Toward High-Level Reuse of Statechart-based AI in Computer Games

Christopher Dragert, Jörg Kienzle, Clark Verbrugge
McGill University
Montréal, Québec
christopher.dragert@mail.mcgill.ca, {joerg.kienzle, clark.verbrugge}@mcgill.ca

ABSTRACT

Designing an interesting AI for a computer game is a complex undertaking, providing motivation to reuse portions of successful AIs. Here we advocate a *layered Statechart-based AI* as a modular approach that simplifies reuse. We analyze Statechart interactions and communications with respect to AI design, and propose an interface for Statechart-based *AI-modules* that summarizes interactions.

Reuse is accomplished by adding and removing modules to a new AI, largely enabled through event-renaming to ensure coherence with the interfaces. We describe an approach to module composition using *functional groups*, which allow for the encapsulation of high level behaviours (e.g., fleeing or exploring). This enables a designer to compose new AIs by assigning high-level behaviours. Additionally, the interface describes interactions with the game at-large, leading naturally to portability between games and even implementation languages. Finally, we look ahead to the requirements for a tool that would implement these ideas.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse Models*

General Terms

Software Reuse

Keywords

AI, Computer Games, Reuse, Statecharts

1. INTRODUCTION

In modern computer games, non-player controlled characters (NPCs) provide interaction, competition, and immersion for the player. Powering NPCs with a complex AI can increase player fun and lead to increased sales. While some game components, such as graphics engines, are highly reused, AIs are often written from scratch, wasting development time on reimplementing basic behaviours. Reusing AI is thus an appealing option, since it allows effort to spent on interesting behaviours rather than basic ones.

The ease of AI reuse is largely dependent on the underlying formalism. Unlike the texture mapped triangle in graphics, there is no generally accepted basic unit of AI. Current industry practice involves little AI reuse in part due to this lack of an agreed upon AI format. Akin to software reuse, a formalism that is modular and free of cross-cutting concerns is clearly superior to the alternative. Untangling behaviours in a non-modular approach runs counter to the goal of reducing development time. Ideally, an AI can be made modular at the behaviour level, raising the level of abstraction at which a designer can work.

Behaviour trees [5] are one popular AI formalism. Here each branch performs a specific task, suggesting a reuse model based on pruning and reusing branches. Unfortunately, individual branches do not possess a clear task-based abstraction. If a task being performed by a branch must be interrupted, the branch itself must be aware of interrupting game events and how to return to the appropriate branch. Interrupting events therefore act as a cross-cutting concern, complicating the separation and reuse of branches.

A more appealing alternative is offered by Kienzle et al. [4], who introduce an AI based on an abstract layering of Statecharts. Here each Statechart acts as a modular component by implementing a single behavioural concern, such as sensing the game-state, memorizing data, making high-level decisions, and so on. Due to the clear demarcation of duties, the components are ideal for reuse.

Accordingly, this work explores reusing Statechart-based AIs, detailing the practical concerns. In section 2, we give a brief primer on Statecharts along with an overview of the layered model. Section 3 examines the interactions of both Statecharts and their associated classes under this model. Next we delve into reuse in section 4, introducing a reuse interface for AI-modules. This includes grouping behaviours for high-level reuse and porting AI into new games. Section 5 provides some concluding thoughts.

2. BACKGROUND

2.1 Statecharts

Statecharts were introduced by David Harel in 1987 [2] as a formalism for visually modelling the behaviour of reactive systems. With the introduction of UML 2.0 came the Rhapsody semantics [3], which are more in tune with mod-

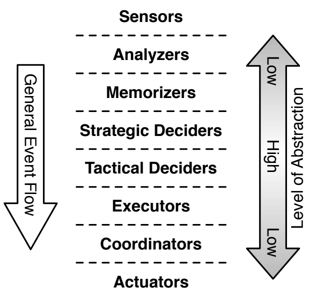


Figure 1: The Layered AI Model Architecture

elling software systems. Since the main purpose of the AI models is to define reactions to game events, the event-based formalism of Statecharts is a natural choice.

Statecharts are comprised of a set of states with transitions between them. Transition labels are in the form "$m[c]/a$", where m is the event name, $[c]$ is a guard condition, and a is an action. Upon receiving event e, if there is a transition from the current state on e with $[c]$ evaluating to true, then the transition is triggered, executing a and moving the Statechart to the target state. States can have *OnEntry* and *OnExit* blocks where more actions reside. The current state is considered a *modal* property of the Statechart. The Rhapsody semantics additionally allow for sub-states, inner and outer transitions, history states (when returning to a state with sub-states), and orthogonal components. Readers interested in the full semantics are referred to [3].

Statecharts exist within their own execution environment and cannot directly interact with a game at-large. Instead, Statecharts may have *associated classes* that are free to interact. Statechart actions access the associated class by calling methods or reading and writing parameters. Transition guards can also be comprised of calls to the associated class, so long as the result is boolean in nature. These can be referenced in the transition labels and both OnEntry and OnExit blocks, thus linking the graphical Statechart to the program at-large. The members of the associated class are the *non-modal* properties of the Statechart.

2.2 Layered Statechart-based AI

Developing complex AI benefits greatly from a separation of concerns. In [4], a complete Statechart-based AI is designed using the layer-based separation of behavioural tasks shown in Fig. 1. Independent behaviours are implemented in separate Statecharts that interact with layers above and below to form a complete AI. Communication takes place via event broadcasting and synchronous method calls.

At the lowest layer lie *sensors*. These read the game state, typically through listeners or observers that generate events when a change is detected. Events are passed up to *analyzers* that interpret and combine sensing data to form a coherent picture of the game state. The next layer contains *memorizers*, which store analyzed data and complex state information for later reference. The highest layer is the *strategic decider*[1], which interprets analyzed and memorized data to decide upon a high level goal. The high level goal is passed down to the *tactical deciders* to determine how it will be executed. Becoming less abstract, the next layer provides

[1]Typically, there is only one strategic decider, but an AI that needs to perform orthogonal tasks could have a strategic decider for each of them.

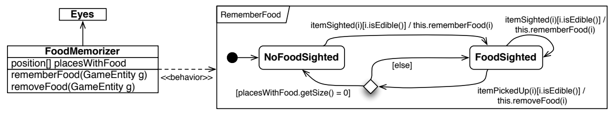

Eyes
FoodMemorizer
position[] placesWithFood
rememberFood(GameEntity g)
removeFood(GameEntity g)
<<behavior>>
RememberFood
itemSighted(i)[i.isEdible()] / this.rememberFood(i)
itemSighted(i)[i.isEdible()] / this.rememberFood(i)
NoFoodSighted
FoodSighted
[else]
[placesWithFood.getSize() = 0]
itemPickedUp(i)[i.isEdible()] / this.removeFood(i)

Figure 2: A Sample AI Statechart

executors that enact execution decisions, translating goals into actions. Depending on the current state of the actuators, certain commands can cause conflicts or sub-optimal courses of action. Conflicts of this type are resolved by the *coordinators*. The final layer contains *actuators*, which execute actions by modifying the game-state.

Together, the various Statecharts form an AI where low-level sensing data gradually transforms into high-level goals that filter down to become actions. Each Statechart is modular and placed in the appropriate layer. In Fig. 2, a sample FoodMemorizer is presented. It reacts to `itemSighted` events by memorizing the item if it is food. Other Statecharts access this information by calling the FoodMemorizer's associated class.

2.3 Generics

Similar to package-level parameters in UML, Statecharts can be parameterized to yield generic Statecharts. Typically, non-modal properties in the associated class would be set when the Statechart is initialized/instantiated.

A simple example comes in the form of a KeyItemMemorizer, where the key item is a type parameter set at runtime. A game implementing several items (such as flowers, shirts, and boxes), could have a KeyItemMemorizer for each item. When a KeyItemMemorizer receives an ItemSpotted event, the payload would be inspected and compared against the key item parameter. Only if it matches the parameter would the item be memorized. Other items would be ignored.

3. AI-MODULE INTERACTION

Reusing Statechart modules in a layered AI requires a solid understanding of how modules communicate. For this discussion, the term *AI-module* is defined as the pair consisting of a Statechart and its associated class. AI-modules permit three types of communication: message passing between Statecharts, calls between associated classes, and calls from associated classes to the game at-large.

3.1 Message Passing

In an abstract layering of Statecharts, generated events are broadcast asynchronously to all Statecharts. An event is received by a Statechart when it has a transition on that event from the current state. Typically, events are largely ignored except by one or two receiving Statecharts. Under this asynchronous model, the send is the end of the communication. The sending Statechart continues normal operation, and receiving Statecharts are free to proceed as they will.

In some implementations, events can carry payload. For instance, if an event `item_spotted` is generated by a sensor, the event could include a reference to the item for further processing at higher levels of abstraction. Analogous to a method signature, an *event signature* is the event name plus the payload type. Thus, the event `ItemSpotted` is different than the event `ItemSpotted(Item)`, despite sharing the event name. Resolving overloaded event names can be dealt with at the modelling level by using event renaming to eliminate name duplication, or at the implementation level through type-checking of payloads.

Synchronous communication is also employed in the layered model. AI-modules can offer methods in their associated classes for other AI-modules to call. By design, these methods are primarily getters, and do not change modal or non-modal properties of the offering AI-module. As an example, a memorizer stores obstacle positions; other Statecharts requiring that data obtain it by calling a method in the memorizer's associated class.[2]

Message passing is *unintended* when unrelated Statecharts use duplicate event names and begin to send and receive with each other. While not inherently problematic, there are many situations where a bug could result. For instance, if a sensor transmits all player sightings, and a combat module which accepts all player sightings is introduced, then the AI might act as though all players are enemies. The player sighted message is intended only for the PlayerAnalyzer to determine friend or enemy status and pass a new event accordingly. Naming issues are addressed in detail in §4.2.

3.2 Game Interaction

The range of calls and syntax employed is dependent on the execution environment employed. Most provide either direct access to the associated class, or a way to furnish the execution environment with a context where calls can be made. Either way, Statechart actions found in transitions, as well as entry and exit blocks for states, access the associate class, which in turn can access the game at-large.

In the layered model, most calls that travel from a Statechart to the game at-large occur in an actuator or a sensor. Calls from the actuators' associated classes attempt to change the game-state, and calls from the sensors' associated classes attempt to read the game-state.

4. AI-MODULE REUSE

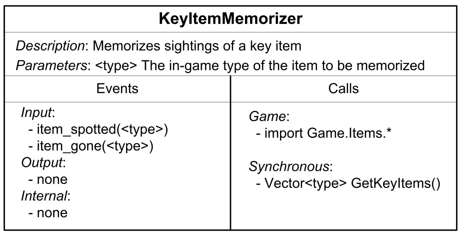
Designing a new AI through the reuse of existing AI-modules is a process of selecting behaviours described by existing modules, and then linking them together to form a working system. By clearly detailing Statechart interactions in an interface, the process reduces to satisfying the interface for the selected components. Additionally, a specified interface enables the development of tools to aid and simplify the process.

4.1 The AI-Module Interface

Like an API, the interface for an AI-module needs to denote the possible interactions. We build such an interface by addressing the interactions highlighted in §3.

Primarily, communication comes in the form of event-based message passing. By classifying events, the designer can communicate the intended interaction. *Input events* originate outside of the Statechart and act as notification of an event occurrence. *Output events* are generated by the current Statechart with the intent to trigger events in other Statecharts. It is possible for there to be events generated

[2]Synchronous communication *can* occur through events. A requester sends out an event, then transitions into a blocking state where the only transition out is a callback event. The receiving Statechart responds with the callback to complete the communication. However, this introduces the potential for deadlock if Statecharts are mismatched, and thus is not considered good practice in the context of reuse



KeyItemMemorizer	
Description: Memorizes sightings of a key item *Parameters*: <type> The in-game type of the item to be memorized	
Events	Calls
Input: - item_spotted(<type>) - item_gone(<type>) *Output*: - none *Internal*: - none	*Game*: - import Game.Items.* *Synchronous*: - Vector<type> GetKeyItems()

Figure 3: A sample AI-module interface for a hypothetical KeyItemMemorizer

by a Statechart with a matching transition in the same Statechart. If no other Statecharts are meant to send or receive these events, they should be classified as *internal.* While event classification is done manually by the designer, a tool could suggest likely classifications to streamline the process.

Next, the interface must detail linkages formed through the associated class. Synchronous communications must be presented in terms of outgoing calls and synchronous calls made available. For reasons of portability (as detailed in §4.4), a similar treatment needs to be given to all game linkages in the associated class. This includes imports along with calls to the game-state (including the methods these calls reside within). Finally, any generic parameters should be noted along with a description.

The actual interface is thus a listing of event and call information, along with a description of the Statechart including parameters. An example interface for a KeyItemMemorizer is given in Fig. 3. While the arrangement of information is arbitrary, the information shown fully describes the interactions of the AI-module.

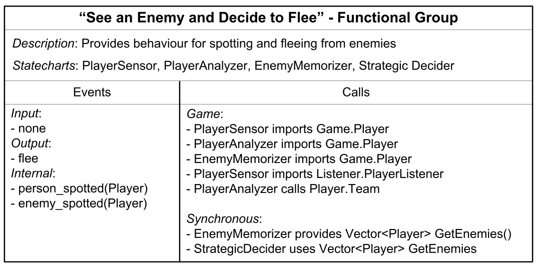
4.2 Renaming

Statechart communication requires *event signature coherence*, where the output of one Statechart matches the expected input of another. In a non-reuse scenarios, this is ensured by construction. In a reuse scenario, event name coherence cannot be assumed; we must actively create it by renaming events. This is done to enable message passing, or to prevent unintended message passing. Essentially, renaming connects AI-modules together.

Consider an NPC with a PlayerSensor that senses player movement and outputs a `player_spotted` event. We wish to reuse this Statechart by adding it into a new NPC. However, the higher level PlayerMemorizer has `player_seen` as an input event. The simplest solution is to rename one of the events (e.g., rename `player_spotted` to `player_seen`) so that a send-receive pairing is established.

The other scenario is the prevention of unintended message passing. Renaming in this case correctly severs unintended communications between unrelated Statecharts. This is usually necessary to protect internal events. By definition an internal event should never be sent or received by another Statechart. In case of conflict, the internal event can be renamed as this poses little risk of affecting existing message passing relationships.

All renaming must take into account existing event names, since there could be additional relations using the old or new event name. Largely, this can be managed through tools that warn of potential conflicts. This is somewhat analogous to the use of global variable names in programming, because the broadcast model lacks a concept of scoping.



"See an Enemy and Decide to Flee" - Functional Group

Description: Provides behaviour for spotting and fleeing from enemies

Statecharts: PlayerSensor, PlayerAnalyzer, EnemyMemorizer, Strategic Decider

Events	Calls
Input: - none *Output*: - flee *Internal*: - person_spotted(Player) - enemy_spotted(Player)	*Game*: - PlayerSensor imports Game.Player - PlayerAnalyzer imports Game.Player - EnemyMemorizer imports Game.Player - PlayerSensor imports Listener.PlayerListener - PlayerAnalyzer calls Player.Team *Synchronous*: - EnemyMemorizer provides Vector<Player> GetEnemies() - StrategicDecider uses Vector<Player> GetEnemies

Figure 4: The interface for a functional group.

4.3 Behaviour Reuse

When building a new AI, the most intuitive approach is to consider what the AI will do. Ideas like "I want the NPC to flee from enemies", or "I want the AI to collect flowers" should be primary design goals. A designer wants their AI to carry out these goals, and is not as concerned with the events and Statecharts involved. To permit design at this level, we need to first isolate behaviours.

To track how high level goals traverse the hierarchy, we introduce *event chains*, defined as a series of events covering more than one Statechart that together effect a goal. An event chain "spotting an enemy" may have the following steps: at the PlayerSensor, seeing another player in-game and generating a `person_spotted` event; at the PlayerAnalyzer, receiving the `person_spotted` event, analyzing the player to determine their threat, and generating an `enemy_spotted` event; at the EnemyMemorizer, storing a reference to the enemy; and at the StrategicDecider, receiving the `enemy_spotted` event and determining a response. The event chain is thus {`person_spotted`, `enemy_spotted`}.

Event chains trivially support nesting. Since an event chain is a series of events, nesting an event chain is shorthand for including all events from the nested chain in the outer chain. For instance, the event chain for "see an enemy and decide to flee" could be constructed as {"spotting an enemy", `flee`}, where `flee` is the high level goal chosen by the StrategicDecider.

Ultimately, behaviours are expressed by the series of events in an event chain. Accordingly, we define a *functional group* as the set of Statecharts that together contain all senders and receivers used in an event chain. The most interesting result is that a functional group can be given a composite interface. All inputs paired with outputs and all synchronous events that are both offered and called are reclassified as internal. Parameters and game calls from all constituent interfaces are combined. The resulting group interface subsumes all member interfaces. The result is a tremendous simplification to design at the behavioural level.

Looking at the "see an enemy and decide to flee" event chain, the Statecharts involved are the PlayerSensor, the PlayerAnalyzer, the EnemyMemorizer, and the Strategic Decider. Note that the `flee` event does not have a receiver. This is the output event for this event chain, and anything building onto this functional group would need to have a receiver for the `flee` event. In Fig. 4, the combined interface for this functional group is shown.

4.4 Portability

Portability depends on the Statechart implementation that is employed. To easily port an AI from one game to another, there must be a Statechart execution environment in the target language that can use the existing Statecharts. One candidate for Statechart representation is SCXML [1], which defines a standardized representation of Statecharts in a human-readable XML format. Execution environments for SCXML are available for Java, C++, Python, and several other languages. Another workable standard is the XMI format [6], commonly employed by UML tools.

Porting the associated class is more complicated and may require re-coding, especially in situations where the target game is in a different language than the source game. The process is aided by the interface, since it details game calls and synchronous methods. Rewriting associated classes could also be simplified through the introduction of a wrapper class to manage game calls.

5. CONCLUSIONS AND FUTURE WORK

When developing an AI, working at a high level of abstraction is valuable. Our notion of functional groups allows a customizable encapsulation of behaviour with a simple interface. Over time, a developer can build a library of behaviours, and develop new AIs largely by adding and piecing together existing components. New development would be limited to the subset of novel behaviours needed.

The default broadcast model is responsible for most of the renaming issues. As an alternative, narrow-casting would allow Statecharts to communicate directly, eliminating unintended message passing but introducing event targets. While this creates extra work for designers, a tool could support this by leveraging event-chains.

The next step in this work is the development of a tool that realizes these notions. AI-module interfaces would need to represented in a standardized format such as XML, or could even be inlined with the actual Statechart representation (e.g., SCXML comments). Importantly, good tool support would manage both message sending and the use of functional groups. Such a tool could then be thoroughly tested to quantify any improvement in development time resulting from these reuse techniques.

6. REFERENCES

[1] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, T. Raman, K. Reifenrath, and N. Rosenthal. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, May 2010.

[2] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Programming*, 8:231–274, 1987.

[3] D. Harel and H. Kugler. The Rhapsody semantics of Statecharts (or, on the executable core of the UML). *LNCS*, 3147:325 – 354, 2004.

[4] J. Kienzle, A. Denault, and H. Vangheluwe. Model-based design of computer-controlled game character behavior. In *MODELS*, volume 4735 of *LNCS*, pages 650–665. Springer, 2007.

[5] C.-U. Lim, R. Baumgarten, and S. Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of Evolutionary Computation*, volume 6024 of *LNCS*, pages 100–110. Springer, 2010.

[6] Object Modeling Group. XML metadata interchange (XMI), 2003. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI`.