Model-based Design of Computer-Controlled Game Character Behavior

Jörg Kienzle, Alexandre Denault, Hans Vangheluwe

McGill University, Montreal, QC H3A 2A7, Canada
{Joerg.Kienzle, Alexandre.Denault, Hans.Vangheluwe}@mcgill.ca

Abstract. Recently, the complexity of modern, real-time computer games has increased drastically. The need for sophisticated game AI, in particular for Non-Player Characters, grows with the demand for realistic games. Writing consistent, re-useable and efficient AI code has become hard. We demonstrate how modeling game AI at an appropriate abstraction level using an appropriate modeling language has many advantages. A variant of Rhapsody Statecharts is proposed as an appropriate formalism. The Tank Wars game by Electronic Arts (EA) is used to demonstrate our concrete approach. We show how the use of the Statecharts formalism leads quite naturally to layered modeling of game AI and allows modelers to abstract away from choices between, for example, time-slicing and discrete-event time management. Finally, our custom tools are used to synthesize efficient C++ code to insert into the Tank Wars main game loop.

1 Introduction

Recently, global sales of the world's computer game industry have grown higher than those of the movie industry. Consequently, there is a growing demand for technology which supports rapid, re-usable game development accessible to by non software experts. Computer games can be roughly classified into two categories: turn-based games (such as board games, adventures, and some role playing games) and real-time games (such as action or arcade games, and real-time strategy games). The kind of *artificial intelligence* found in computer games is different for turn-based and real-time games.

Board games are usually computerized versions of existing board games. Real board games typically require 2 or more players, but in a computerized version the computer can play the opponent. A good example of a board game that has seen many successful computerized implementations is *Chess* [6]. In turn-based games and particularly in board games, an artificial intelligence component that plans the moves of a player typically uses advanced search algorithms and heuristics to evaluate many possible future game situations. It then chooses as the current move the one that maximizes the likelihood of winning the game in the future. Timing is not that critical. Since the game is turn-based, the state of the game does not change until a player makes a move. Usually, waiting several seconds for an artificial intelligence component to make a move is acceptable.

Real-time games are very different in nature. The state of the game changes continuously (or in tiny increments), and the screen is continuously updated to present the new game state to the player. Modern computer games usually provide at least 30 frames-per-second updates. In real-time games (with the exception of real-time strategy games) the player usually controls one character (or a small number of characters), and plays within a game environment against a set of computer controlled characters (or in multiplayer games against characters controlled by other players).

In such games, the term *artificial intelligence* is used to designate the algorithms that specify the behavior of computer-controlled game characters, often also called *non-player characters* (NPC). The ultimate goals is to make the NPCs' own actions and reactions to game events seem as intelligent and natural as possible. For example, a guard protecting a building might walk back and forth in front of the main door. If ever he hears shots nearby, he should not simply continue this behavior, but for instance seek cover and call for backup. In its simplest form, such AI can be specified with scripts or rules that specify the NPC's behavior case by case. More realism can be achieved if the NPC has the ability to analyze a situation and evaluate different options, taking into account even the game history.

We believe that the specification of such advanced real-time AI should not be done within a programming language, but at a higher level of abstraction using visual modeling formalisms. Since the main focus of the models is to define reactions to game events, an event-based formalism seems to be the most natural choice. We decided to use our own variant of *Rhapsody statecharts* [5], a combination of state diagrams and class diagrams, for our experiments.

Our paper is structured as follows. Section 2 describes our approach to modeling game AI, and explains the details by designing a game AI that controls the behavior of a tank. Section 3 shows how we used our model to generate code that executes within the EA Tank Wars environment. Section 4 presents some related work and section 5 discusses the benefits of our approach and concludes.

2 Modeling Game AI

In games or simulations, a character perceives the environment through his senses or *sensors*, and reacts to it through actions or *actuators*. For instance, a character might observe an obstacle with his eyes, and subsequently decide to turn left. Our AI modeling framework follows this control-inspired philosophy. The transformation from sensor input to actuator output is described by means of simple components. Each component's structure is modeled by a class, and its behavior by a statechart. The main mechanism of communication between the components is the asynchronous sending/receiving of events. This lowers the coupling between components and hence makes reconfiguration and reuse easier. In some situations, a component may also synchronously invoke an operation of another component.

The architecture of our AI models is described in Fig. 1. The first level contains components that represent the *sensors* that allow the character to observe the environment as well as its own state. The sensors filter the abundant information and send events of interest on to the next levels. The second level contains components that *analyze* or correlate the events from individual sensors, which might lead to the generation of further events. The *memorizer* components keep track of the history of events. The *strategic decider* components are conceptually at the highest level of abstraction. They have to decide on a strategy for the character based on the current state and memory. At the next level, the *tactical deciders* plan how to best pursue the current strategy. The *executors* then translate the decisions of the tactical components to low-level commands according to the constraints imposed by the game or simulation. *Coordinator* components understand the inter-relationships of actuators and might refine the low-level commands further. Finally, the *actuators* perform the desired action.

To illustrate the power of our approach, we show in the following sections how we modeled the AI of a computer-controlled tank.

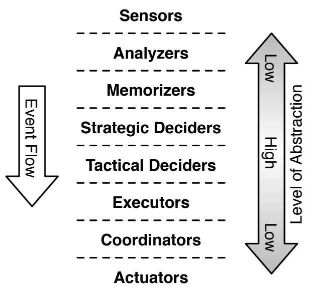

Fig. 1. AI Model Architecture

2.1 Modeling the State of a Tank

A tank is a heavy armored fighting vehicle carrying guns and moving on a continuous articulated metal track. When developing a model of a real-world object such as a tank, the modeler abstracts away certain details depending on the context in which the model is going to be used.

Some games actually model game objects such as vehicles and their physical interactions with the environment in great detail. These games are typically called *simulators*, such as flight simulators, helicopter simulators and tank simulators. Simulating the physics of a tank requires a detailed model of the physical components of a tank (physical shape, material, mass) and equations describing the physical interactions of these components.

Our interest in this paper is to reason about the behavior of a tank pilot. Therefore we can model a tank at a much higher level of abstraction (see Fig. 2). In the particular game that our AI is going to be playing, a tank has a given physical size, approximated by a bounding rectangle. The gun is mounted on a rotating turret anchored in the middle of the tank. A tank also has a set of sensors that relay information about the state of the tank and the surrounding environment to the pilot. The instruments in the cockpit tell the driver the position of the tank, in which direction the tank is facing, what speed it is going at, and at what angle the turret is currently positioned. A fuel indicator shows the current fuel level of the tank, and a status indicator reports on the current damage. Finally, two radars, one mounted in the front of the tank, and the other one on the turret, scan the environment for enemies and obstacles. The tank has also an advanced weapon detection system, which informs the pilot when the tank is under attack, and from what position the enemy attack is originating.

The above mentioned state of a tank can naturally be modeled using class diagrams as shown in Fig. 3. Each sensor of the tank, such as the radar, can be modeled as a stand-alone class. The composition association is then used to connect the different components together to form the complete state of a particular tank. The advantage of using hierarchical composition is easy to see: models of

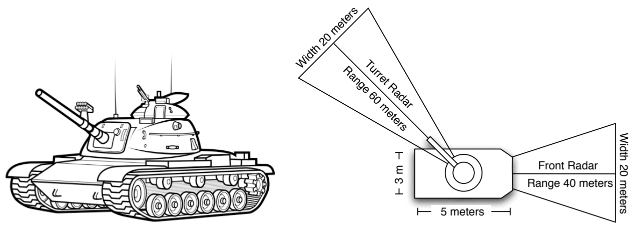

Fig. 2. Tank and it's Abstraction

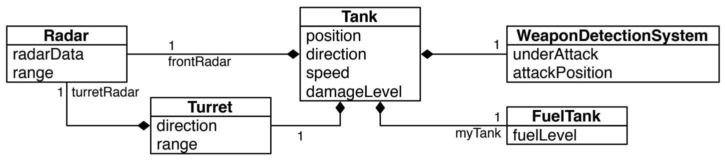

Fig. 3. Modeling the State of a Tank with Class Diagrams

tanks with different components, for example with 2 turrets, can easily be constructed by combining the individual components in different configurations.

2.2 Sensors – Generating Important Game Events

During a game or simulation, the state of the tank and the states of its components evolve (according to the game rules or laws governing the simulation). As mentioned in the introduction, a tank pilot (or a computer player) pursues a specific high level goal and performs actions that work towards the achievement of that goal. High level goals usually remain the same as long as no significant changes in the tank's state or in its environment occur.

We suggest to explicitly model the generation of significant events using state diagrams. The state diagrams are attached to the class that has access to all the state needed to generate the event, either by inspecting the values of its own attributes, or by looking at attributes of other classes associated by composition relationships.

A simple example is shown in Fig. 4. The *FuelTank* class encapsulates an attribute that stores the current fuel level of the tank. Fuel is essential for the tank to function, but the exact fuel level is not of great importance. Hence we abstract from the continuous fuel level to two discrete states, *FuelLevelOK* and *FuelLow*. Only when the fuel is low, the tank pilot should take appropriate measures. We can model the generation of a *fuelLow* event in case the fuel level crosses a certain threshold by attaching a state diagram to the *FuelTank* class. Note that this simple behavior introduces hysteresis: once the fuel level drops below 10%, the *FuelLow* state is entered, to only be exited once the level reaches 100% again.

A more complicated example is shown in Fig. 5. In this case, the *Radar* component wants to signal *EnemySighted* and *EnemyLost* event when an enemy enters/exits the radar surveillance zone. This behavior is described in the first orthogonal component of the statechart *Announcements*. Analyzing the radar data for enemy presence, and calculating the enemy position are both operations that

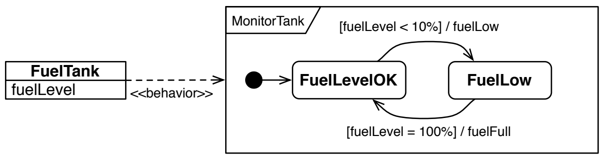

Fig. 4. Generating *FuelLow* and *FuelFull* Events

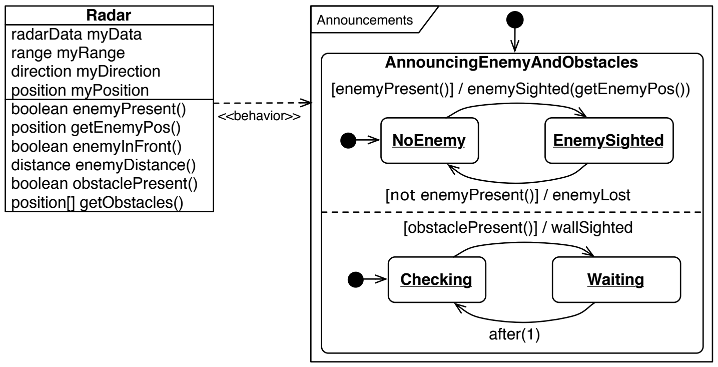

Fig. 5. Generating Events based on Simple Computations

require a small computation. They can be modeled as simple operations such as `getEnemyPos()` attached to the *Radar* class. The state diagram attached to the *Radar* class can use these operations to trigger the transition that sends the desired *EnemySighted* and *EnemyLost* events. The orthogonal *AnnounceObstacles* component also shown in Fig. 5 performs similar event generation for detected obstacles.

2.3 Analyzers – Correlating Sensor Events

Some significant events can only be detected or calculated based on the state of several tank components. For instance, to determine if the enemy is in range, information from the turret as well as the turret radar is needed. The *InRangeDetector* state diagram shown in Fig. 6 takes care of this. While in the *Seeking* state, if the front radar ray of the turret radar detects an enemy, and the distance is smaller than the turret range, then the *ReadyToShoot* event is sent.

2.4 Memorizers – Modeling Memory

A tank pilot does not only react to current events, but also makes decisions based on events/state from the past. In order to remember interesting state or events for future strategical decisions, we need to add state to the model that acts as the tank pilot's "memory".

Occurrences of events can be remembered using boolean or enumeration fields, or states in a statechart. An example of the latter is shown in Fig. 7, which depicts an *EnemyTracker* class that contains an `enemyPosition` field that remembers at what position the enemy has last been seen, even if the enemy is not within range of one of the radars anymore.

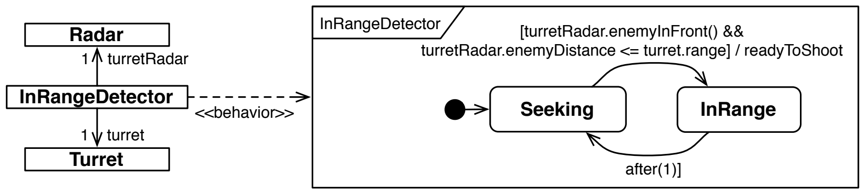

Fig. 6. Generating Events based on the State of Several Components

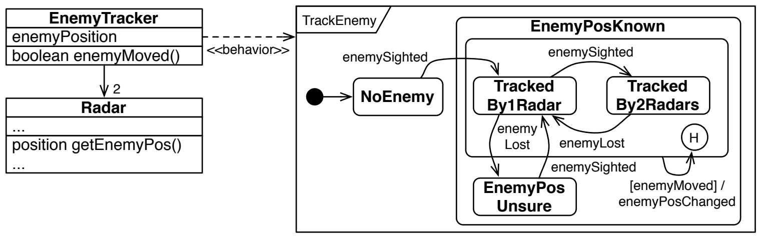

Fig. 7. Remembering the Position of the Enemy

While the enemy is in range of at least one of the radars (# received *enemySighted* events > # received *enemyLost* events), the `enemyMoved` operation compares the enemy position of the *EnemyTracker* with the position obtained from the two radars. If the positions differ by a significant amount, the stored position is updated and an *enemyPosChanged* event is sent.

Remembering complex state, for instance geographical information, is less trivial, and usually requires the construction of an elaborate data structure that stores the state to be remembered in an easy-to-query form. This is done in the *ObstacleMap* class shown in Fig. 8. It reacts to the *WallDetected* events sent by the radars and updating the map data structure accordingly. The actual algorithm is not shown here, but abstracted within the `updateMap()` operation.

2.5 Strategic Deciders – Deciding on a High-Level Goal

Now that we have the event generation (based on environment sensors, current state of the tank and memory) in place, it is possible to model the high level strategy of the tank pilot. This is depicted in Fig. 9. At the highest level of abstraction, a tank pilot switches between different operating modes based on events. He starts in *Exploring* mode, and switches to *Attacking* mode once the enemy position is known (and there is still enough fuel). If at any point in time the sustained damage is too high, then, if the location of the repair station is known, he switches to *Repairing* mode. Otherwise, *Fleeing* is the best strategy. In the event that the fuel is low, if the location of the fuel station is known, the tank pilot chooses to switch to *Refueling* mode. Otherwise, it is best to continue *Exploring*, hoping to find a fuel station soon. When the fueltank is full, the pilot switches back to whatever he was doing before he was interrupted.

The mode changes are announced by sending corresponding events: when *Exploring* is entered, the *explore* event is sent, when *Attacking* is entered, the *attack* event is sent, etc.

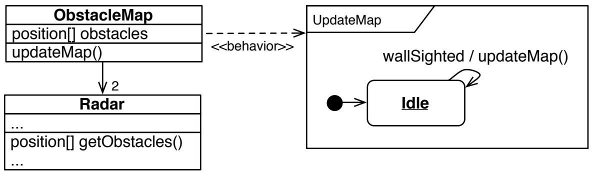

Fig. 8. Creating a Map of the World

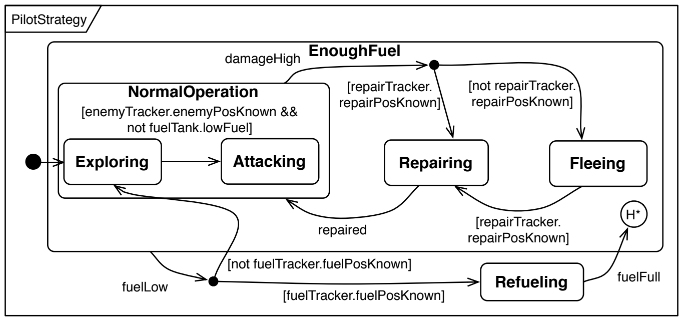
PilotStrategy
EnoughFuel
damageHigh
NormalOperation
[enemyTracker.enemyPosKnown && not fuelTank.lowFuel]
Exploring
Attacking
[repairTracker. repairPosKnown]
[not repairTracker. repairPosKnown]
Repairing
Fleeing
repaired
[repairTracker. repairPosKnown]
H*
[not fuelTracker.fuelPosKnown]
fuelLow
[fuelTracker.fuelPosKnown]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

Fig. 9. The Tank Pilot Strategy

2.6 Tactical Deciders – Planning how to Achieve the Goal

The high-level goals sent by the pilot strategy component have to be translated into lower-level commands that can be understood by the different actuators of the tank, such as the motor and the turret. This translation is not trivial, since it can require complex tactical planning decisions to be made. In addition, the planning should take into account the history of the game, i.e. consult the memorizers for important game state or events that happened in the past.

Each strategy of the pilot should have a corresponding *planner* component. Fig. 10 illustrates how the *AttackPlanner* decides to carry out an attack: whenever the tank is ready to shoot, a *shoot* event is sent. The movement strategy is also simple: the planner chooses to move the tank to the position of the enemy. Whenever the enemy position changes, it sends out a *newDestination* event.

The *Pathfinding* component, shown in Fig. 11, knows how to perform obstacle avoidance. It translates the *newDestination* event into a list of waypoints by analyzing the current world information obtained from the *Map*. The *Pathfinding* component then announces the first waypoint by sending an event. Whenever the tank reaches a waypoint, the next waypoint is announced.

2.7 Executors – Mapping the Decisions to Actuator Commands

The executors map the decisions of the tactical deciders to events that the actuators can understand. The mapping of events is constrained by the rules of the game or simulation. There is typically one executor for each actuator.

In our case the *Steering* component shown in Fig 12translates the waypoints into events that the *MotorControl* understands. Every second, depending on whether the waypoint is ahead of, left of, right of or behind the tank, the corresponding command event is sent to the *MotorControl*. A more sophisticated *Steering* component would take into account the dynamics of the tank such as speed, mass, and acceleration.

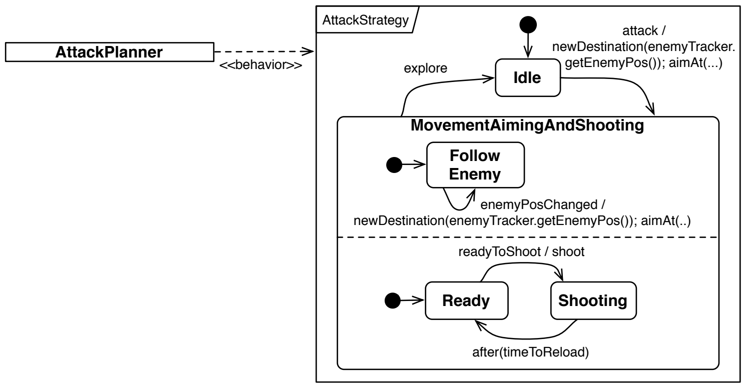
AttackPlanner
<<behavior>>
AttackStrategy
attack / newDestination(enemyTracker.getEnemyPos()); aimAt(...)
explore
Idle
MovementAimingAndShooting
Follow Enemy
enemyPosChanged / newDestination(enemyTracker.getEnemyPos()); aimAt(..)
readyToShoot / shoot
Ready
Shooting
after(timeToReload)

Fig. 10. Attack Movement Strategy

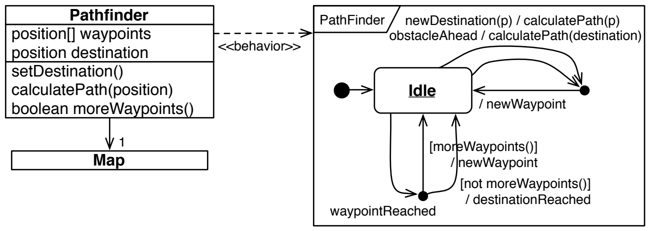

Fig. 11. Pathfinding

2.8 Coordinators – Resolving Undesired Actuator Interactions

For modularity and composability reasons, executors individually map tactical decisions to actuator events. This mapping can result in inefficient and maybe even incorrect behavior when the effects of actuator actions are correlated. In such a case it is important to add an additional *coordinator* component that deals with this issue.

For example, while attacking, the turret should turn until it is facing the enemy tank and then shoot. However, the optimal turning strategy depends on whether the tank itself is also turning or not. Fig. 13 illustrates a *Turret-TankMovementCoordinator* class that performs this coordination step. The calculations required to determine if it is faster to turn right or turn left based on the current turning decisions of the motor are done in the operations `reachTurnLeft()` and `reachTurnRight()`.

2.9 Actuators – Signaling the Action to the Game

At our level of abstraction, the tank actuators are very simple. A tank pilot can decide whether to advance or move the tank backwards at different speeds, and whether to turn left or right. Likewise, a turret can be turned left or right, and shots can be fired at different distances. Finally, commands can be given to refuel or repair, if the tank is currently located at a fuel or repair station. We suggest to model each actuator as a separate *Control* class.

Fig. 14 shows the *MotorControl* class, an actuator that controls the movement of the tank. The state diagram shows how the motor reacts to *turnLeft*,

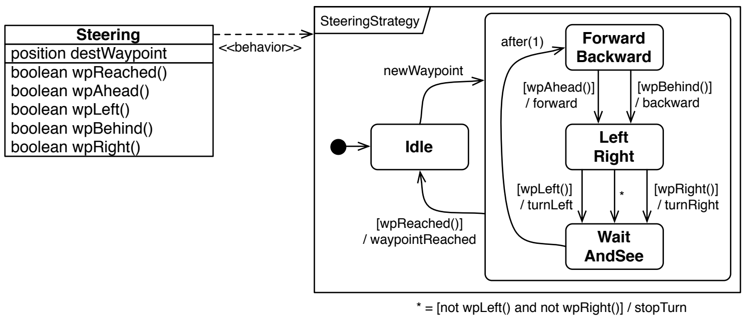

Fig. 12. Steering the Tank

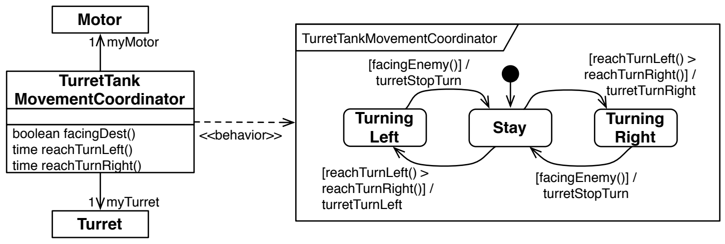
Motor
1 myMotor
TurretTank MovementCoordinator
boolean facingDest()
time reachTurnLeft()
time reachTurnRight()
1 myTurret
Turret
<<behavior>>
TurretTankMovementCoordinator
[facingEnemy()] / turretStopTurn
[reachTurnLeft() > reachTurnRight()] / turretTurnRight
Turning Left
Stay
Turning Right
[reachTurnLeft()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

Fig. 13. Coordinating and Controlling the Turret

turnRight, stopTurn, *forward*, *backward* and *stop* events. How this action is finally executed within the game or simulation is going to be discussed in section 3.

2.10 Tank AI Model Summary

The detailed architecture that shows all the components of our tank AI is depicted in Fig. 15. Most communication is done using events, and hence the individual components are only loosely coupled. Only when conceptually necessary, i.e. when a component's functionality depends on data from another component, synchronous communications must occur. In this case, the dependency between the involved classes is shown with directed associations.

Fig. 16 shows a possible sequencing of events in case the *PilotStrategy* component decides to attack. The *AttackPlanner* (concurrently) sets a new destination and tells the turret to aim at the enemy. The *Pathfinder* calculates new waypoints by consulting the *ObstacleMap* and instructs the *Steering* component to move towards the first waypoint. The *Steering* component instructs the Motor to move in the appropriate direction. Simultaneously, the *TurretSteering* component instructs the turret to turn by an absolute angle to face the enemy. The turning is coordinated with the tank movement by the *TurretTankMovementCoordinator*. In our example, the tank is moving left (maybe to avoid an obstacle), and the turret turns right to compensate and then stops.

The following events could interrupt the current movement at any time:

- A *waypointReached* event sent by the *WaypointDetector* component when it detects that the tank reached its current waypoint causes the *Pathfinder* to announce the next waypoint.

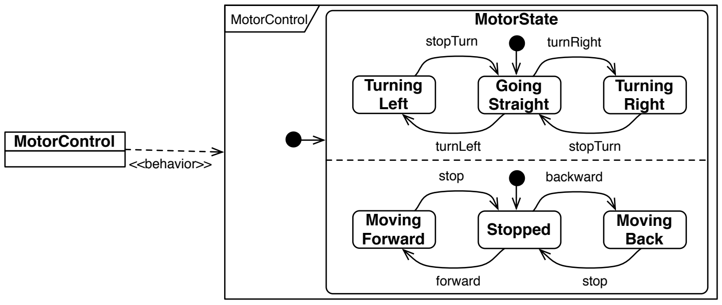

Fig. 14. The *Motor* Actuator

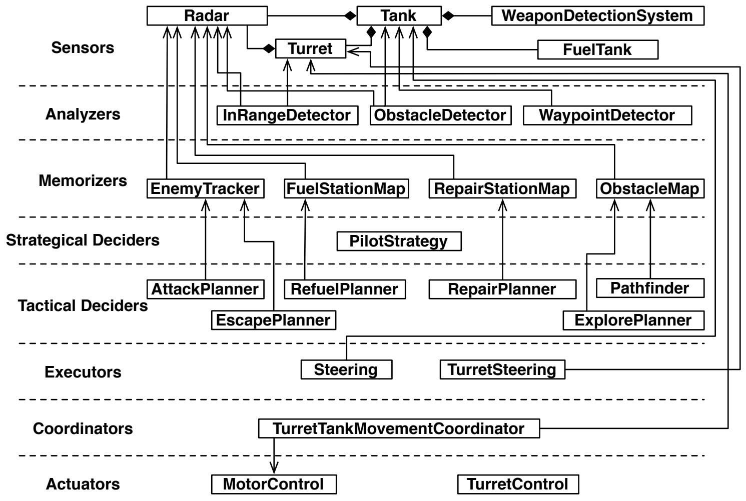

Fig. 15. The Tank AI Components

- An *obstacleAhead* sent by the *ObstacleDetector* causes the *Pathfinder* to calculate a new path and announce the first waypoint.
- A *fuelLow* event sent by the *FuelTank* causes the *PilotStrategy* to transition into the refueling state and send the *refuel* event, which causes the *RefuelPlanner* to announce the position of the fuel station as the new destination.
- Similarly, a *damageHigh* event sent by the *Tank* can cause the pilot to decide to repair.

3 Mapping to an Execution Platform

3.1 The EA Tank Wars Simulation Environment

In 2005, Electronic Arts announced the EA Tank Wars competition [2], in which Computer Science students compete against each other by writing artificial intelligence (AI) components that control the movements of a tank. EA released a simulation environment written in C++, prepared to compile on Windows,

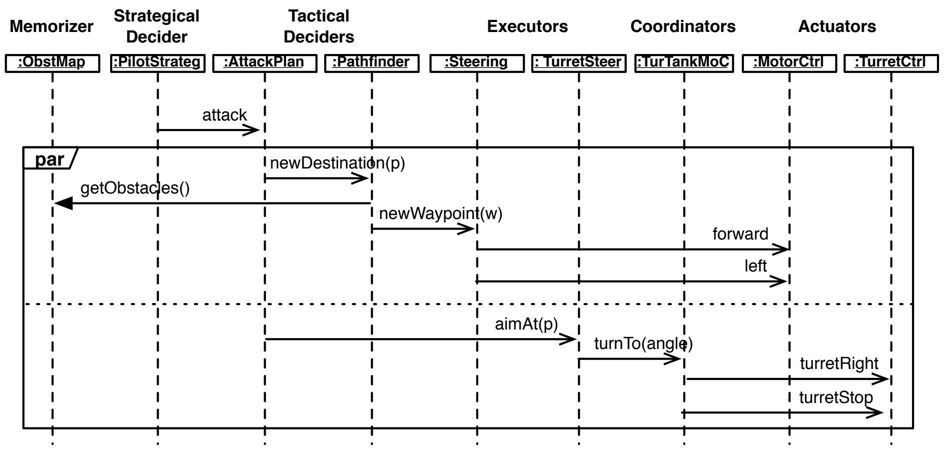

Fig. 16. Possible Event Sequence in Case of an Attack

Linux and MacOS X, in which, two tanks, both controlled by an AI component, fight a one-on-one battle in a 100 by 100 meter world. In Tank Wars, a tank is equipped with sensors and actuators identical to our example model of section 2 (see also Fig. 2). During the simulation, an animation shows the moving tanks, their radar ranges and their state.

The Tank Wars simulation environment is *time-sliced* (as opposed to discrete-event). Every time slice, the AI component of the tank is given the current state of the world as seen by the tank sensors. The AI then has to decide whether to change the speed of the tank, whether to turn, whether to turn the turret, whether to fire and how far, and whether to refuel or repair, if possible. Each turn lasts *50 milliseconds*. If the AI does not make a decision when the time limit elapsed, the tank does not move during that time slice.

3.2 Time-slicing vs. Continuous Time

The simulation in Tank Wars is built on a time-sliced architecture. Every 50ms, the new state of the environment is sent to the AI component. Statecharts on the other hand are purely event-based. At the modeling level, as well as when the model is simulated, time is continuous, i.e. infinite time precision is available. There is no time-slicing: a transition that is labeled with a time delay such as `after(t)` means that the transition should fire exactly after the time interval `t` has elapsed, `t` being a real number. Continuous time is most general, and is most appropriate at this level of abstraction for several reasons:

- Modeling freedom: The modeler is not unnecessarily constrained or encumbered with implementation issues, but can focus on the logic of the model.
- Symbolic analysis: Using timed logic it is possible to analyze the model to prove properties.
- Simulation: Simulation can be done with infinite accuracy (accuracy of real numbers on a computer) in simulation environments.

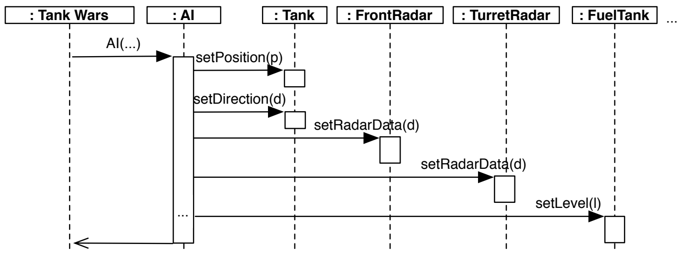

Fig. 17. Converting Time Sliced Execution to Events

- Reuse: Continuous time is the most general formalism, and can therefore be used in any simulation environment.

When a model is used in a specific environment, actual code has to be synthesized, i.e. the continuous time model has to be mapped to the time model used in the target simulation. In games that are event-based such a mapping is straightforward. This is however not the case for Tank Wars, in which an approximation has to take place: the synthesized code can execute at most once every time-slice. Fortunately, if the time slice is small enough compared to the dynamics of the system to be modeled (such as the motion of a tank), the approximation is acceptable and the resulting simulation close to equivalent to a continuous time simulation.

3.3 Bridging the Time-Sliced – Event-Driven Gap

In order to use event-based reasoning in a time sliced environment, a bridge between the two worlds has to be built. In Tank Wars, at every time slice, the framework calls the C++ function `static void AI (const TankAIInput in, TankAIInstructions & out)` of an AI object. We implement the bridge between the time-sliced game environment and our statechart model in this function.

In section 2.1, we modeled the state of the sensors in separate classes. The `in` parameter of the AI function contains a data structure that describes the current state of all sensors. The function proceeds by storing the new sensor states in the appropriate objects (see Figure 17).

The mapping to events is done at the level of the sensor objects according to the attached statecharts. After the operation updated the state of the sensor, the guards in the statechart are evaluated, and the corresponding event is fired, if any. For instance, according to the statechart shown in Fig. 4, the execution of the `setLevel` operation of the *FuelTank* might generate a *fuelLow* event in case the fuel level sinks below 10%.

From then on, propagation of events and triggering of actions entirely done within the statechart formalism. After all events have been processed, or at least just before the 50 ms deadline expires, the state in the actuator objects is copied into the `out` struct of the `AI` function and returned to the Tank Wars simulation.

3.4 ATOM3 and Code Generation

To validate our approach, we built our tank model in our AToM^3 visual meta-modeling and model transformation environment [1]. We compiled the model

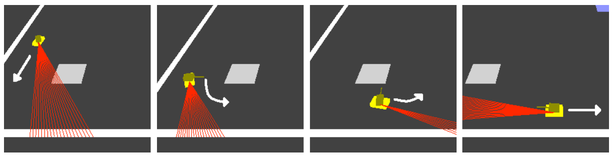
Fig. 18. Wall Encounter Execution Trace

into C++ code with our own custom-built Statechart compiler. After inserting this code into the Tank Wars game (in the `AI` function), realistic behavior is observed as shown in Fig. 18. The figure shows a trace of a scenario where a tank encounters a wall, initiates turning until the wall is no longer in the line of sight, and finally continues on its way.

4 Related Work

The use of visual modeling environments is not new to the gaming industry. Also known under the name of *Visual Scripting Languages*, finite state machines and other formalisms have been used to model various features of games, including cinematics and story narratives [8]. The main objective of developing such systems is to offload work from the programmers to the game designers and the animators, allowing them to participate to the development of the game without requiring any programming or scripting knowledge [4].

More interesting is the use of modeling environment to define the behavior of agents, as proposed by *Simbionic* and its toolset which allow a developer to describe the behavior of intelligent agents using finite state machines [3]. Although similar to our approach, the *Simbionic* toolset represents states as actions, transitioning from one action to another solely through the use of conditions and guards. In addition, the toolset functions exclusively in a time-slice fashion, abstracting time as simple clock ticks.

Viusal modeling environments can also be found in commercial engines. The Unreal Engine 3 [9] includes *UnrealKismet*, a visual scripting system, which provides artists and level designers the freedom to design stories and action sequences for non player characters within a game without the need for programming. One key feature of *UnrealKismet* is the support for hierarchy of components, which makes it possible to structure complicated behavior descriptions nicely. The difference with our approach is that the models in *UnrealKismet* essentially describe the decision making steps of an AI algorithm graphically. Our approach does not model the control flow explicitly. The behavior emerges based on the components that listen for and react to events.

Also worth mentioning is *ScriptEase* [7], a textual tool for scripting sequences of game events and reactions of non player characters. Although it doesn't use a visual formalism, *ScriptEase* introduces a pattern template system – a library of frequently used sequences of events – that allows designers to put together complex sequences with little programming.

5 Discussion and Conclusion

In this section, we situate our efforts into the broader context of Model-Based Design and highlight the benefits of this approach. The core idea of Model-

Based Design is to explicitly *model* the structure and behavior of systems under study. Such models can be described at different *levels of abstraction* or detail as well as by means of different *formalisms.* The particular formalism and level of abstraction chosen depends on the background and goals of the modeler as much as on the system modeled.

Working at the Appropriate Level of Abstraction In general, the process of abstraction can be considered a type of transformation that preserves some invariant properties (usually behavioral) of the system. In the case of our Tank Wars example, several types of abstraction take place. First of all, there is the explicit *layering* of levels of abstraction. At the lowest levels (most detailed data, closest to the physical entities/game engine) are sensors and actuators. At the highest level is the strategic planning level. Intermediate levels help bridge the information gap between these levels. Different levels of abstraction are crossed quite naturally by means of event aggregation and synthesis.

Abstraction is also applied to data. Sensors filter the large amounts of data and propagate only salient events to higher abstraction layers. As is common in object-oriented design, an abstraction is made by choosing only relevant properties to be modeled as object attributes.

Appropriate Formalism and Visual Notation Orthogonal to the choice of model abstraction level is the selection of suitable formalisms in which the models are described. The choice of formalism is related to the abstraction level, the intended audience, and the availability of solvers/simulators/code generators for that formalism. In the case of our Tank Wars example, a variant of *Rhapsody Statecharts* were chosen as the main formalism. This formalism allows for modular description of both structure (in the form of Class Diagrams) and behavior (in the form of associated Statecharts) of the different components described above. Statecharts have been used extensively to model behavior of reactive systems. It is hence no surprise that this formalism is a natural choice to model the types of modern computer games we are interested in. The formalism has an intuitive visual notation which makes it suitable for use by non software experts. It also allows us to almost perfectly encapsulate the individual components.

During the modelling (and analysis and possibly simulation) stage of development, Statecharts allow us to ignore implementation details such as whether the game engine uses time-slicing or event-scheduling time management. This issue, albeit very important, is taken care of transparently by the model compiler. Thus, the game AI modeler is no longer burdened with making coding detail decisions, but only with higher-level choices. This shows the power of working at an appropriate level of abstraction, using appropriate formalism(s). Note that complexity does of course never disappear. In Model-Based Design, *accidental complexity* is kept to a minimum hidden and is factored out and encoded in formalism transformation models (the model compiler in this case).

Enhanced Modularization There is the more detailed de-composition, within each abstraction layer, of structure and behavior into easily identifiable components. These components either correspond to physical entities (such as a *Turret*) or to conceptual entities such as a *AttackPlanner*. This high degree of modularity allows both for independent development and understanding of components. While working on a specific component within a well-defined abstraction level, a developer is maximally focused on the task at hand.

Enhanced Evolution and Reuse The abstraction layers we presented are commonly found in a variety of modern computer games and provide a conceptual framework within which models can easily be formulated and (re-)used. For instance, the *AttackPlanner* could be easily reused within a game in which a computer-controlled knight has to decide how to attack an enemy soldier.

The elegant breakup into loosely coupled components makes it easy to evolve an AI by simply replacing an existing components with a more sophisticated component that provides similar functionality. For instance, the performance of our tank AI could be enhanced by using a better *Pathfinder*. An enhanced *AttackPlanner* could hide behind an obstacle and ambush the enemy.

Finally, the loose coupling makes it possible to create AI for tanks with different sensors and actuators by removing or adding individual components. For instance, a tank could have a better radar, or just one radar, or 2 turrets.

References

1. Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSyM)*, 3(3):194–209, August 2004. `DOI: 10.1007/s10270-003-0047-5`.
2. Electronic Arts. EA Tank Wars. http://www.info.ea.com/company/company-tw.php, 2005.
3. Daniel Fu and Ryan T. Houlette. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.
4. Sunbir Gill. Visual Finite State Machine AI Systems. Gamasutra: http://www.gamasutra.com/features/20041118/gill-01.shtml, November 2004.
5. David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the UML). *LNCS*, 3147:325 – 354, 2004.
6. Monty Newborn. Deep blue's contribution to AI. *Ann. Math. Artif. Intell*, 28(1-4):27–30, 2000.
7. C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, K. Waugh, M. Carbonaro, and J. Siegel. A Pattern Catalog For Computer Role Playing Games. In *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*, pages 33 – 38. Eurosis, August 2005.
8. Christopher J.F. Pickett, Clark Verbrugge, and Felix Martineau. (P)NFG: A Language and Runtime System for STructured Computer Narratives. In *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*, pages 23 – 32. Eurosis, August 2005.
9. Unreal Technology. The Unreal Engine 3. http://www.unrealtechnology.com/html/technology/ue30.shtml, 2007.