Harel's Statecharts

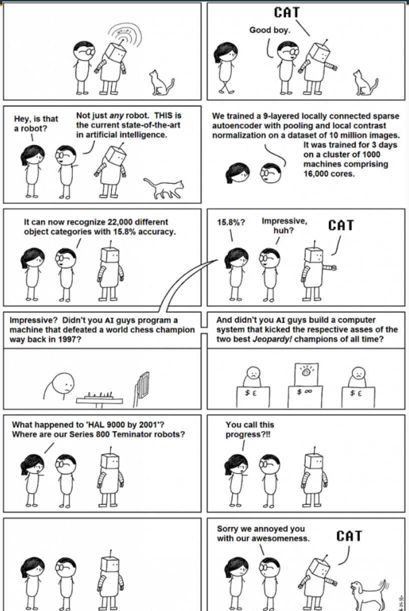
Giacomo Bergami

Newcaslte University

Objectives

- Recognizing the need and utility of non-training based AI.
- Using simple animation state machines within Unity.
- Determining what characterizes different states and how events can be triggered (physics- or probing-based).
- Adding stochastic state machine behaviour.

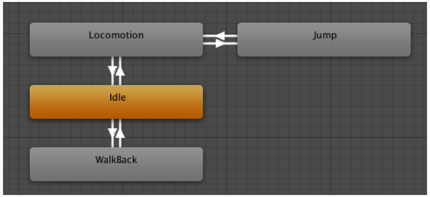
The problems of Training-Based AI

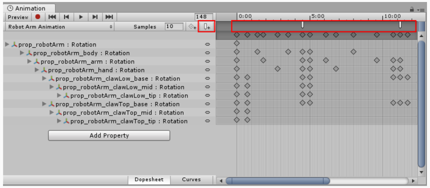

CAT
Good boy.
Hey, is that a robot?
Not just *any* robot. THIS is the current state-of-the-art in artificial intelligence.
We trained a 9-layered locally connected sparse autoencoder with pooling and local contrast normalization on a dataset of 10 million images.
It was trained for 3 days on a cluster of 1000 machines comprising 16,000 cores.
It can now recognize 22,000 different object categories with 15.8% accuracy.
15.8%?
Impressive, huh?
CAT
Impressive? Didn't you AI guys program a machine that defeated a world chess champion way back in 1997?
And didn't you AI guys build a computer system that kicked the respective asses of the two best *Jeopardy!* champions of all time?
What happened to 'HAL 9000 by 2001'? Where are our Series 800 Teminator robots?
You call this progress?!!
Sorry we annoyed you with our awesomeness.
CAT

All that glitters is not gold (*Æsop*).

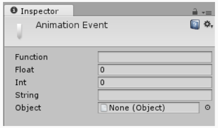
1. Critical (life-or-death) scenarios cannot rely on training-based AI, as they can be trained only over a few given circumstances and are not necessarily perfect.
2. Furthermore, the latter are harder to debug, as they often rely on the feed training data. Was the problem concerning the data or the "reward" function?
3. Hardcoded systems, if well-engineered, always guarantee 100% success (e.g. The Sims)! Still, this is not good for character predictability, and thus randomicity is required for improving over user experience.

Unity: Animation and Events





prop_robotArm : Rotation
prop_robotArm_body : Rotation
prop_robotArm_arm : Rotation
prop_robotArm_hand : Rotation
prop_robotArm_clawLow_base : Rotation
prop_robotArm_clawLow_mid : Rotation
prop_robotArm_clawLow_tip : Rotation
prop_robotArm_clawTop_base : Rotation
prop_robotArm_clawTop_mid : Rotation
prop_robotArm_clawTop_tip : Rotation



Animation Event

Function	
Float	0
Int	0
String	
Object	None (Object)

The animations that the characters play are often themselves controlled via a state machine.

- You can't jump while crouching, as you need to stand first.
- You can't run while jumping, as you need to land first.

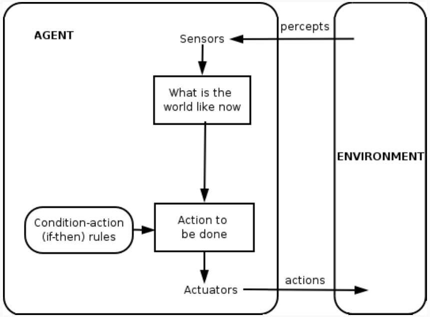
Any action can be decomposed into distinct sub-phases, and hence in different sub-steps.

- Run might have sub-states for speeding up and slowing down.

States can be associated to variables, to persist decision in time:

- When playing the reloading animation, set up a specific animation frame to trigger events within the main Game Engine (e.g., *Ammo Update*).

Game AI: World Metadata



The animations that the characters play are often themselves controlled via a state machine, which is now governed by specific events.

- Any Non Player Character (NPC) reacts to a perception of the world, be it a collision event or the effect of probing the environment in an automated way.
- After sensing the environment, the agnet shall make decisions on how to react to the world.

Some examples of considerations derived by probing the environment are the following. E.g., by raycasting towards specific objects of interest:

- If we hit the opponent, we might decide to change our behaviour (`ChaseEnemy`).
- If we identify an obstacle, we might decide to perform a specific contextual action(`JumpAcross`).

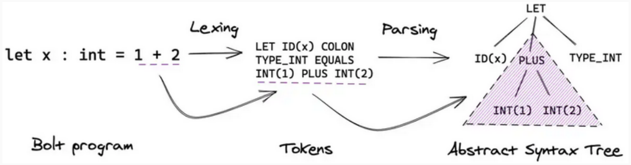
All of the former are examples of possible events

Finite-State Machine

A finite state machine is an abstract computational model organising a thought process by discretising it into subsequent states, while requiring that each system will always be in one and only one state at a time. These are currently used in the following domains:

Many events within a video-game are event-driven: they constantly wait for an external event (e.g., a collision, a timeout) to decide what to do next. Once the system handled the event, it goes back into a receptive state for handling the forthcoming event.

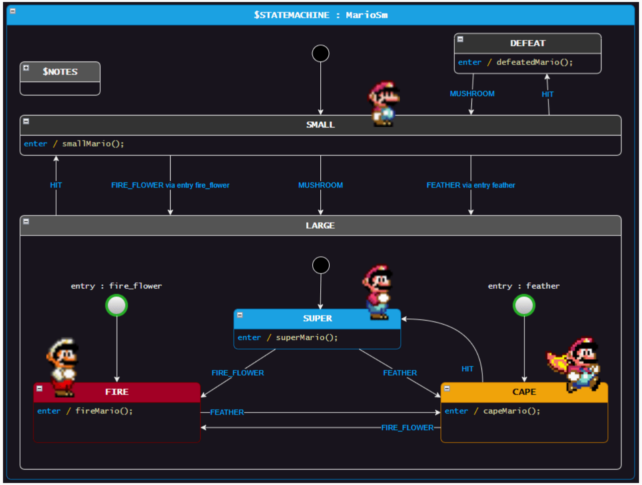
Programming Languages Compilers: Parsers are essentialy state machines emitting binary programs out of abstract syntax trees.



Software Engineering: You can use automated tools for first designing your own system through graphically representing the model, for then generating the associated code → We are going to leverage such tools!

Embedded Systems: Provides an efficient abstraction for handling hardware and OS events (e.g., interrupts) and describing in a graphical way what the system should do next. Given that they can debugged easily, they allow the user to better determine what causes the issue. This applies also to robotics, control systems, and communication protocols.

Example (1a)

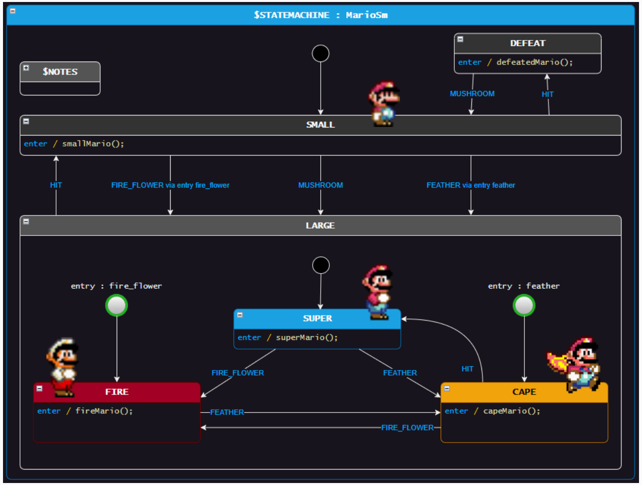


https://github.com/StateSmith/StateSmith

States

1. A state is characterised by a distinct response to the system to specific stimuli leading to a behavioural change.
2. States can perform actions (e.g. method invocation) on specific phases that might take time to compute: *enter* and *exit* are two examples. These are considered atomic operations.
3. States can be expressed in a hierarchy, where each state contains yet another state machine.

Example (1b)

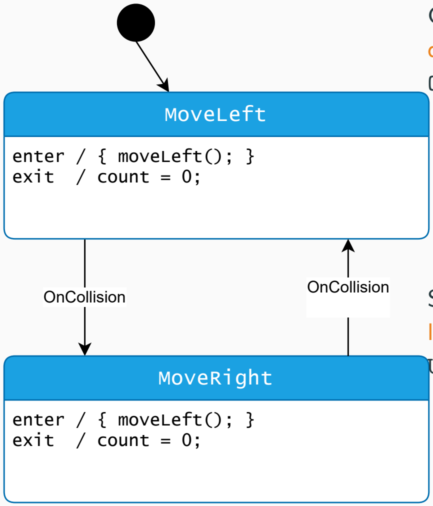


`https://github.com/StateSmith/StateSmith`

Events

1. An event changes the state of the system.
2. In a more general set-up, events might also pass information, which is then elaborated by the receiving state.
3. Events can be either internal (e.g., generated by the state-machine) or external (external stimulus).
4. These generate transitions among states.

Example (2a)



Goombas don't do much, and are mainly physical-event driven, that can be easily handed through `OnCollision⋆/OnTrigger⋆` methods:

1. When turned on (start), they start moving towards a specific direction (e.g., *left*).
2. When reaching an obstacle (OnCollision), they turn around (flipping *left* to *right* and viceversa).

Simple arcade enemies extend the former by considering other logical conditions, where events should be determined within `Update()`:

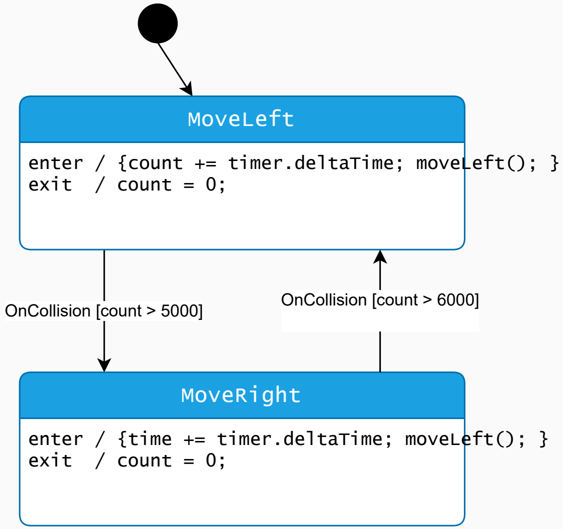
1. All enemies *start* in Idle mode (*state*).
2. When detecting an enemy (*event*), they go into Attack mode (*state*) until the enemy goes away (*event*).

Game AI

By now, you should have been able to set up a simple Unity GameObject controlled by a human player. You can structure an NPC similarly:

- At NPCScript's `Start()`, you initialise your state machine within the script controlling the player, as well as invoking the state machine's associated `Start()` method for event dispatching.
- Methods such as `OnTrigger⋆()/OnCollision⋆()` are directly generating events. After updating the state machine's variables with the information related to the handled event, you will call directly the `DispatchEvent()` method with the appropriate event type.
- By extending `Update()` to probe the environment for possible events to be dispatched to the state machine, we extend the model to also react to non-physical events.
- Similar considerations can be also provided for random chances for transiting to another state: Given the possibility of events E_1 (E_2 and E_3) to occur with probability p_1 (p_2 and p_3) over a specific state, we can then use a random number generator to generate a number p between 0 and 1 (`Random.NextDouble()`) and dispatch E_i if $\frac{p_1+\dots+p_{i-1}}{p_1+p_2+p_3} \leq p < \frac{p_1+\dots+p_i}{p_1+p_2+p_3}$.

Example (2b)

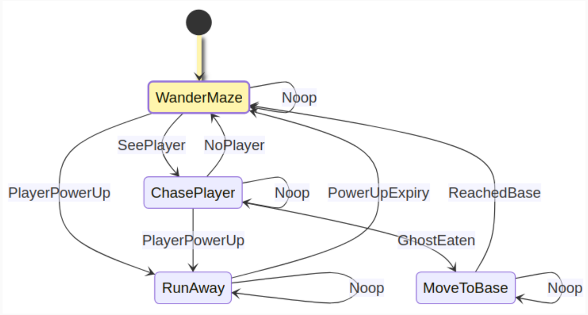

MoveLeft
enter / {count += timer.deltaTime; moveLeft(); }
exit / count = 0;
OnCollision [count > 5000]
OnCollision [count > 6000]
MoveRight
enter / {time += timer.deltaTime; moveLeft(); }
exit / count = 0;

Poorly designed transitions can lead to state oscillation:

1. A goomba can switch between the MoveLeft and MoveRight state intermittently if suddenly constrained by a narrow passage.
2. A moving enemy can move slightly across the boundaries of a threshold distance value, thus appearing undecided on what to do. This can also be a consequence of the enemy also changing its direction w.r.t. the player.

To solve this, we can add the idea of *hysteresis* into the state machine by adding a padding zone between the two threshold values triggering the events. This can also be solved by adding a minimum transition time: our solution does both! This requires extending the generated `DispatchEvent` method with time counting.

Example (3)

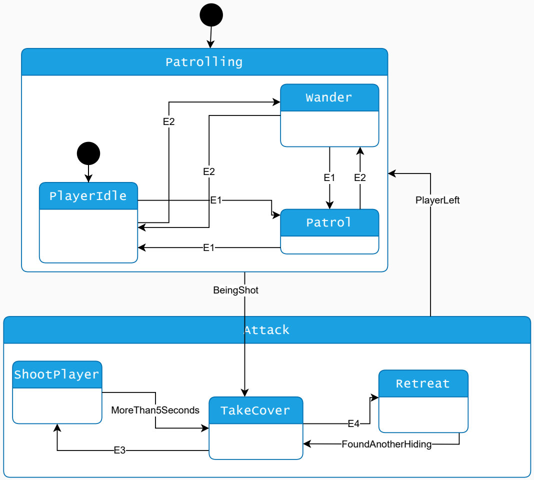


The statechart on the left represents a straightforward representation of a PacMan ghost. Statecharts allow us to encapsulate behaviour into specific states, and to separate the phase when events are detected (e.g., `Update()`) from their handling (`DispatchEvent()`). By exploiting specific tools:

1. We can use simulators to verify we can actually achieve the desired behaviour.
2. We reduce the amount of human error by just focussing on the implementation of the actual behaviour rather than coding the entire state machine.

NB: `Noop`-erations are used to enter/exit the same state on StateSmith: the usual UML semantics self-loops never exits for then entering.

Hierarchical State Machines: Example (4)



Hierarchical State Machines (HSM) provides states *nesting* inside other (H)SM.
The tool will automatically unpack the nested states for efficiency purposes by considering the following semantics:

- Transitions towards a *nesting* state will start from a starting state of interest.
- Transitions between a nested state and another state are inherited by all the nested states.

In this example, we represent E1, E2, E3, and E4 as randomly generated events with probability 50%, 50%, 80%, and 20% respectively.

Exercise: how to randomly generate events and associate those to Update?