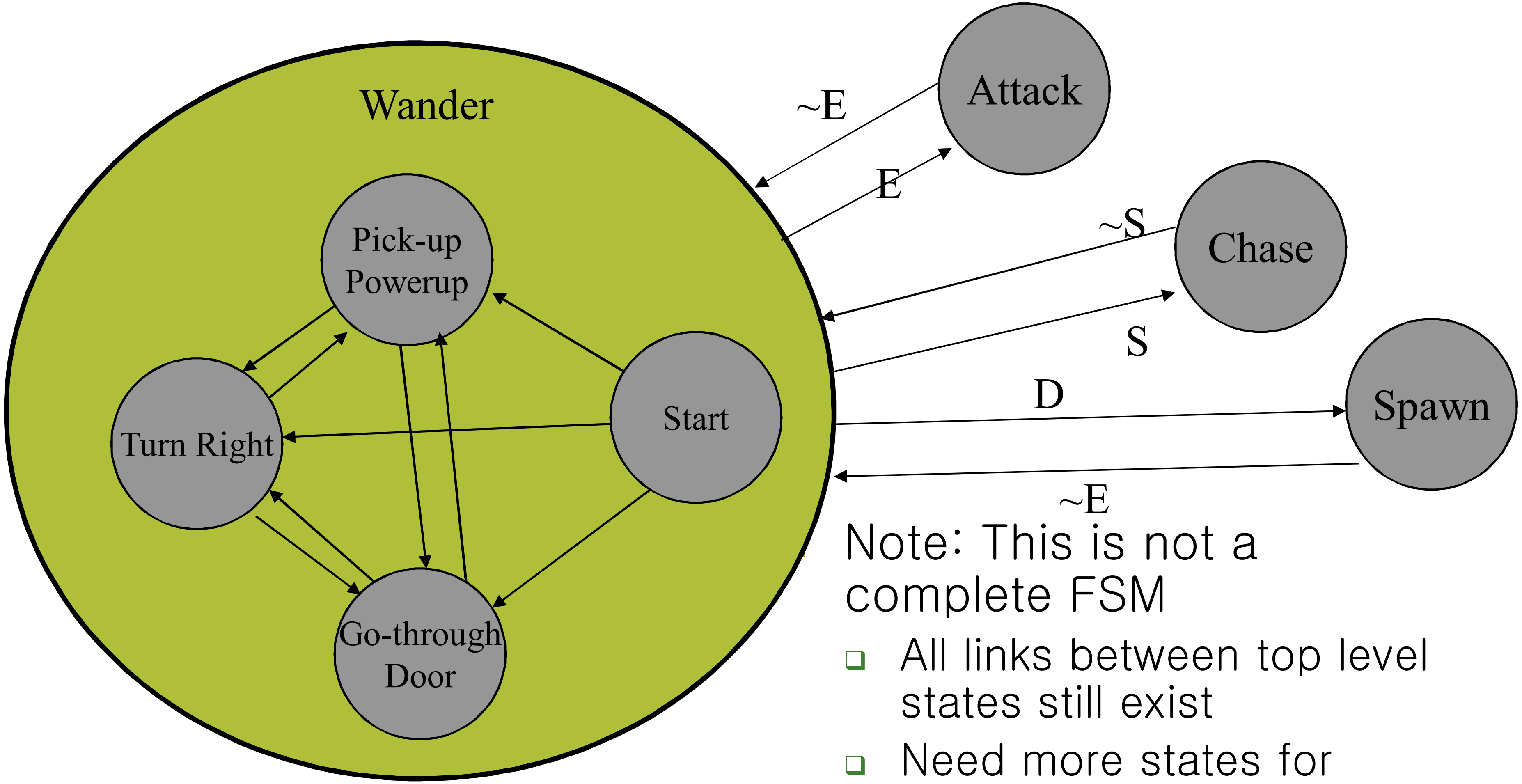


- States:
  - E: enemy in sight
  - S: sound audible
  - D: dead
  - L: Low health
- Worst case: Each extra state variable can add $2n$ extra states
  - n = number of existing states

Hierarchical FSMs

- What if there is no simple action for a state?
- Expand a state into its own FSM, which explains what to do if in that state
- 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
  - Or, random choice
  - Depends on the nature of the behavior

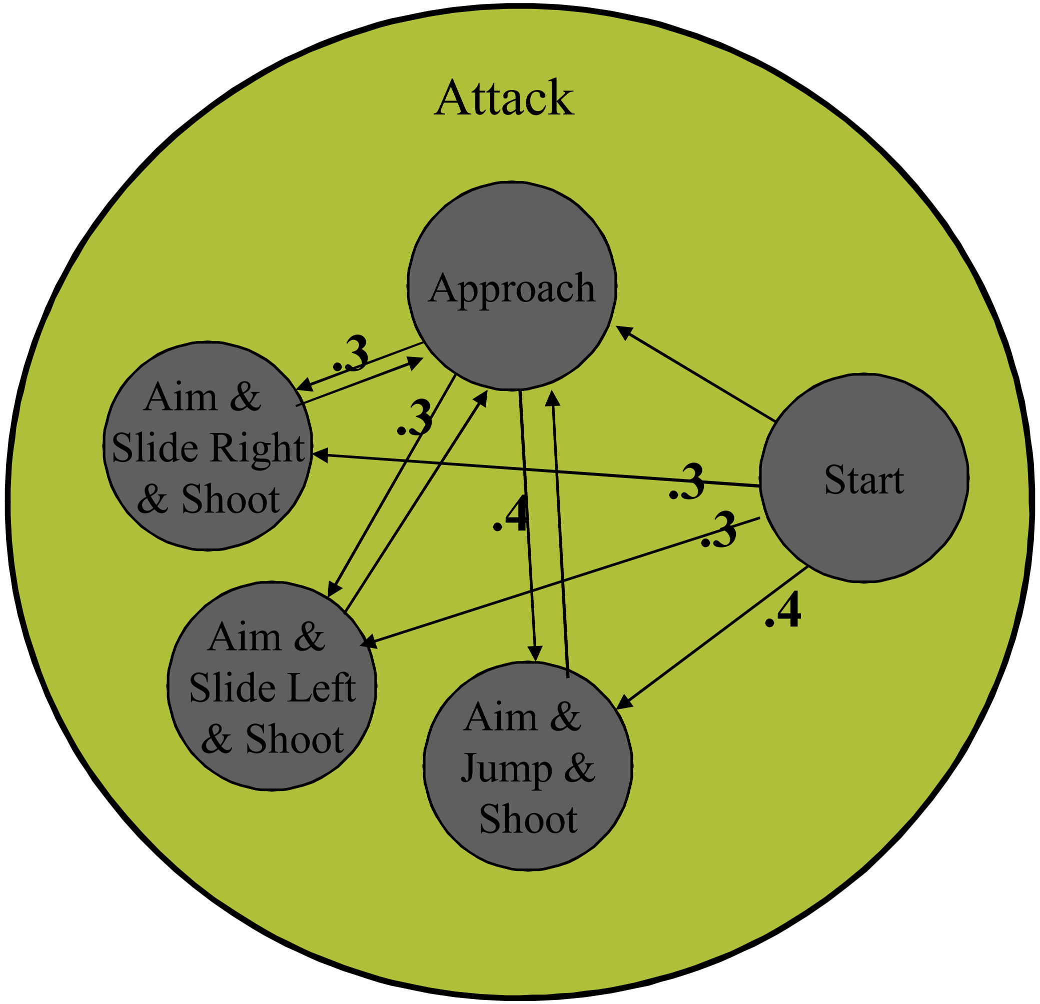
Hierarchical FSM Example



Note: This is not a complete FSM

- All links between top level states still exist
- Need more states for wander

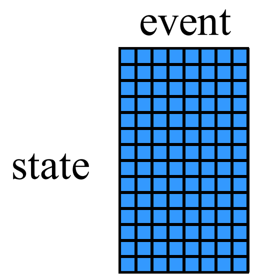
Non-Deterministic Hierarchical FSM

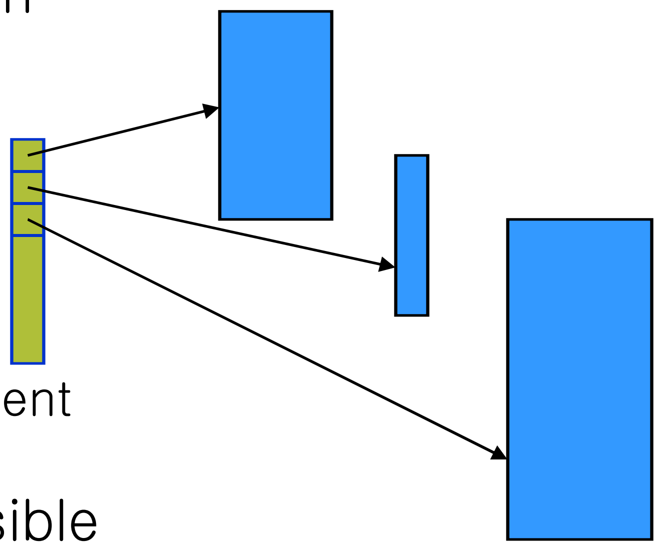


- Adds variety to actions
- Have multiple transitions for the same event
- Label each with a probability that it will be taken
- Randomly choose a transition at run-time
- New state only depends on the previous state

Efficient Implementation

- Compile into an array of [state-name, event]
- state-name_{i+1} := array[state-name_i, event]
- Switch on state-name to call execution logic
- Hierarchical
  - Create array for every FSM
  - Have stack of states
    - Classify events according to stack
    - Update state which is sensitive to current event
- Non-deterministic: Have array of possible transitions for every (state-name,event) pair, and choose one at random





FSM Advantages

- Very fast – one array access
- Expressive enough for simple behaviors or characters that are intended to be "dumb"
- Can be compiled into compact data structure
  - Dynamic memory: current state
  - Static memory: state diagram – array implementation
- Can create tools so non-programmer can build behavior
- Non-deterministic FSM can make behavior unpredictable

FSM Disadvantages

- Number of states can grow very fast
  - Exponentially with number of events: $s=2^e$
- Number of arcs can grow even faster: $a=s^2$
- Propositional representation
  - Difficult to put in “pick up the better powerup”, “attack the closest enemy”
  - Expensive to count: Wait until the third time I see enemy, then attack
    - Need extra events: First time seen, second time seen, and extra states to take care of counting

Classification

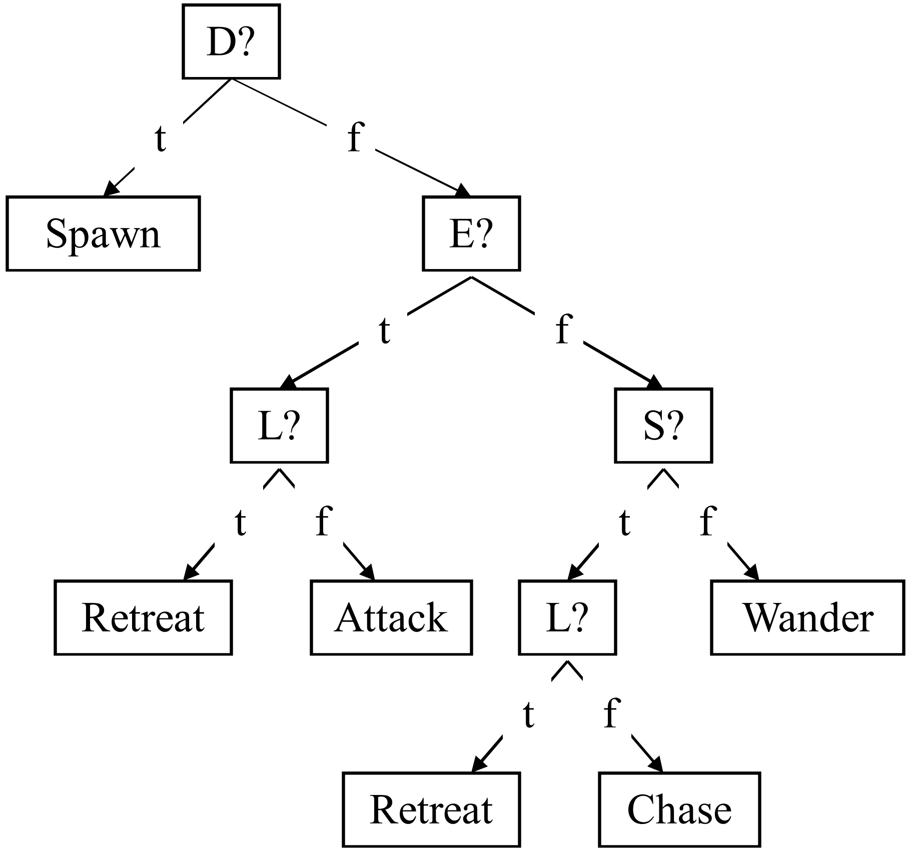
- Our aim is to decide which action to take given the world state
- Convert this to a classification problem:
  - The state of the world is a set of *attributes* (or *features*)
    - Who I can see, how far away they are, how much energy, …
  - Given any state, there is one appropriate action
    - Extends to multiple actions at the same time
  - The action is the *class* that a world state belongs to
    - Low energy, see the enemy means I should be in the retreat state
- Classification problems are *very* well studied

Decision Trees

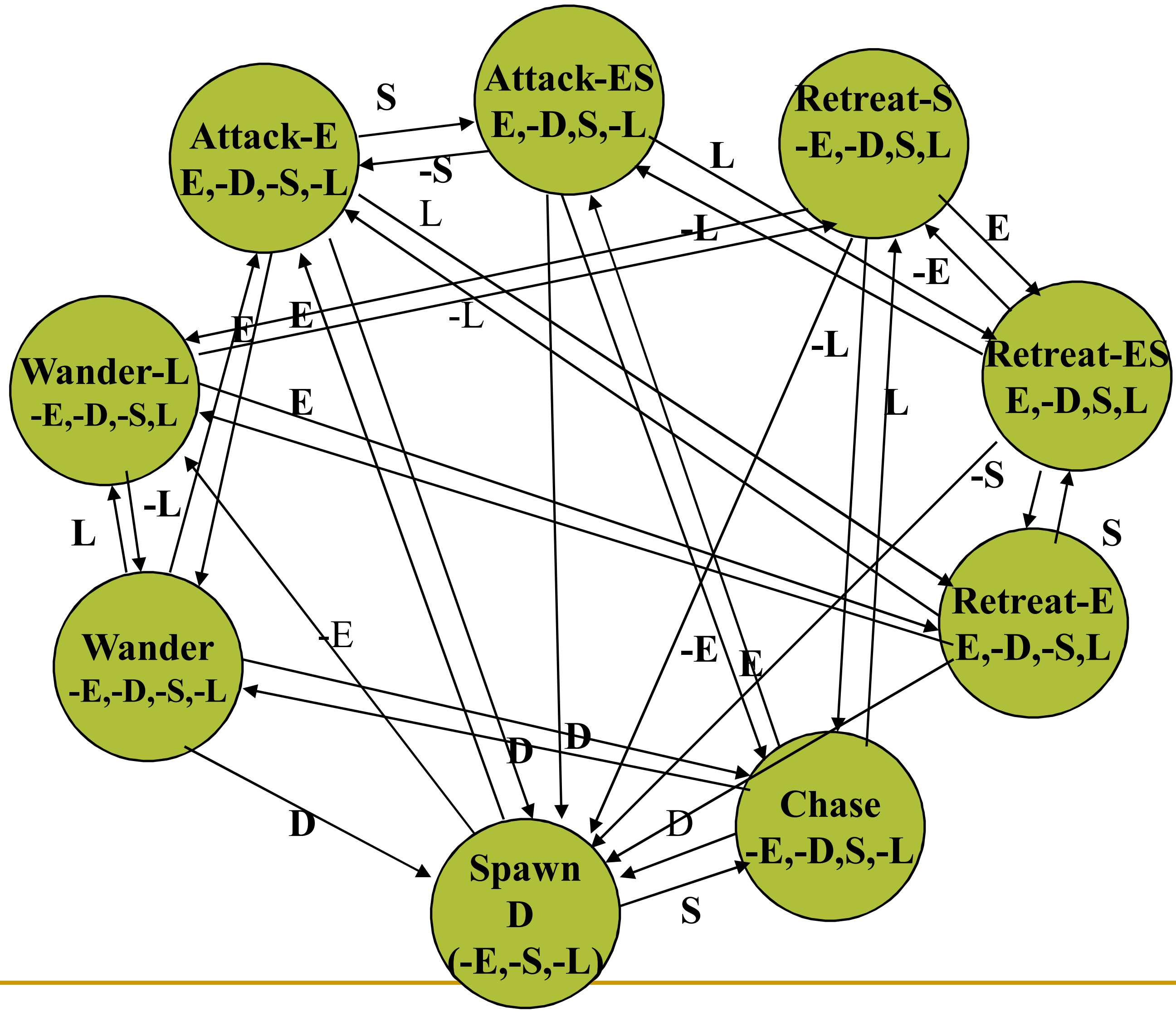
- Nodes represent attribute tests
  - One child for each possible outcome of the test
- Leaves represent classifications
  - Can have the same classification for several leaves
- Classify by descending from root to a leaf
  - At each node perform the test and descend the appropriate branch
  - When a leaf is reached return the classification (action) of that leaf
- Decision tree is a "disjunction of conjunctions of constraints on the attribute values of an instance"
  - Action if (A and B and C) or (A and ~B and D) or (…) …
  - Retreat if (low health and see enemy) or (low health and hear enemy) or (…) …

Decision Tree for Quake

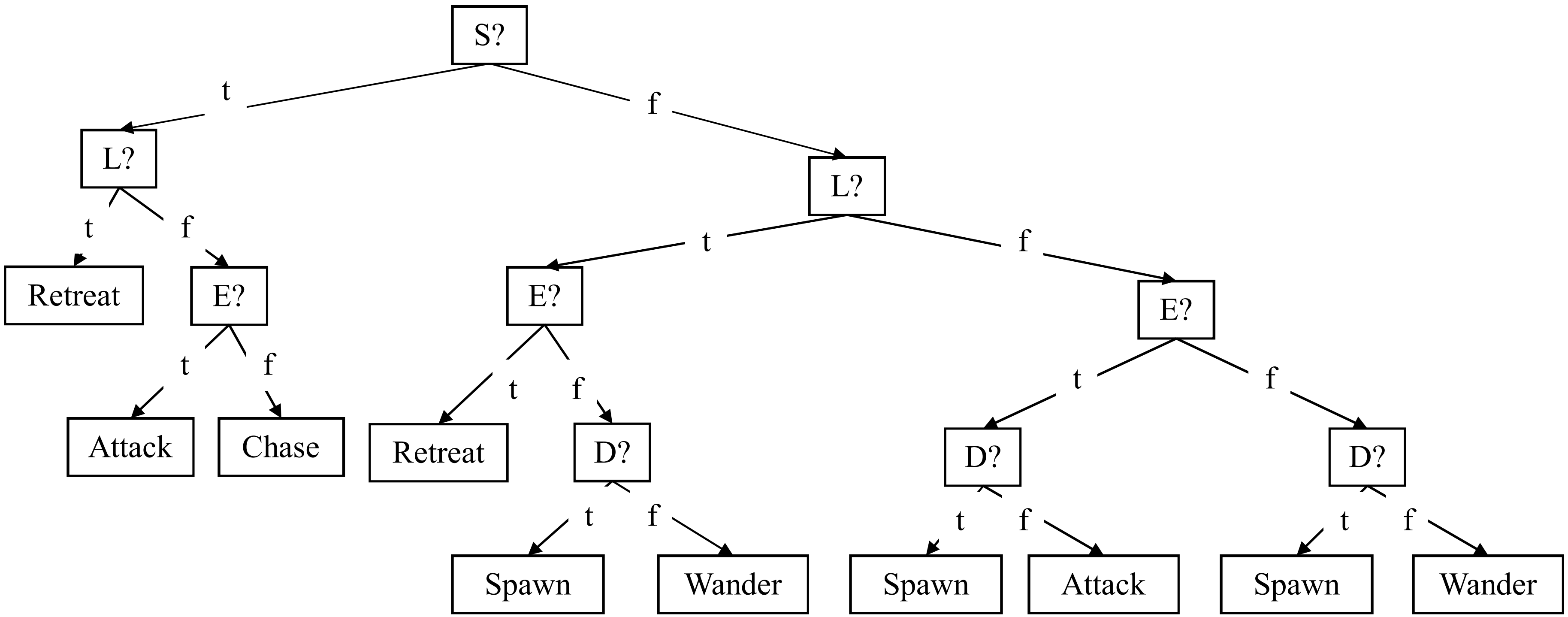
- Just one tree
- Attributes: Enemy=<t,f> Low=<t,f> Sound=<t,f> Death=<t,f>
- Actions: Attack, Retreat, Chase, Spawn, Wander
- Could add additional trees:
  - If I'm attacking, which weapon should I use?
  - If I'm wandering, which way should I go?
  - Can be thought of as just extending given tree (but easier to design)
  - Or, can share pieces of tree, such as a Retreat sub-tree



Compare and Contrast



Different Trees – Same Decision

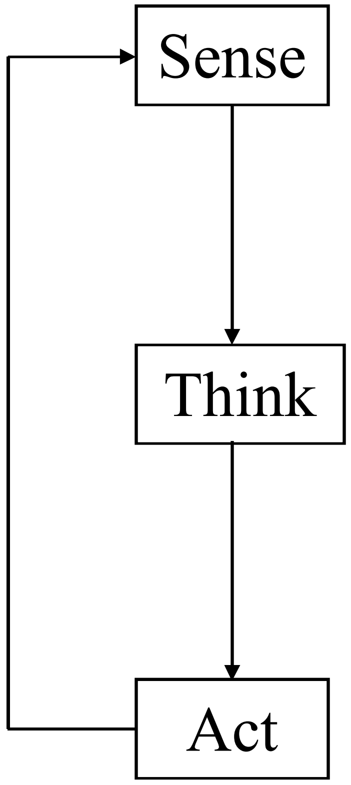


Deciding on Actions

- Each time the AI is called:
  - Poll each decision tree for current output
  - Event driven – only call when state changes
- Need current value of each input attribute
  - All sensor inputs describe the state of the world
- Store the state of the environment
  - Most recent values for all sensor inputs
  - Change state upon receipt of a message
  - Or, check validity when AI is updated
  - Or, a mix of both (polling and event driven)

Sense, Think, Act Cycle

- Sense
  - Gather input sensor changes
  - Update state with new values
- Think
  - Poll each decision tree
- Act
  - Execute any changes to actions



Building Decision Trees

- Decision trees can be constructed by hand
  - Think of the questions you would ask to decide what to do
  - For example: Tonight I can study, play games or sleep. How do I make my decision?
- But, decision trees are typically *learned*:
  - Provide examples: many sets of attribute values and resulting actions
  - Algorithm then constructs a tree from the examples
  - Reasoning: We don’t know how to decide on an action, so let the computer do the work
  - Whose behavior would we wish to learn?

Learning Decision Trees

- Decision trees are usually learned by induction
  - Generalize from examples
  - Induction doesn't guarantee correct decision trees
- Bias towards smaller decision trees
  - Occam's Razor: Prefer simplest theory that fits the data
  - Too expensive to find the very smallest decision tree
- Non-Incremental Learning
  - Process all the examples at once
  - Incremental learning is inefficient

Induction

- If X is true in every example that results in action A, then X must always be true for action A
  - More examples are better
  - Errors in examples cause difficulty
    - If X is true in most examples X must always be true
  - Note that induction can result in errors
    - It may just be coincidence that X is true in all the examples
- Typical decision tree learning determines what tests are always true for each action
  - Assumes that if those things are true again, then the same action should result

Learning Algorithms

- Recursive algorithms
  - Find an attribute test that separates the actions
  - Divide the examples based on the test
  - Recurse on the subsets
- What does it mean to separate?
  - Ideally, there are no actions that have examples in both sets
  - Failing that, most actions have most examples in one set
  - The things to measure is entropy – the degree of homogeneity (or lack of it) in a set
    - Entropy is also important for compression

Induction requires Examples

- Where do examples come from?
  - Programmer/designer provides examples
  - Capture an expert player's actions, and the game state, while they play
- # of examples needed depends on difficulty of concept
  - Difficulty: Number of tests needed to determine the action
  - More is always better
- Training set vs. Testing set
  - Train on most (75%) of the examples
  - Use the rest to validate the learned decision trees by estimating how well the tree does on examples it hasn't seen

Decision Tree Advantages

- Simpler, more compact representation
- State is recorded in a memory
  - Create “internal sensors” – Enemy-Recently-Sensed
- Easy to create and understand
  - Can also be represented as rules
- Decision trees can be learned

Decision Tree Disadvantages

- Decision tree engine requires more coding than FSM
  - Each tree is "unique" sequence of tests, so little common structure
- Need as many examples as possible
- Higher CPU cost – but not much higher
- Learned decision trees may contain errors

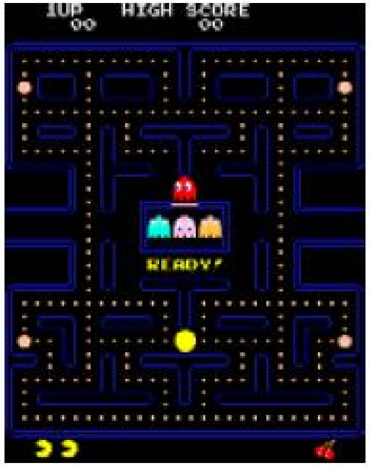
Rule-Based Approaches

- Rule-based approaches: popular because
  - Familiar
  - Predictable and testable
- Rules specify the action depending on circumstances:
  - Deterministic: one action for each situation
  - Random: multiple possible actions in the same situation
  - Weighted: certain actions have a higher probability
  - Based on State/Environment information: take the direction that leads you to the opponent

Rule-Based Approaches

- Example: a simple rule-based system for moving a ghost agent in Pac-Man

Ahead	Right	Left	Action
Open	—	—	Go ahead
Blocked	Open	—	Turn Right
Blocked	Blocked	Open	Turn Left
Blocked	Blocked	Blocked	Go backwards

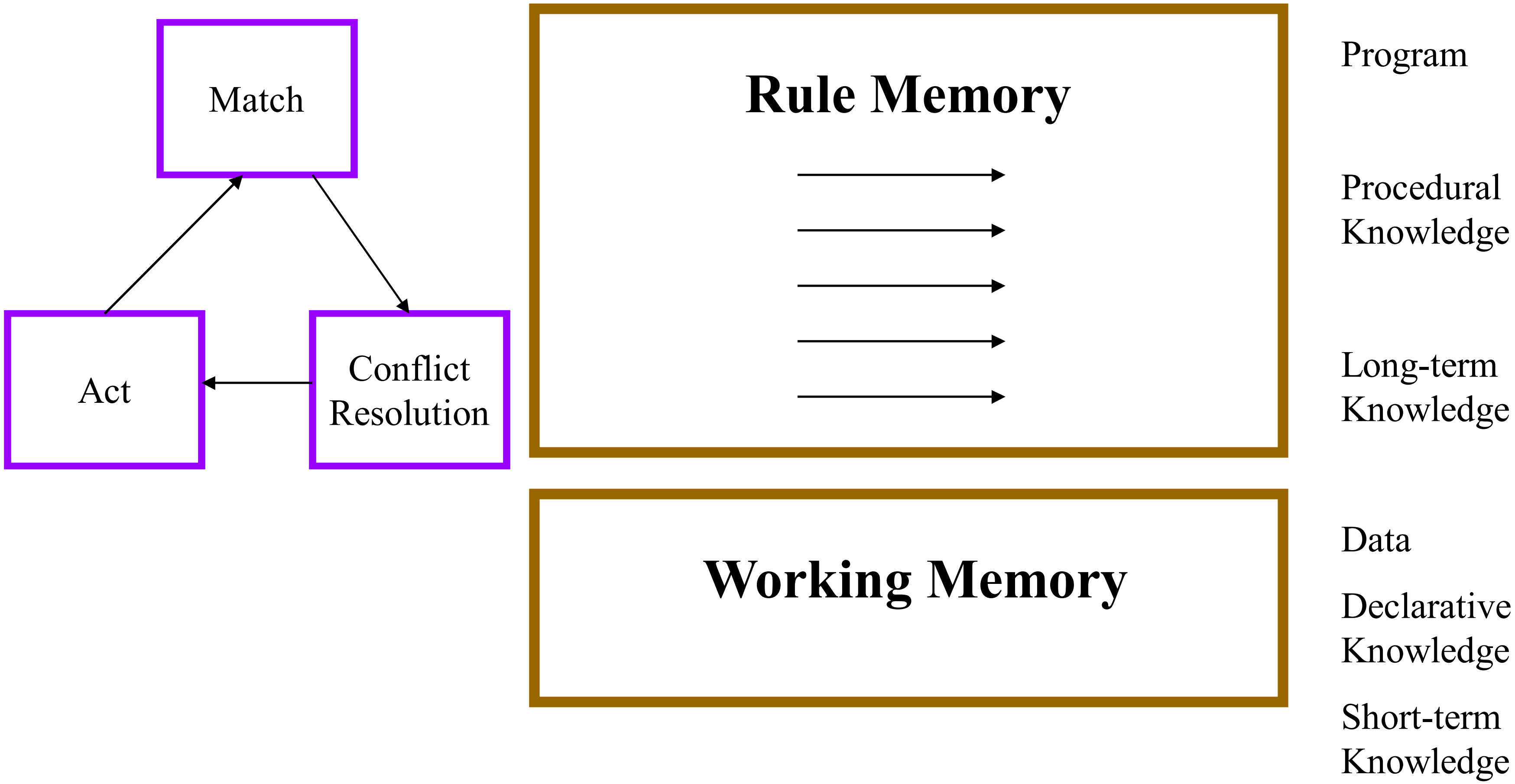


- Note:
  - Easy to add randomness if we like
  - Does not consider the location/state of the player

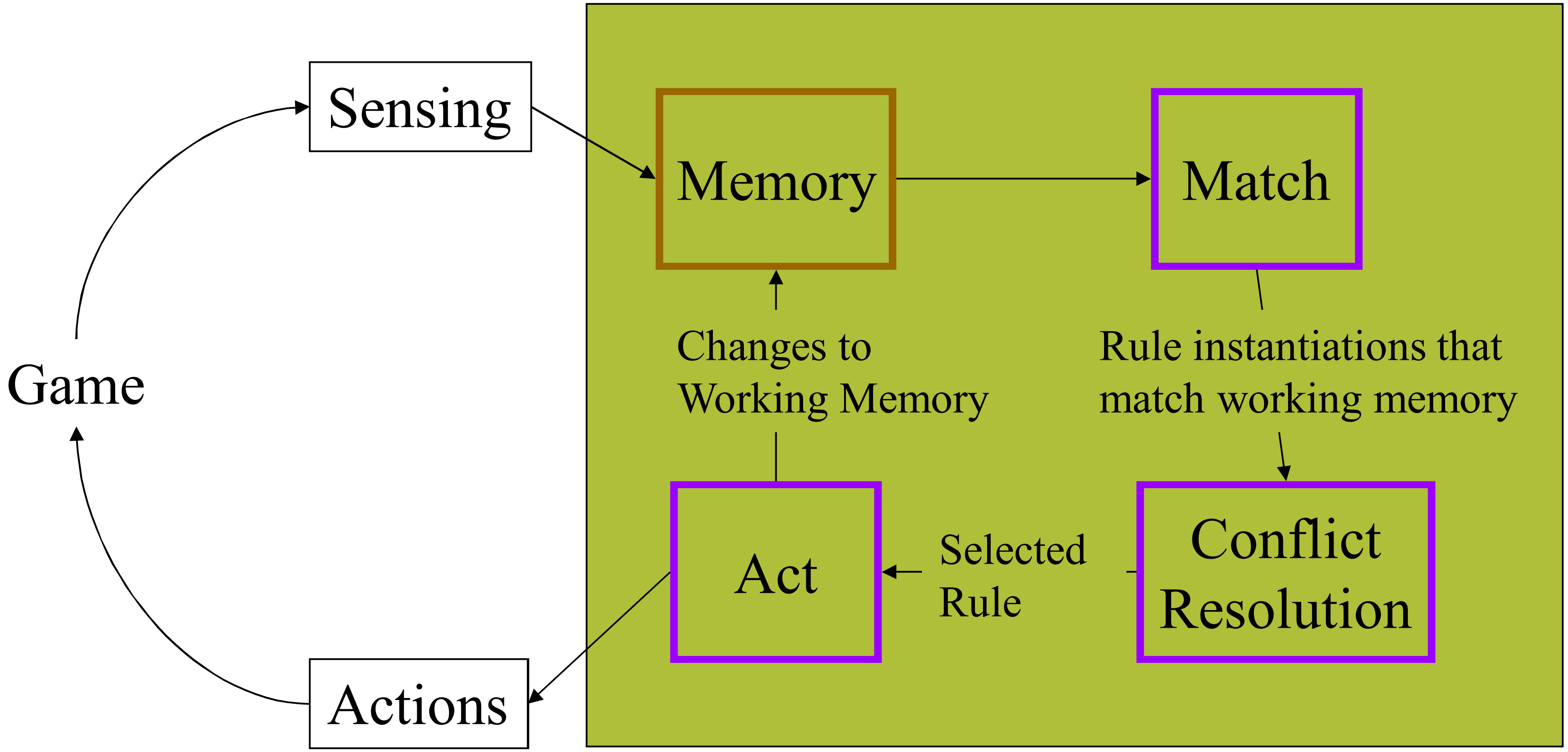
Rule-Based Systems

- Decision trees can be converted into rules
  - Just test the disjunction of conjunctions for each leaf
- More general rule-based systems let you write the rules explicitly
- System consists of:
  - A rule set – the rules to evaluate
  - A working memory – stores state
  - A matching scheme – decides which rules are applicable
  - A conflict resolution scheme – if more than one rule is applicable, decides how to proceed
- What types of games make the most extensive use of rules?

Rule-Based Systems Structure



AI Cycle


Sensing
Memory
Match
Game
Changes to Working Memory
Rule instantiations that match working memory
Act
Selected Rule
Conflict Resolution
Actions

Age of Kings

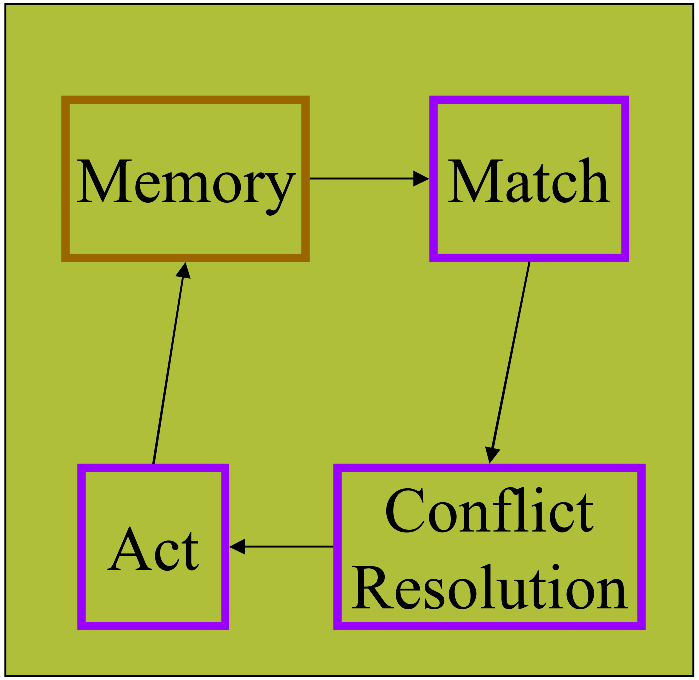
```
; The AI will attack once at 1100 seconds and then again
; every 1400 sec, provided it has enough defense soldiers.

(defrule
      (game-time > 1100)            Rule
=>
      (attack-now)
      (enable-timer 7 1400))        Action

(defrule
      (timer-triggered 7)
      (defend-soldier-count >= 12)
=>
      (attack-now)
      (disable-timer 7)
      (enable-timer 7 1400))
```

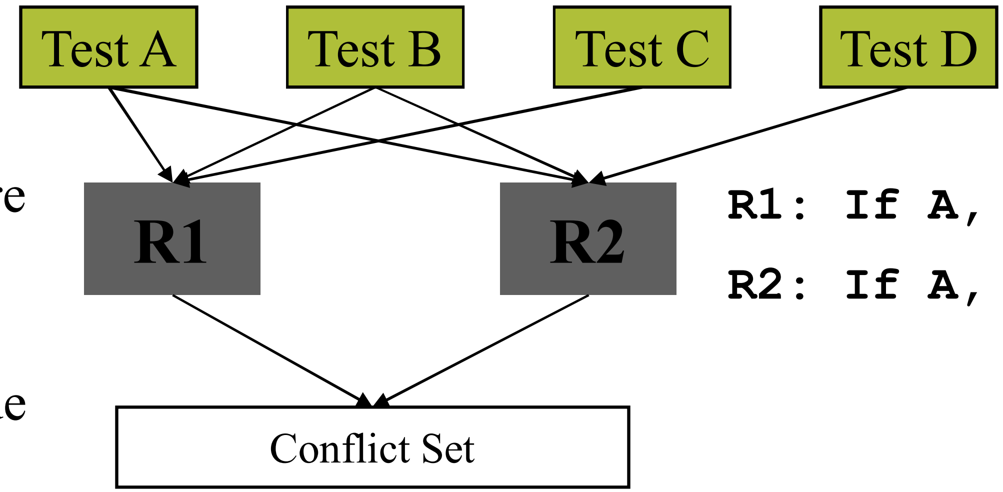
Implementing Rule-Based Systems

- Where does the time go?
  - 90-95% goes to Match
- Matching all rules against all of working memory each cycle is way too slow
- Key observation
  - # of changes to working memory each cycle is small
  - If conditions, and hence rules, can be associated with changes, then we can make things fast (event driven)



Efficient Special Case

- If only simple tests in conditions, compile rules into a *match net*
  - Simple means: Can map changes in state to rules that must be reevaluated
- Process changes to working memory
- Associate changes with tests
- Expected cost: Linear in the number of changes to working memory



Rules: Bit vectors store which tests are true

Rules with all tests true go in conflict set

```
R1: If A, B, C, then …
R2: If A, B, D, then …
```

General Case

- Rules can be arbitrarily complex
  - In particular: function calls in conditions and actions
- If we have arbitrary function calls in conditions:
  - Can't hash based on changes
  - Run through rules one at a time and test conditions
  - Pick the first one that matches (or do something else)
  - Time to match depends on:
    - Number of rules
    - Complexity of conditions
    - Number of rules that don't match

Baulders Gate

```
IF
  Heard([PC],UNDER_ATTACK)
  !InParty(LastAttackerOf(LastHeardBy(Myself)))
  Range(LastAttackerOf(LastHeardBy(Myself)),5)
  !StateCheck(LastAttackerOf(LastHeardBy(Myself)),
              STATE_PANIC)
  !Class(Myself,FIGHTER_MAGE_THIEF)
THEN
  RESPONSE #100
  EquipMostDamagingMelee()
  AttackReevaluate(LastAttackerOf(LastHeardBy(Myself)),30)
END
```

Conflict Resolution Strategies

- What do we do if multiple rules match?
- Rule order – pick the first rule that matches
  - Makes order of loading important – not good for big systems
- Rule specificity – pick the most specific rule
- Rule importance – pick rule with highest priority
  - When a rule is defined, give it a priority number
  - Forces a total order on the rules – is right 80% of the time
  - Decide Rule 4 [80] is better than Rule 7 [70]
  - Decide Rule 6 [85] is better than Rule 5 [75]
  - Now have ordering between all of them – even if wrong

Rule-based System: Good and Bad

- Advantages
  - Corresponds to way people often think of knowledge
  - Very expressive
  - Modular knowledge
    - Easy to write and debug compared to decision trees
    - More concise than FSM
- Disadvantages
  - Can be memory intensive
  - Can be computationally intensive
  - Sometimes difficult to debug