Appendix A: The Halo AI as Layered Statecharts

Given in this appendix is the full Halo AI recreated as a layered statechart-based AI. The full description of the AI, module listing and description, and key features are outlined in Chap. 5. Verification and corrections from Chap. 8 have been performed and are shown here.

A.1 Sensors

Figure A–1: The AttackSensor.
A stateless Statechart. The listener broadcasts new `ev_PlayerSpotted(player)`, `ev_PlayerOutOfView(player)`, and `ev_PlayerKilled(player)` events.

**Figure A–2: The CharacterSensor.**

Internal listener creates `ev_NewCommand` events.

**Figure A–3: The CommandSensor.**
A Stateless Statechart. The listener generates the following events:

- `ev_GrenadeThrown(Grenade)`
- `ev_GrenadeExploded(Grenade)`
- `ev_HasGrenades`
- `ev_NoGrenades`

Figure A–4: The *GrenadeSensor*.

Figure A–5: The *HealthSensor*.
A stateless Statechart. The listener broadcasts the following events:

- ev_ItemSpotted(item)
- ev_ItemOutOfView(item)
- ev_ItemRemoved(item)
- ev_ItemAcquired(item)

Figure A–6: The *ItemSensor*.

A stateless Statechart. The listener will create and broadcast the following events:

- ev_ObstacleSpotted(Obstacle)
- ev_ObstacleRemoved(Obstacle)

Figure A–7: The *ObstacleSensor*.
ev_PositionUpdate and the cooldown event are generated internally. The cooldown prevents new event generation on every frame.

Figure A–8: The PositionSensor.

Internal listener generates ev_ShieldChange events

Figure A–9: The ShieldSensor.
Figure A–10: The VehicleSensor.
The Associated Class generates the following events:
- `ev_ClipEmpty`
- `ev_ReloadComplete`
- `ev_WeaponChange`

***NPCs don't run out of ammo in Halo, so tracking it is not necessary***

Figure A–11: The *WeaponSensor*. 
A.2 Analyzers

Figure A–12: The *EnemyAnalyzer*. 
Figure A–13: The *EnemyProximityAnalyzer*.
analyzing
no_grenades_near
[store.empty]
broadcastEvent(new ev_NoGrenadesNearby());
grenade_near
[!store.empty]
ev_GrenadeExploded[|inRange(grenade)|]
/remove(grenade)
ev_PositionChanged/updateStore()
check_store
[leaderCount>0]
leader_alive
ev_PlayerKilled[isLeader(player)]
[leaderCount==0]
declining
leaderCount--;
leaders_dead
broadcastEvent(new ev_MoraleLow());

Figure A-14: The GrenadeProximityAnalyzer.

leader alive
deciding
leaderCount--;
leaderCount==0
leaders dead
broadcastEvent(new ev_MoraleLow());

Figure A-15: The LowMoraleAnalyzer.
Figure A–16: The *SpecialEventAnalyzer*.

Figure A–17: The *SquadAnalyzer*.

Figure A–18: The *ThreatAnalyzer*.
Figure A–19: The ThreatCompilerAnalyzer.

Figure A–20: The VehicleAnalyzer.
Figure A–21: The *VehicleProximityAnalyzer*.
A.3 Memorizers

Contains 4 data structures:
- visible allies
- allies out of view
- visible enemies
- enemies out of view

Figure A–22: The CharacterMemorizer.

Commands are stored in a fifo queue, and are removed when a command is read.

Figure A–23: The CommandMemorizer.
Figure A–24: The *ObstacleMemorizer*.

Figure A–25: The *VehicleMemorizer*. 

215
A.4 Strategizer

Figure A–26: The *Brain*.
A.5 Deciders

Figure A–27: The CombatDecider.
Figure A–28: The *FleeDecider*. 

```plaintext
Figure A–28: The *FleeDecider*.
```
Figure A–29: The *IdleDecider*. 
Figure A–30: The SearchDecider.

Figure A–31: The SelfPreservationDecider.
A.6 Executors

findClosestSafeSpot looks at the stored list of area threat locations (e.g., grenades, burning vehicles), and finds the closest safe spot.

Figure A–32: The ClearAreaExecutor.
tracking
fleeing
broadcastEvent(new ev_Move(determineFleePoint());
ev_FleeAll
ev_MoveComplete/
broadcastEvent(new ev_StopMove())
determineFleePoint will get characterMemorizer.VisibleEnemies and characterMemorizer.EnemiesOutOfSight, then use them to determine progressively more distant safe spots to flee to.

Figure A–33: The FleeAllExecutor.
determineFleeTarget takes in a list of nearby enemies and returns a position that is the chosen spot to which to flee.

Figure A–34: The *FleeNearbyExecutor*. 

223
Figure A–35: The **ItemExecutor**.

Figure A–36: The **MeleeCombatExecutor**.

Being "in melee range" means the enemy is close but not necessarily in striking distance.
Figure A–37: The *RangedCombatExecutor*. 
Figure A–38: The SearchExecutor.
Figure A–39: The TakeCoverExecutor.

Figure A–40: The UseItemExecutor.
This Statechart uses 4 methods to effect vehicle combat. First, the selectTarget(Players[]) selects the most appropriate attack target. Then, the isTargetInFront method quickly determines if the target is in a cone in front of the NPC. If not, the attackRunStartSpot method will select an appropriate spot from which to attack. Finally, the calculateAttackRunTarget will choose a spot behind the target, in an attempt to run down the target.

Figure A–41: The VehicleCombatExecutor.

ev_WanderTimer is generated internally when the wander timer goes off.

Figure A–42: The WanderExecutor.
A.7 Coordinators

Figure A–43: The MovementCoordinator.
A.8 Actuators

Figure A–44: The *GrenadeActuator*.

Figure A–45: The *ItemActuator*.
Figure A–46: The *MeleeActuator*.

Figure A–47: The *SoundActuator*.
Figure A–48: The *RangedCombatActuator*.
idle

moving

ObstacleMemorizer.getObstacles(); pathfind();
ev_Run

[moveComplete()&&atTarget()]/broadcastEvent(new ev_RunComplete())
[moveComplete()&&!atTarget()]/broadcastEvent(new ev_RunFailed())
ev_NewObstacleSpotted
ev_ObstacleRemoved

Figure A–49: The RunActuator.

vehicle

idle

pathfinding

obstacleMemorizer.getObstacles(); vehiclePathfind();... sembarkSuccessful()

not_firing

firing

ev_VehicleOpenFire/
vehicle.fire()
ev_VehicleCeaseFire/
vehicle.ceaseFire();

Figure A–50: The VehicleActuator.

233