Statechart modelling of NPC behaviour

Kevin Wyckmans

Universiteit Antwerpen

Overview

Introduction

Case study: Tank Wars

- Modelling Game AI
- Time
- Code generation

Agent based spreading of diseases

Introduction

As the realism in games increases, so does the demand for more sophisticated AI. This leads to more complex code. We can abstract this to a higher level:

- Define *REACTIONS* for NPC's on game *EVENTS*

⇒ Statecharts

Overview

Introduction

Case study: Tank Wars

- Modelling Game AI
- Time
- Code generation

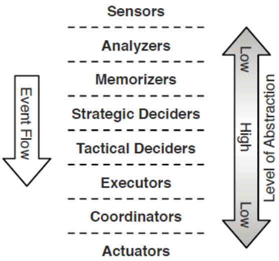
Agent based spreading of diseases

Structure of our models

Based on paper written by Jrg Kienzle, Alexandre Denault and Hans Vangheluwe: Model-Based Design of Computer-Controlled Game Character Behavior

- Character uses sensors to detect events.
- Reacts using actions or actuators
- Describe transformation of sensor input to actuator output using simple components.
  - Structure defined by class diagrams
  - behaviour defined by statecharts
- Communicate using asynch. events.

Different abstraction levels



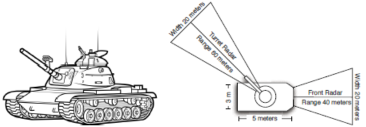
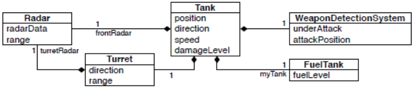
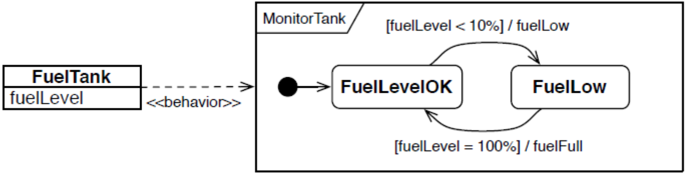


Fig. 2. Tank and it's Abstraction



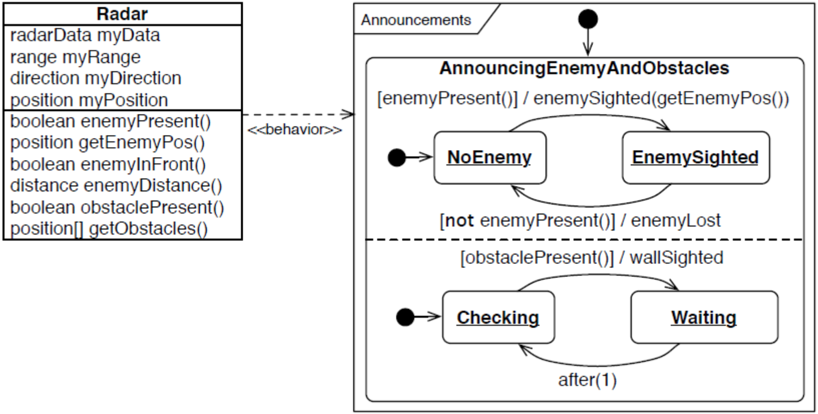
Sensors

- State of tank and it's components evolve.
- Explicitly model generation of events using state diagrams
  - Attach to class that contains all the state necessary



Universiteit Antwerpen

More complex example...



Radar

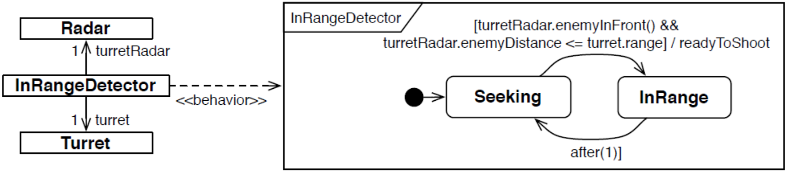
radarData myData
range myRange
direction myDirection
position myPosition

boolean enemyPresent()
position getEnemyPos()
boolean enemyInFront()
distance enemyDistance()
boolean obstaclePresent()
position[] getObstacles()

<<behavior>>
Announcements
AnnouncingEnemyAndObstacles
[enemyPresent()] / enemySighted(getEnemyPos())
NoEnemy
EnemySighted
[not enemyPresent()] / enemyLost
[obstaclePresent()] / wallSighted
Checking
Waiting
after(1)

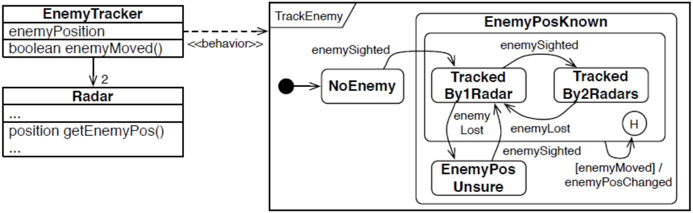
Analyzers

- Some events depend on multiple tank components
    - Enemy in range?



Memorizers

- Make descisions based on events from the past
- Occurances of events can be remembered using attributes or statecharts

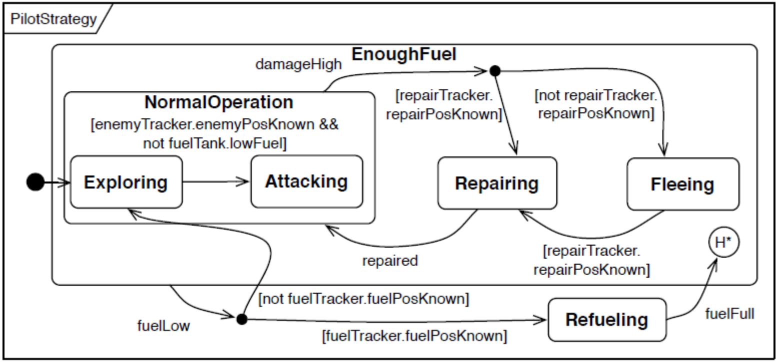


- Sometimes elaborate data structures necessary (maps, ...)

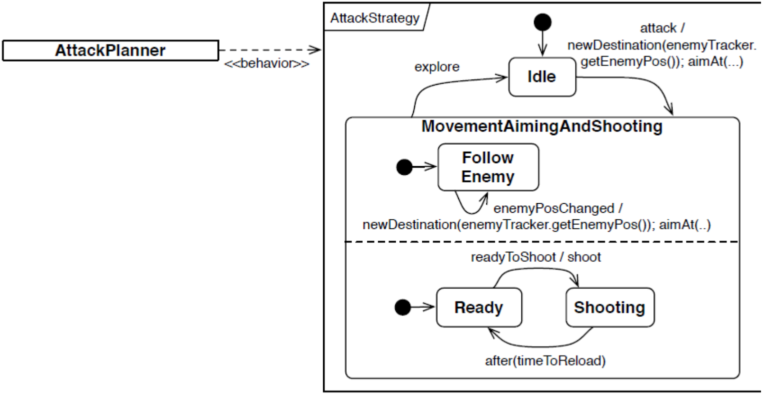
Strategical and Tactical Deciders

- Strategical Decider: Decides on what goal to achieve
- Tactical Decider: How to achieve that goal
  - This can be very complex!
  - Each strategy should have a corresponding planner.

Strategical decisions


PilotStrategy
EnoughFuel
damageHigh
NormalOperation
[enemyTracker.enemyPosKnown && not fuelTank.lowFuel]
Exploring
Attacking
[repairTracker. repairPosKnown]
[not repairTracker. repairPosKnown]
Repairing
Fleeing
repaired
[repairTracker. repairPosKnown]
H*
[not fuelTracker.fuelPosKnown]
fuelLow
Refueling
fuelFull
[fuelTracker.fuelPosKnown]

Tactical decisions

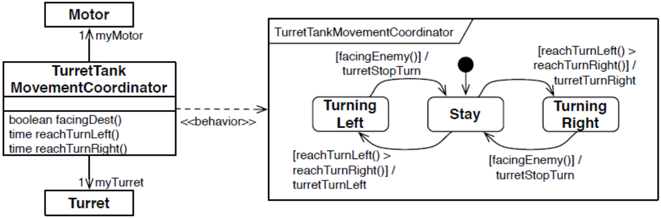

AttackPlanner
<<behavior>>
AttackStrategy
attack / newDestination(enemyTracker. getEnemyPos()); aimAt(...)
explore
Idle
MovementAimingAndShooting
Follow Enemy
enemyPosChanged / newDestination(enemyTracker.getEnemyPos()); aimAt(..)
readyToShoot / shoot
Ready
Shooting
after(timeToReload)

Executors

- Maps the decisions of tactical deciders to events that the actuators understand
  - Convert waypoints into directions, ...
  - Can be made more complex by taking physics into account

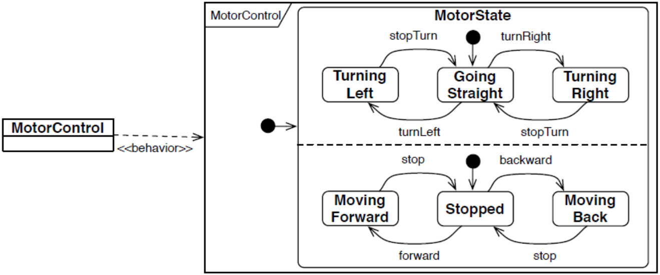
Coordinators

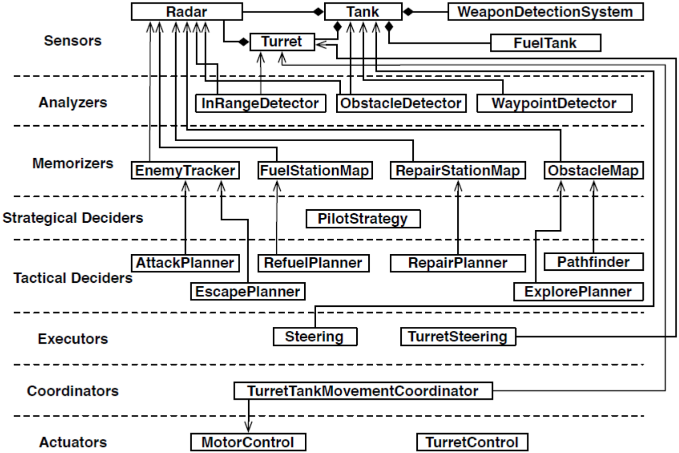
- Executors map events directly to actuators ⇒ Might lead to inefficient and even incorrect behaviour
- Example: Turning of turret while attacking


Motor
1 myMotor
TurretTank MovementCoordinator
boolean facingDest()
time reachTurnLeft()
time reachTurnRight()
1 myTurret
Turret
<<behavior>>
TurretTankMovementCoordinator
[facingEnemy()] / turretStopTurn
[reachTurnLeft() > reachTurnRight()] / turretTurnRight
Turning Left
Stay
Turning Right
[reachTurnLeft() > reachTurnRight()] / turretTurnLeft
[facingEnemy()] / turretStopTurn

Actuators

- At this level of abastraction: very simple actuators
- Each actuator is a seperate control class




Sensors
Radar
Tank
WeaponDetectionSystem
Turret
FuelTank
Analyzers
InRangeDetector
ObstacleDetector
WaypointDetector
Memorizers
EnemyTracker
FuelStationMap
RepairStationMap
ObstacleMap
Strategical Deciders
PilotStrategy
Tactical Deciders
AttackPlanner
RefuelPlanner
RepairPlanner
Pathfinder
EscapePlanner
ExplorePlanner
Executors
Steering
TurretSteering
Coordinators
TurretTankMovementCoordinator
Actuators
MotorControl
TurretControl

Time Slicing

- Time-slicing vs. continuous time
- Statecharts purely eveny based
  - On model level: Time is continuous
    - Modelling freedom
    - Symbolic analysis
    - Simulation
    - Reuse
  - This has to be mapped to the target simulation
  - If the slice is small enough, the approximation is acceptable

Bridging the gap

- Every slice a function with updated data is called
- Fill objects with new data
- map data to events using sensors ⇒ starting here, propagation/triggering of events done entirely in statechart
- If all events finished or just before slice ends, return the necessary commands

From statecharts to code

- Use *atom*^{3} to model statecharts
- Use a statechart compiler to generate code

Overview

Introduction

Case study: Tank Wars

- Modelling Game AI
- Time
- Code generation

Agent based spreading of diseases

Overview

- Correlates to Roland's project
- Visualisation of agent based spreading of an infectious disease
- Comparable to the system described above.
  - The same abstraction levels are adequate.
  - Sensors: eyes, Actuators: legs, . . .

Scientific possibilities

- Visualise the behaviour of people using various algorithms
- Use probabilities to introduce randomness
  - Most people run away from sick persons
  - A small amount tries to help them (doctors?)
  - A hospital (cfr. refuel station) has a probability of curing a sick person

Game-design possibilities

- If a person dies in a hospital, he becomes a zombie
- A subset of healthy people can be soldiers
- Very dynamic and complex system
- One person can be a player controlled character

Thank you for your attention.

Questions?