

Model-Based Design of Game AI

Alexandre Denault, Jörg Kienzle and Hans Vangheluwe
School of Computer Science, McGill University
Montréal, Canada, H3A 2A7
email: {adenau,joerg,hv}@cs.mcgill.ca

KEYWORDS

Modern Computer Games, Model Compilers, Rhapsody Statecharts, Game AI.

ABSTRACT

The complexity of modern computer games has increased drastically over the last decades. The need for sophisticated game AI, in particular for Non-Player Characters (NPCs) grows with the demand for realistic games. Writing meaningful, consistent, modular, re-useable and efficient AI code is not straightforward. In this article, we suggest to model rather than to code game AI. A variant of Rhapsody Statecharts is proposed as an appropriate modelling formalism. The Tank Wars game by Electronic Arts (EA) is introduced to demonstrate our approach in a concrete setting. By modelling a simple AI, it is demonstrated how the modularity of the Rhapsody Statecharts formalism leads quite naturally to layered modelling of game AI. Finally, our Statechart compiler is used to synthesize efficient C++ code for use in the Tank Wars main game loop.

STATECHARTS

Statecharts were introduced by David Harel in 1987 [Har87] as a formalism for visual modelling of the behaviour of reactive systems. A full definition of the STATEMATE semantics of Statecharts was only published in 1996 [HN96]. More recently, with the introduction of UML 2.0, the Rhapsody semantics as described in [HK04] is more tuned to the modelling of software systems. In this article, we will use a sub-set of the Rhapsody Statecharts semantics.

At the heart of the Statechart formalism is the notion of discrete *states* and the transition between. Statecharts are a discrete-event formalism which means it takes a timed sequence of discrete *events* as inputs and produces a timed sequence of discrete as output. Internally, the system transitions between discrete states due to either external or internal events. This leads to a piecewise constant state trajectory inside the system as illustrated in Figure 1. In Figure 2, a simple model is shown with two states *start* and *end*. The small arrow pointing to *start* denotes that state as the *default* initial state. If the system is in state *start* and it receives *event*,

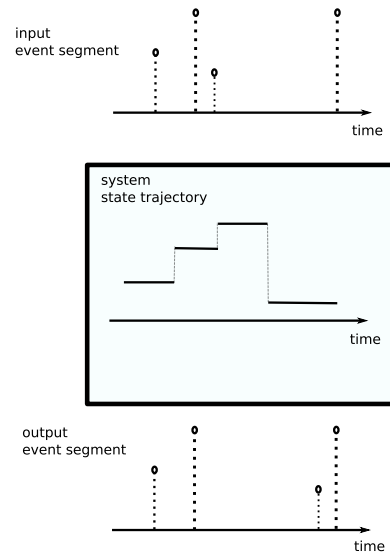


Figure 1: Discrete-Event In/State/Out Trajectories

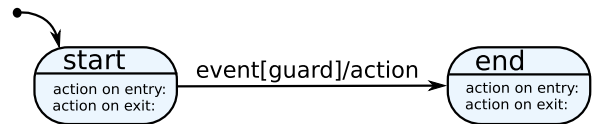


Figure 2: Statechart Basic Transition

and condition *guard* evaluates to *true*, the transition to state *end* is taken and the side-effect *action* is executed. Additionally, *entry*/*exit* actions are executed whenever a state is entered/exited. All of the parts of *event*[*guard*]/*action* are optional. The special event *after*(Δt) indicates that a transition will be taken autonomously after Δt time units (unless interrupted earlier). Statecharts add hierarchy to the above basic notion of state automata. Figure 3 shows a composite state *s1* with several nested states. Initially, the system will start in nested state *s11* as at the top level, *s1* is the default state and within *s1*, *s11* is the default state. To understand the nesting, when in a state such as *s11*, upon arrival of an event such as *f*, an outgoing transition is looked which is triggered by event *f*. This lookup is performed traversing all nested states, from the inside outwards. The first matching transition is taken. This approach keeps the semantics deterministic despite the seemingly conflicting *f* trigger on transitions to both *s13* and *s2*. When in state *s12*, there is

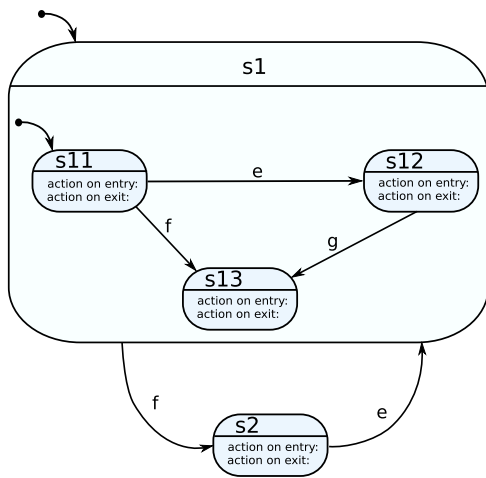


Figure 3: Hierarchy in Statecharts

```
def process(EventQueue):
    while EventQueue not empty:
        evt = EventQueue.pop()
        if CurrentState reacts to evt:
            t = transition reacting to evt
                whose guard evaluates to True
            compute states that will be
            exited and entered
            as a result of taking t
            next = last state to be entered
            perform exit actions, trigger
            and enter actions
            set CurrentState = next
```

```
def processAll(self):
    for obj, evtQ in self.objectQueues.items():
        o.process(evtQ)
```

Figure 5: Processing Concurrent Objects

deleted asynchronously or synchronously (in which case they are similar to remote method calls). We will mostly use asynchronous message passing. To support concurrency between objects, our Statechart compiler will give each object an event queue. All object queues will be processed fairly as shown in the pseudo-code in Figure reffig:alg.

MODELLING GAME AI

Tank Wars

In 2005, Electronic Arts announced the EA Tank Wars competition ¹, in which computer science students compete against each other by writing artificial intelligence (AI) components that control the movements of a tank. In Tank Wars, two tanks, both controlled by an AI, fight a one-on-one battle in a 100 by 100 meter world. Each tank has a set of environment sensors, that sense information about the tank's remaining life points and fuel, its position, the direction in which it is facing, it's front radar information (what objects – walls or enemy – are located within 40 meters in front of the tank), if a tank is hit and from where the shot was fired, and if the tank is standing on top of a fuel or health station. In addition, a tank has a rotating turret with a direction and a second, more powerful radar, mounted on the turret, that detects obstacles at distances up to 60 meters (see Figure 6). The Tank Wars simulation is *time-sliced* (as opposed to discrete-event). Every time slice, the AI component of a 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 will not move during this time slice.

¹www.info.ea.com/company/company-tw.php

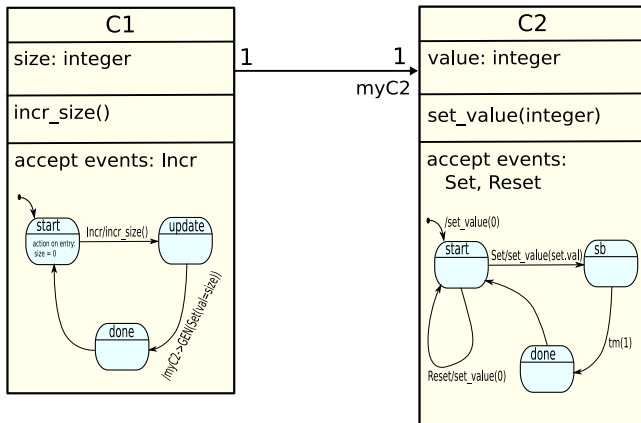


Figure 4: Modelling Structure and Behaviour

no conflict and event f will take the system to state s2. When in state s2, event e will take the system to state s1. As the latter is a composite state, the system will transition (after executing s1's entry action) to the s11, the default state of s1. In addition to hierarchy, Statecharts add orthogonal components and broadcast communication to state automata. In our implementation these features will not be used. The most interesting feature of Rhapsody statecharts is that it allows for a combined description of structure and behaviour of objects. This is achieved by adding Statechart behaviour descriptions to UML Class Diagrams as shown in Figure reffig:classdiagram. The behaviour of individual objects (class instances) is described by the class' statechart. For conceptual clarity we require that methods in a class will only have local effects. They can only change the object's attributes. All external effects must be modelled in the Statechart. This allows for a clean separation of externally visible, reactive, timed behaviour from internal (computation) details. Objects communicate by means of a GEN action which sends an event to a target object as shown in Figure reffig:classdiagram (myC2->GEN(Set(size=2))). Events can be han-

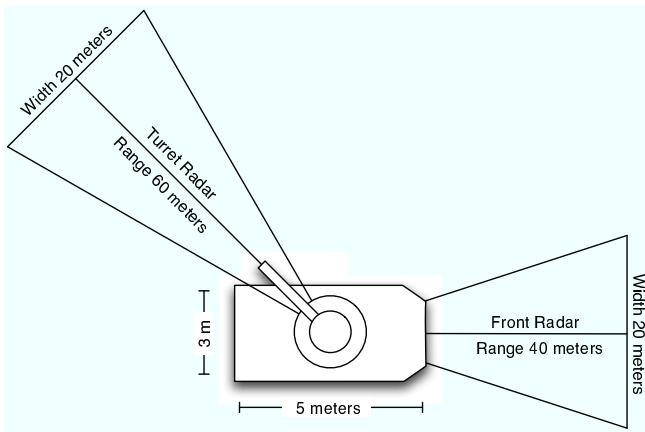


Figure 6: Tank Input

Time-slicing vs. Continuous Time

As mentioned above, 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 such as SMV (reference).
- **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.

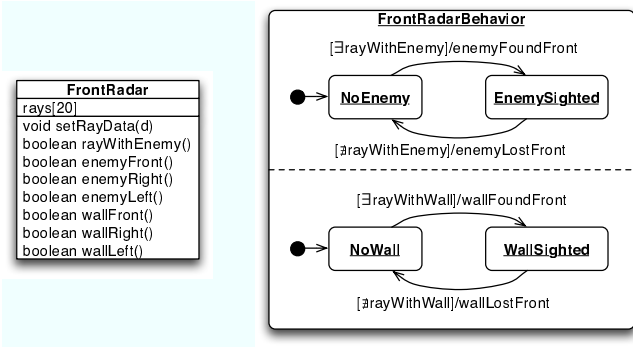


Figure 8: Input Event Generation

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 a previous section, we described the semantics of Rhapsody Statecharts where each object's behavior is described in a separate Statechart. We exploit the modularization capability offered by object-orientation and define objects that encapsulate the perceived state of the world. One object is defined for each sensor. At every time slice, the Tank Wars framework calls the C++ function `static void AI (const TankAIInput in, TankAIInstructions & out)` of the AI object. The `in` parameter contains a struct that describes the state of all environment sensors. The function proceeds by storing the new sensor states in the appropriate objects (see Figure 7).

The mapping from time-sliced to event-based simulation is done at the level of the sensor objects. If a significant change occurred in the environment, then the sensor should generate a corresponding event. What kind of changes are significant and should therefore be signalled with an event depends entirely on the AI. A simple AI might only react to coarse grained environment changes, whereas a more complex AI might want to react to each slightest change.

The idea is illustrated in this paragraph using the `FrontRadar` object. The class definition is given in the left hand side of Figure 8. The actual radar information obtained at each time slice is very accurate. In fact, each radar sends out 20 rays, which reflect when they hit an obstacle. During each turn, the reflection data of every ray is made available to the AI, and the AI stores the complete ray data in the front radar object by calling `setRayData()` as shown in Figure 7.

Important events for a simple AI are the sighting of an enemy or a wall, or the fact that an enemy or wall is no longer on the radar. The creation of these events is illustrated in the `FrontRadarBehavior` Statechart in Figure 8. Whenever event handling takes place (we chose in our implementation to process events at every time slice), the `rayWithEnemy` condition is evaluated by calling the corresponding function provided by the

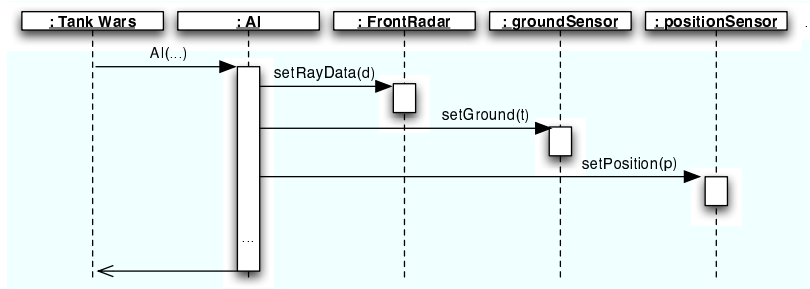


Figure 7: Converting Time Sliced Execution to Events

class, and if the condition evaluates to true, then the transition fires and a corresponding event is generated. The events from the sensors are then broadcast to other statecharts at higher levels of abstraction. Sophisticated AIs might have analyzer objects, for instance objects that keep track of the wall configuration in the world, or objects that track the enemy. These objects react to sensor events and update their state. In case an important situation is detected, for instance `tankEnteredDeadEnd`, an appropriate event is created and broadcast.

At the topmost level of abstraction is the object that defines the high-level strategy of the tank. This object might define different modes or priorities, for instance *following the enemy*, or *looking for fuel*. Modes changes or other high-level command events such as `fleeFromEnemy` are sent on to coordinator and planner objects that take care of the detailed execution of these high-level commands. Finally, actuator objects update their state when receiving low-level events such as `advanceFullSpeed`. After all events have been processed, the state in the actuator objects is copied into the `out` struct of the AI function and returned to the Tank Wars simulation.

The event propagation through the different levels of abstraction is illustrated in Figure 9.

MODELLING TANK WARS

In the sequel, we show a few small parts of our simple Tank Wars AI model. All Statecharts are modelled our AToM³ visual modelling environment [dLVA04]. Note that default states are not indicated by a small arrow but are rather denoted in green. Figure 10 shows the Statechart for the main tank behaviour. The tank is highly conservative and toggles between `Searching` and resting (`Stopped`) mode. This strategy conserves fuel and turns out to pay off. The autonomous behaviour gets interrupted when a wall is encountered in which case the tank turns. Figure 11 shows wall detection at work. While in `SearchingForWall` mode, the default state is `WallUnknown`. From any state in `SearchingForWall`, a parametrized `UPDATE` triggers a transition to an appropriate state reflecting the tank's knowledge about the proximity to a wall. Figure 12 demonstrates how the different fuel levels are

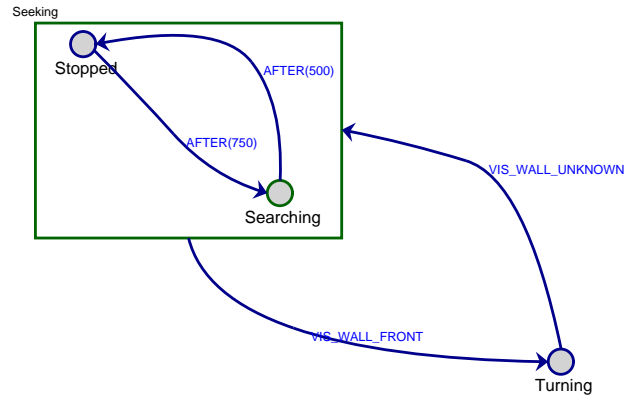


Figure 10: Conservative Tank Behaviour

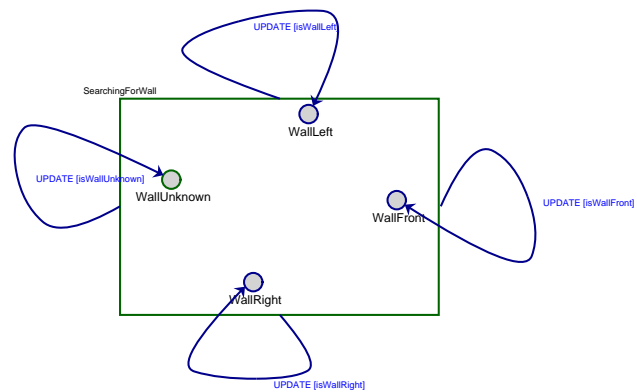


Figure 11: Wall Detection

death with by means of different modes. Similarly, Figure 13 shows how overall tank health is monitored. It also shows how the tank is ultimately destroyed when health becomes negative. It will be clear from the above that judicious use of state nesting combined with concurrent objects allows for concise and easy to understand models.

We have compiled the above models into C++ code with

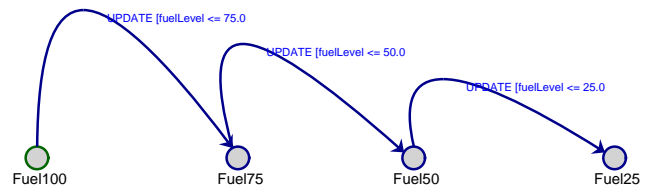


Figure 12: Fuel Level Monitoring

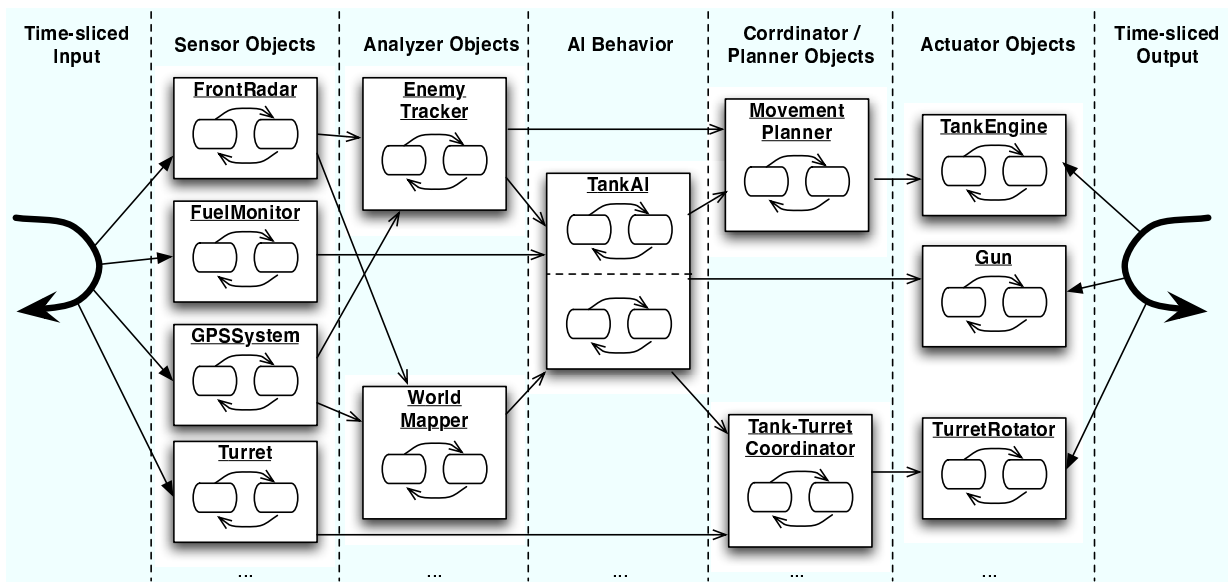


Figure 9: Event Propagation

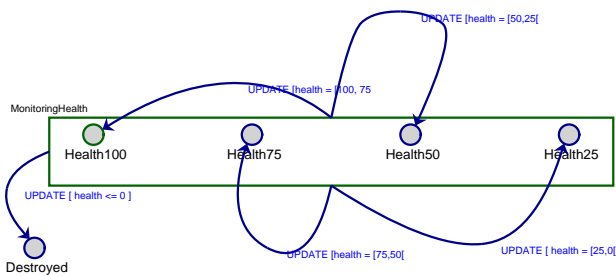


Figure 13: Tank Health Monitoring

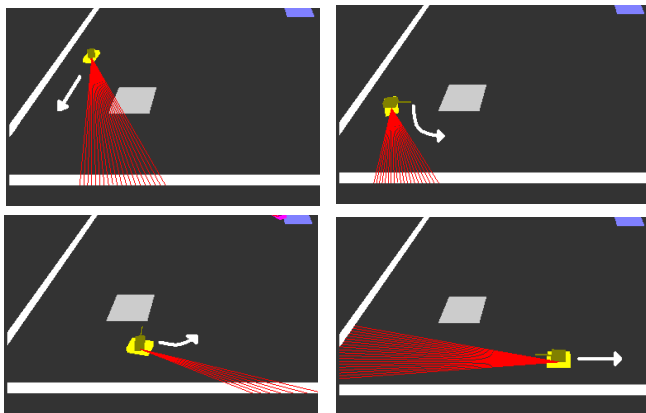


Figure 14: Wall Encounter Execution Trace

our Statechart compiler. After inserting this code into the Tank Wars game (in the AI function), realistic behaviour is observed as shown in Figure 14. 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.

ACKNOWLEDGEMENTS

Huining Feng built the DChart visual modelling environment, simulator and compiler [Fen04]. David Meunier re-used the visual modelling environment and built

the first prototype of our Rhapsody Statecharts compiler (generating Python code). Both of these efforts formed the basis for the work described in this paper. Jörg Kienzle and Hans Vangheluwe gratefully acknowledge partial support for this work through their National Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant.

REFERENCES

- [dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM³. *Software and Systems Modeling (SoSyM)*, 3(3):194–209, August 2004. DOI: 10.1007/s10270-003-0047-5.
- [Fen04] Thomas Huining Feng. DCharts, a formalism for modeling and simulation based design of reactive software systems. M.Sc. dissertation, School of Computer Science, McGill University, February 2004.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.
- [HK04] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). *LNCS*, 3147:325 – 354, 2004.
- [HN96] David Harel and Amnon Naamad. The statechart semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.