

Reusable Components for Artificial Intelligence in Computer Games

McGill School of Computer Science, Technical Report #SOCS-TR-2011.3

Christopher Dragert, Jörg Kienzle, and Clark Verbrugge

McGill University,
Montreal, QC, H3A 2A7, Canada
Christopher.Dragert@mail.mcgill.ca
{Joerg.Kienzle, Clark.Verbrugge}@mcgill.ca

Abstract. While component reuse is a common concept in software engineering, it does not yet have a strong foothold in Computer Game development, in particular the development of computer-controlled game characters. In this work, we take a modular Statechart-based game AI modelling approach and develop a reuse strategy to enable fast development of new AIs. This is aided through the creation of a standardized interface for Statechart modules in a layered architecture. Reuse is enabled at a high-level through functional groups that encapsulate behaviour.

These concepts are solidified with the development of the SkyAI tool. SkyAI enables a developer to build and work with a library of modular components to develop new AIs by composing modules, and then output the resulting product to an existing game. Efficacy is demonstrated by reusing AI components from a tank to quickly make a much different AI for a simple animal.

1 Introduction

Complex and ubiquitous artificial intelligence (AI) has become a staple of modern computer games; players expect non-player characters (NPCs) to exhibit some amount of intelligence, react to player actions, and exhibit appropriate behaviour depending on their role within the game simulation context. Developing a good AI for a real-time game environment, however, is a difficult task, and while a variety of formalisms may be used, practical AI designs typically resort to strongly customized approaches that are closely connected with the underlying game architecture and NPC type. This results in a relative lack of reuse in game AI, increasing development costs and requiring repetitive development of often quite similar AI behaviours.

Here we describe an AI development strategy that promotes reuse. Our approach builds on a basic Statechart design [10], taking advantage of its inherent modularity and nesting properties. By providing a formalism and structure that encapsulates behaviours in terms of *functional groups* we are able to define

a development strategy that allows for extensive reuse of different AI components, including both high and low-level elements. Clear identification of code dependencies further permits analysis and the tool-based presentation required to ensure proper integration into the actual game code. This design results in a relatively quick development process that can take advantage of a library of AI behaviours to construct complex AIs, while at the same time simplifying the adaptation of the AI to an actual game context.

We illustrate our approach by constructing a new AI for a squirrel by reusing large portions of an AI designed for a tank. These are quite different game AI contexts, and would typically be approached as unique and very separate development tasks: the tank is a combat element intended for a competitive game, while the squirrel is a background NPC within the massively multiplayer “Mammoth” game world [11]. At a high-level, however, there are many behavioural similarities, and by expressing and reusing AI behaviours at a suitable level our approach is able to capture many of these commonalities: over half of the *AI modules* in our final squirrel AI are reused from the tank.

Our development and reuse approach is summarized and reified in an actual tool, *SkyAI*. This software framework directs and facilitates the development workflow, taking in visual Statechart components, providing an interface for producing novel *AI Modules* from a library of Statechart behaviours, and exporting code that can be directly incorporated into the game itself. By formally representing and understanding game code associated with specific behaviours, SkyAI is able to perform basic analysis of the constructed AI, ensuring code and functional group dependencies are properly satisfied and thus the AI module is well-constructed. SkyAI represents a useful illustration of the practical value and general feasibility of our design.

Major contributions of our work include:

- Describing a practical reuse strategy for developing game AIs based on Statecharts. This significantly extends previous work demonstrating the feasibility of Statecharts for game AIs [10], by defining efficient development, component definition, and porting strategies.
- A visceral demonstration of reuse by designing a basic AI for a squirrel, making extensive reuse of components originally defined for a tank.
- Concretizing the approach through the design of a software tool that facilitates our strategy. *SkyAI* is part of a practical workflow for AI generation, simplifying component reuse and including non-trivial analysis that aids in ensuring AI components are properly composed and integrated.

2 Background and Related Work

Artificial intelligence in modern computer games focuses upon controlling NPCs such that they exhibit behaviours relevant to the character’s role in the game context. This type of AI is referred to as *computational behaviour* or *computational intelligence*, distinguishing it from classical AI approaches. In a game context, efficiency, rapid development, and testability are paramount, strongly

constraining design approaches. Rather than more complex AI designs, game AI thus focuses on choosing behaviours through either arbitrary code expressed through a custom scripting context [17, 16], or relatively simple tree or graph structures, such as decision trees; Millington’s introductory text summarizes the variety of approaches [15]. Abstract models such as finite state machines are a commonly used formalism, wherein states represent behaviours and transitions are triggered to change the behaviour exhibited [6, 7]. Hierarchical Finite State Machines (HFSMs), a more structured approach frequently used in industry, incorporate aspects of Statecharts by allowing states to contain substates with internal transitions. However, the strict hierarchical nature can be limiting and places restrictions on how transitions between states are modelled. This limitation is shared by behaviour trees [9], which although they more clearly delineate how the system chooses behaviours, are also strongly hierarchical, and further suffer from a lack of modal states that encapsulate different behaviour groupings.

In the context of reuse, superstates in HFSM are treated as modules and exported to new AIs as such [13]. This approach is valuable, but omits important details, such as code portability, and provides for no interactions with internal states, limiting reuse to taking superstates as a whole. Behaviour trees offer a reuse model based on pruning and reusing branches [14]. In a practical sense, however, the extent of reuse is limited as individual tree nodes are often highly game or AI-specific—code actions and abstract, high-level behaviour are intimately entwined in behaviour tree models. Reuse has thus been demonstrated primarily in terms of modifying an existing AI [5] with incrementally improvements rather than porting a design to a fundamentally new context.

Alternative computational intelligence models are found in the robotics community where the notion of developing AI at the behaviour level is well established. The *subsumption architecture* proposed by Brooks [2] is highly influential, wherein each layer, made from a finite state machine, encapsulates a behaviour with lower layers being subsumed by higher layers. Later work by Bryson [3] presents *behaviour-oriented design*, a methodology to develop behaviour of an agent through hierarchical reactive plans, though code reuse is not explicitly considered. Our work is distinguished by the use of Statecharts [8], allowing us to flatten the AI structure, and design at each level of abstraction using the same formalism. Through associated classes, Statecharts have the additional advantage of elegantly including code, further improving modularity and reusability. In Fig. 1, a sample `FoodMemorizer` Statechart is presented. It reacts to *item-Sighted* events by memorizing the item if it is food. Other Statecharts access this information by calling the `FoodMemorizer`’s associated class. Statecharts have become of interest relatively recently to the game development industry, with initial designs focusing on low-level issues such as efficient interpretation within a game context [12].

2.1 Layered Statechart-based AI

Our work adopts the formalism developed by Kienzle et al. [10], who introduce an AI based on an abstract layering of Statecharts. This approach is inspired by



Fig. 1. A Sample AI Statechart

the *sense-plan-act architecture* common in robotics. 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. Importantly, there are many Statecharts in this system, meaning the exhibited behaviour is based upon the superposition of states.

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

3 The AI Module

The layered Statechart-based AI approach clearly suggests the Statechart as the reuse component. While HFSMs also use Statecharts to an extent in their modelling formalism, the associated class is neglected. As an elegant way of including and reusing code, the associated class is essential in a reuse scenario. As such, we use the term *AI Module* to describe our fundamental reuse component and define it as a Statechart and its associated class. The advantage of this approach is that we can now reason at a high level about both AI behaviour and implementing code. Fig 2 presents a UML diagram for a sample AI module from the Tank Wars AI in [10].

Reuse of an AI Module is dependent largely on how it interacts with other modules, along with its own internal settings. In this section, these factors will be examined in detail, culminating in the presentation of an interface for AI module reuse. This section is based upon our work in [4] that first suggested the AI module as the fundamental component for AI reuse.

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.

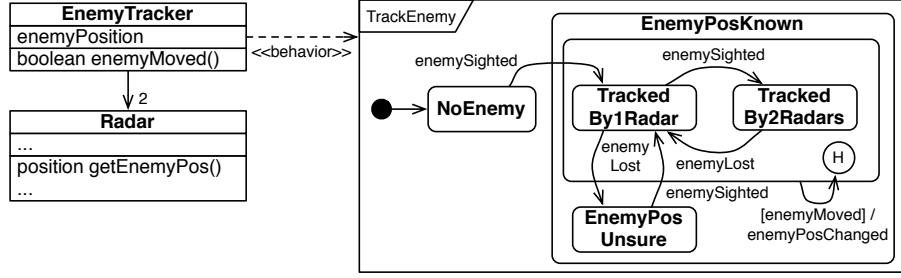


Fig. 2. The **EnemyTracker** AI module that tracks the position of an enemy

3.1 AI Module Interaction

Modules can communicate in two ways: either by *event-based message passing*, or by making *synchronous calls* between associated classes. Event-based message passing is the primary means of communication. Depending on the implementation, events can be either broadcast or narrowcast. Broadcasting events can result in naming conflicts, while narrowcasting requires additional work at the modelling level to correctly target communications. We adopt a broadcasting model since it simplifies interfaces and makes event renaming straightforward. Additionally, event broadcasting is asynchronous in nature and is thus more loosely coupled than a narrowcasted system, simplifying reuse.

Synchronous calls are fundamentally different than events, in that they tightly couple components. A synchronous call is made from the associated class of one module by calling a method in another module's associated class. A typical usage scenario would be requesting complex information that is stored or computed by another module, such as a memorizer storing information about the game-state and making it available through a synchronous call.

Depending on the implementation, events may carry a payload. Instead of just having a name, events gain a *signature* stating the type of the payload, much like a function or method would. Using payloads reduces the need for synchronous calls, since events can include pertinent information and thus satisfy upstream data requirements. This allows for a more loosely-coupled system overall, better suited for reuse.

Some associated classes interact not only with the AI, but access the game at-large. A necessary precondition to doing so is including or importing game classes. If an AI is to be reused in a different game, these calls would be points of failure and thus tracking them is important with regards to reuse.

Finally, the associated class can have parameters which are set at run time. By addressing these parameters at the AI module level, these can be modelled and set to enable further reuse. A example comes in the form of a **KeyItemMemorizer** module, a generalization of the **FoodMemorizer** shown in Fig. 1, where the item to be memorized 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 relevant to the AI. When a **KeyItemMemorizer** receives an

EnemyTracker	
<i>Description:</i> Tracks an object's position using two observation points	
<i>Game:</i> Tank Wars	
<i>Parameters:</i> <type T>::The in-game type of the object to be tracked	
<i>Language:</i> C++	
Events	Calls
<i>Input:</i> -enemySighted(<T>) -enemyLost(<T>)	Game Imports: -import TankWars.<T>
<i>Output:</i> -enemyPositionChanged(<T>)	Synchronous Calls: -Radar.getEnemyPos()
<i>Internal:</i> none	Available Calls: -boolean enemyMoved()

Fig. 3. The interface for the **EnemyTracker** AI module.

`item_spotted` event from a sensor, the payload would be inspected and compared against the key item parameter, and then memorized if it is a matching type.

3.2 AI Module Interface

The interface for an AI module collect the above mention properties relevant to reuse. This makes it possible to compose AI modules solely by working with module interfaces, and to perform analysis based upon interface interface.

Before we can construct an interface, the nature of event-based interactions must be examined. Events that are generated and broadcast are defined as *output* events. From the point of view of a Statechart that receives an event causing a transition, that event is an *input* event. It is possible for a Statechart to communicate with a self-contained orthogonal region by sending a transition. Such an event would be classified as *internal* if no other Statecharts should interfere with that communication by either sending or receiving that event. During actual reuse, event classification would be done by the designer of the module.

Along with identifying all outgoing synchronous calls, the interface should give the signature of any functions or methods made available by the associated class. Similarly, all game imports should be listed, showing how the module relates to the game. If an AI module has no game imports, then it is *game-agnostic* and can be safely reused in a different game without issue. Additionally, any parameters and their types must be listed, and given values when the module is reused.

Pertinent information can optionally be added to the interface, such as a name for the module, an outline of the behaviour, and the intended layer. While technically this is unnecessary, it improves usability for any designer attempting to reuse the AI module. In Fig. 3, we see an interface for the EnemyTracker module presented in a graphical format. In our SkyAI tool, the XML format is used to represent and store Statecharts.

3.3 Functional groups

An existing AI will have many modules, each sending and receiving events, that together contribute to generate the *behaviour* of the NPC being controlled. The

key notion here is behaviour—merely having communication between modules is somewhat irrelevant. It is behaviour of the AI due to the combined action of the modules that constitutes output. Thus, the effectiveness of a reuse strategy should be considered foremost in terms of behaviour granted. Of course, the primary benefit of reuse is the reduction in development effort. Behaviours that can be reused do not need to be developed, allowing development to focus on getting the behaviour correct. This implies that the correct level of abstraction for AI module reuse is actually at the behaviour level, not the module level, agreeing with the trends in robotics and computational intelligence.

We capture this notion by introducing *functional groups*, which are compositions of AI modules that encapsulates a portion of AI. Each member of a functional group communicates with at least one other member, either through event-based message passing or with a synchronous call. In the Tank Wars AI [10], there are several modules that work together to form the fuel management system for the tank. These are the **FuelTank** sensor, the **FuelStation** map, and the **RefuelPlanner**. These can be combined into a single **FuelManagement** functional group, and reused in a new AI as a single group.

The most interesting aspect of this approach is that a functional group can be given a composite interface identical in form to the interface for an individual module. Input events that pair with output events can be reclassified as internal if there is a need to protect internal message flow, while parameters and game calls from all constituent interfaces can be combined. Unpaired events are added to the interface without change, and become the connection points for other modules. The resulting group interface subsumes all member interfaces, and can be used as the sole interface for the group. Overall, functional groups are a key simplification that raises the level of abstraction at which a designer can work.

4 AI Module Reuse

Creating a new AI through reuse consists of planning the desired behaviour, then importing modules and functional groups to implement those behavioural goals. Undoubtedly there will be behavioural gaps that require the development of new modules. However, it is anticipated that over time, a library of modules will be built up, thereby reducing the amount of new development required.

As each module is added to a new AI, some integration must be performed to connect the new component with modules already present. Since functional groups use the same interface as AI modules, they do not present a special case and the following integration strategies applies equally to both functional groups and modules. Also, the layered approach relies on modal properties of Statecharts to determine behaviour, and uses an asynchronous broadcast model for most communication. In our experience this significantly limits the complexity of inter-module communication, obviating the need for module protocol management through constructions such as protocol state machines.

4.1 Component Integration

New modules are added to an AI by *connecting* them to modules already present. A connection could be in the form of event-based message passing or a synchronous call. If the connection is to be event-based, then the involved AI modules need to pair an input and output event. In the typical case, event renaming is sufficient to create an such a pair between the AI modules. There are edge cases where this approach becomes awkward, such as when an event is received by multiple modules but the intention is for the new module to connect to only only one of them. Here, the simplest solutions are modification of the receiving Statechart or usage of narrowcasting.

As with any event renaming scheme, existing names must be respected. New modules should rename their events when an event name is already used by existing modules and no connection is desired. Certainly, internal events need to be protected by renaming events in the new module such that there is no overlap with events classified as internal by any existing module. Similarly, internal events in the new module could be renamed in case that event name is already used by existing modules.

Integration relating to the associated class is less forgiving, as all game imports and all synchronous method calls must be satisfied to prevent compilation and run-time errors. In the case of unsatisfied method calls, either the target AI module must also be included, or the associated class must be modified to point to a new implementing module. Regarding game imports, reusing a module for an AI in the same game is always safe. However, moving a module to a new game will require updating all game imports to similar calls in the new game. Many AI modules will be purely behaviour driven and will have no game imports. These modules are *game-agnostic*, and may be freely moved between games.

In a reuse context, synchronous calls are more restrictive than event-passing, and limiting the number of synchronous calls simplifies integration. Event passing occurs at the modelling level and can easily be renamed and redirected; unreceived output events will at worse cause a behavioural issue. In general, the following guideline may be used: when a module receives an event and needs to take action immediately based on information known at the sending module, that event should include the relevant information as event payload. When a module needs complex state information, or if it does not need to take immediate action to an event, then a synchronous call is appropriate.

One outstanding issue is that of event prerequisites and preconditions. A module may require a specific input before producing an output event. If that input never arrives, then the module will never output. These issues may be analyzable through verification and model-checking of the Statecharts, providing additional utility to this reuse approach.

5 Case Study: Tank to Squirrel

To demonstrate the validity and usefulness of the presented approach, this section gives a concrete example of AI reuse. Here, we take the AI developed for

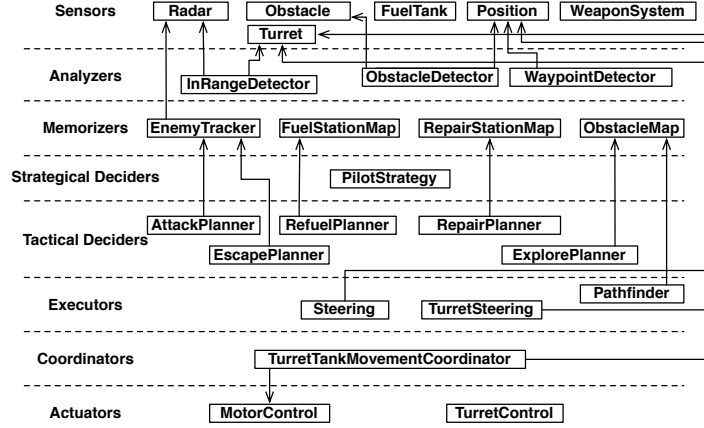


Fig. 4. Detailed Tank Architecture: Lines represent synchronous calls between modules.

the Tank Wars game in [10], and reuse components of it to create the AI for a squirrel. Squirrels are small land-based rodents that collect nuts and acorns, and would seem to have little in common with a military vehicle. Using the described reuse techniques coupled with good modular design practices, we find that many elements of the tank can indeed be reused, greatly simplifying the development time of the new squirrel AI.

The tank AI was originally developed for the Tank Wars simulation by Electronic Arts. A brief overview is presented here; readers interested in full details are referred to the original paper [10]. The AI consists of 24 modules in total, divided across the layers as shown in 4. Each of the connecting arrows represents a synchronous call; event based message passing is not shown on the diagram. Behaviourally, the tank explores the game world looking for enemies, gas stations, and repair stations. As it explores, it notes the position of any obstacles for future use in pathfinding operations. When the **Radar** detects an enemy, the tank engages as defined in the **AttackPlanner**. If the tank is damaged, the **EscapePlanner** can choose to retreat to the repair station. The game actions performed by the tank are limited to moving forward and back, turning left and right, and rotating and firing the turret.

In this work, we are using a new version of the Tank AI written for Mammoth [11]. Mammoth is written in Java, and uses SCXML Commons as the Statechart execution environment [1]. This version of the tank was written such that it generally respects the UML descriptions given in the Tank Wars paper. As Mammoth supports event payloads, we were able to make the AI more loosely coupled by eliminating some synchronous calls. For example, when the **Radar** module spots a player it creates an *enemy_sighted* event, which is always followed by the **EnemyTracker** making a synchronous call to the radar to gather information about the enemy. Using event payloads, this is simplified by having the **Radar** create a *player_spotted* event with enemy information as payload.

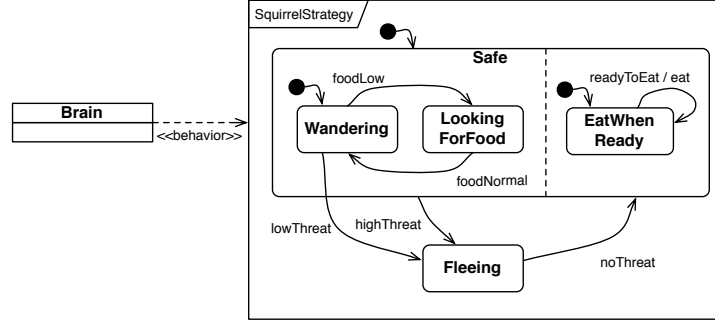


Fig. 5. SquirrelBrain Module.

Tank to Squirrel Reuse Summary		
Tank Module	Reuse Modifications	Squirrel Module
EscapePlanner	Synchronous call to <i>EnemyTracker</i> pointed at <i>NearbyThreats</i>	Flee
RefuelPlanner	<i>look_for_fuel</i> input renamed to <i>look_for_food</i> <i>move</i> output renamed to <i>take_item(Item)</i> <i>FuelStation</i> import replaced with <i>Item</i> import	LookForFood
ExplorePlanner	<i>explore</i> input event renamed to <i>wander</i>	Wander
WaypointPathfinding (functional group)	All intra-group communications reclassified as internal events.	WaypointPathfinding (functional group)
FuelStationMap	<i>FuelStation</i> import replaced with <i>Item</i> import Parameter <i>KeyItemType</i> set to type <i>Item</i>	SeenFood
FuelTank	Parameter <i>fuel</i> set to <i>energyLevel</i>	Energy
Position	<i>Tank</i> import replaced with <i>Squirrel</i> import	Position

Table 1. A summary of the module modifications in reusing tank modules for the squirrel AI.

The target AI seeks to model a squirrel, which in the Mammoth game world is intended to be a minor background character, moving from tree to tree and collecting food when it can. Characters in Mammoth are restricted by energy, which is consumed as the AI acts. For squirrels, it is restored by eating food that they’ve gathered. Naturally fearful, squirrels seek to maintain a healthy distance from any non-squirrel that approaches. Basic squirrel behaviour is thus the ability to find food and eat when energy becomes low, while keeping a healthy distance from threats. We will seek to reuse modules from the tank whenever possible, without compromising on the intended behaviour.

5.1 Applying the Reuse Process

At the highest level, tank behaviour is much different than that of a squirrel, and it is unrealistic to try to reuse the tank’s strategic decider. A new strategic decider called the **SquirrelBrain**, shown in fig. 5, was developed for the squirrel. It can be thought of as the root of the new AI in that building outwards from the required behaviours of the **SquirrelBrain** allows us to select the AI modules to reuse. The SquirrelBrain uses four high level goals: wander, look for food, flee, and eat.

Reuse was employed as much as possible. As modules were inserted, changes were made to connect modules and link them to new modules. A summary of modules reused and changes made is given in Table 1. The remainder of this section gives the rationale for each change, and gives insight into the thought process behind reuse.

Three of the four high level goals were addressed by reusing existing tactical deciders. Fleeing is handled by reusing the **EscapePlanner**. The **RefuelPlanner** performs the look for food goal, insofar as it moves towards a previously spotted fuel depot or searches if it hasn't seen one, the exact functionality required when a squirrel looks for food. Additionally, wandering is akin to the **ExplorePlanner**, and can be reused by simply renaming input event *explore* to *wander*. This connects the three planners to the **SquirrelBrain**.

The process continues by building outward and attaching modules to input and output events that are unconnected. When the tactical deciders choose how to perform a goal, they send output events to executors and coordinators, where they are transformed into concrete actions. Tank executors focus on steering and turret control, neither of which apply to a squirrel. This means that no reuse is possible at this level, and so new executors must be developed. This includes a **MoveAway** executor, designed to receive the *flee(Position)* output event from the **FleePlanner** and pick a concrete flee destination. While the tank could just refuel, the squirrel must pick up and collect food and thus needs another executor. The new **TakeItem** executor ensures that the squirrel is close enough to a target object to pick the item up, and issues move and eat commands accordingly.

Pathfinding is addressed with a functional group. The **Pathfinder**, **ObstacleMap**, **WaypointDetector**, and **Obstacle** sensor together perform waypoint-based pathfinding. These are composed then reused in the squirrel AI.

At the level of concrete actuators, the squirrel is much different than the tank. It has no turret and nothing to coordinate, leaving the coordinator layer empty. The actuators for the tank are relative, while the squirrel can simply move to a location. Thus, the squirrel needs a new **Move** actuator, along with a **Pickup** and **Move** actuator. These receive events with payloads that determine the actuation target. **Move** receives the output *move.destination(Position)* events sent by the **WanderPlanner**, **MoveAway** executor or **Pathfinder**, **Pickup** connects to the **TakeItem** executor, and **Eat** receives *eat* events sent by the brain, directly fulfilling the simple high level eat goal.

The required memorizers are already spelled out by the unsatisfied synchronous calls from the planning modules. The **EnemyTracker** needed by the **Flee** planner is unsuitable since it memorizes the location of only one enemy and multiple threats exist for a squirrel, so a new **NearbyThreats** memorizer is connected and used. However, the **FuelStationMap** is still appropriate, since it memorizes locations. That AI module is actually a **KeyItemMemorizer** (as seen in Fig. 1) and is reused by simple parameter modification.

Last come the sensors and analyzers. The only analysis for the squirrel is determining threats (which the tank did not do) so a new module, the **Threat**

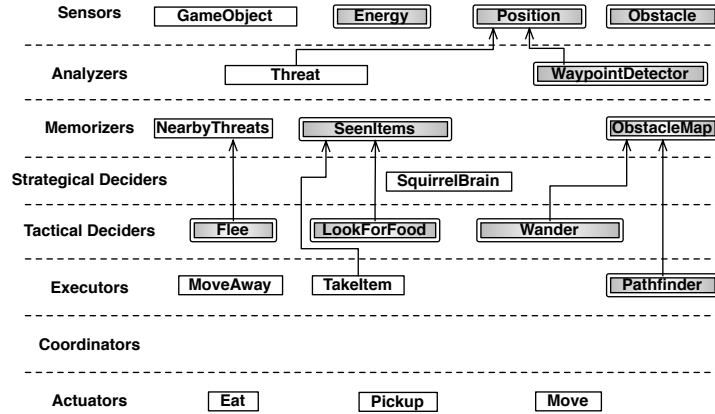


Fig. 6. Detailed Squirrel Architecture: Shaded modules are reused from the Tank.

analyzer, is needed. This connects to the `NearbyThreats` memorizer through event-passing. The `FuelTank` sensor and the `Position` sensor are both reused to complete the sensing input. Lastly, the `GameObject` sensor is new, since the tank did not explicitly detect in-game objects.

5.2 The New Squirrel

The resulting squirrel AI is shown in Fig 6. It contains 19 modules, 10 of which were reused from the tank. By looking at the AI module interfaces, connecting AIs was straightforward; all information relevant to reuse was included in the module. This connection process provides some confidence in the design—we know the new `MoveAway` executor is connected to the reused `Flee` planner since we explicitly connected them through event renaming. We know the `SeenItems` memorizer gets item information since it was connected to the `GameObject` sensor using an event with an `Item` payload, and so on.

This example shows that reusing even an unrelated AI can still yield a large improvement in development time, with more than half of the AI being reused. Similar AIs could be developed even faster. For instance, a Bear AI that scrounges for food but attacks players instead of fleeing could reuse almost the entire squirrel AI, plus a few new modules that handle attacking.

6 The SkyAI Tool

With a formal AI module interface, we enable the creation of a tool for AI module based reuse. Our first iteration of such an application is called SkyAI. It allows a user to grow and manage a library of AI modules, and create new AIs through reuse by adding modules and changing their properties. While SkyAI is still in a pre-alpha state, it already reinforces the validity of our reuse approach.

SkyAI uses an abstract representation of the AI module, building each module from their source files with guidance from the designer. Right now, only

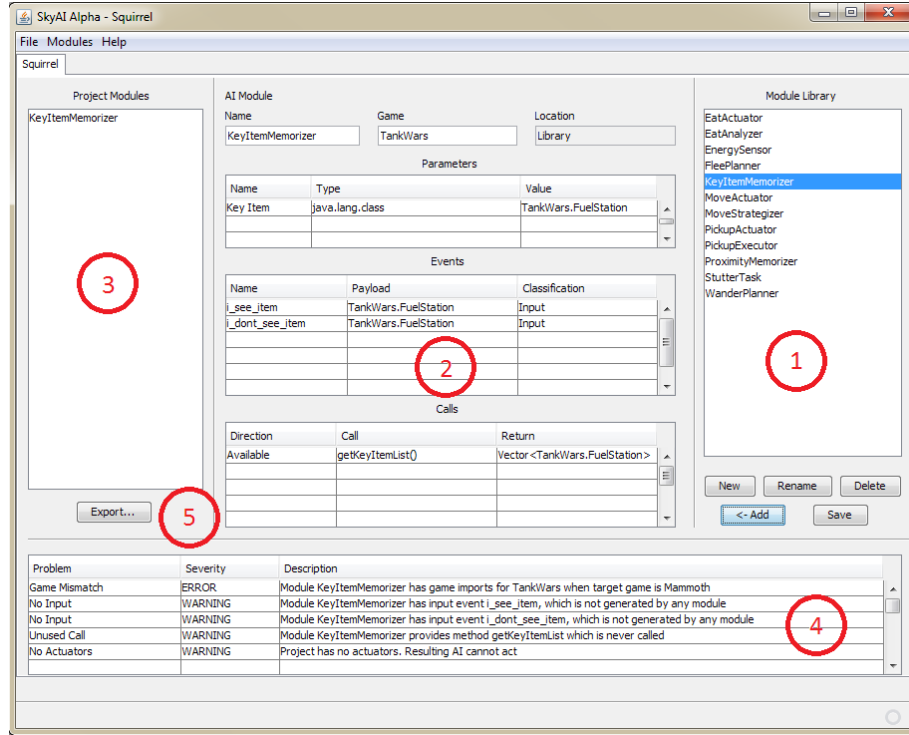


Fig. 7. SkyAI's main GUI

Statecharts represented in SCXML and associated classes written in Java can be processed, but the architecture supports later expansion to different representations and languages. AI modules are stored in an XML format, and managed by SkyAI along with the source files.

6.1 SkyAI Workflow

In Fig.7, we see the main interface for SkyAI with important elements numbered to aid in explanation. Usage of SkyAI begins by creating new modules. The user specifies the source SCXML Statechart and Java associated class, then SkyAI reads the source file to extract information about events, methods, and imports, finally adding the module to the library. In region 1 of the GUI, we see the the Module library, which lists modules that have been imported into SkyAI and are available for reuse. When a module is selected, region 2 lists the AI module properties and supports modification. Some module properties must be adjusted manually as they are design decisions (such as making an event internal).

Once the library is populated, a new project may be built. A project is how SkyAI represents an AI in progress, and targets a specific game. Modules are selected from the library listing and added to the project, showing up in region 3. Selecting a module here will allow the designer to make specific reuse

changes such as those described in Table 1. While the designer works, region 4 displays errors and warnings regarding the project, giving some guidance to design process. Finally, when the AI is constructed, the export button marked as region 5 allows the project to be exported. Any changes to modules are written back into the source files and exported directly to the target game.

6.2 Errors and Warnings

Perhaps the most important support feature in SkyAI is the error and warning system. A number of potential issues arise when building a new AI through module reuse, primarily related to connecting modules. These are classified as *errors* if they will prevent the AI from running, and must be corrected before exporting is allowed. A problem is merely a *warning* if it is a potential source of behavioural error, but will not prevent the AI from running. These are also listed in the project interface, and may be ignored. The current set of errors and warnings is restricted solely to errors at the interface level, and is shown in Table 2.

Severity	Problem	Description
Error	Game Mismatch	Module x has game imports for g when target game is j .
Error	Unsatisfied Call	Module x calls m in <i>class</i> , which does not exist.
Error	Event Interference	Event e is internal to module x , but is used by module y .
Warning	No Input	Module x has input event e , which is not generated by any module.
Warning	No Receiver	Module x outputs event e , which is not received by any module.
Warning	No Actuators	Project has no actuators. Resulting AI cannot act.
Warning	Unused call	Module x provides method m which is never called.
Warning	Null Parameter	Parameter p in module x is null.

Table 2. The list of warnings and errors generated by SkyAI.

7 Conclusions and Future Work

There is strong commonality within game AIs, even between apparently different character classes—certainly at a high-level, NPCs have many similar behaviours. Historically, however, reuse has been complicated by a focus on context-dependent reactive behaviour, and the need to express the AI in terms of strong code dependencies. As we have shown here, a Statechart-based approach greatly helps in exposing the reuse opportunities, encapsulating the reuse at an appropriate level that encompasses not just a specific mechanic, but the high-level, behavioural abstraction. By composing functional groups of behaviours, novel AIs can then be directly constructed from a library of AI modules, a very practical strategy we demonstrate in the design of the SkyAI development tool.

A primary benefit of formalizing reuse as we have done is in further being able to validate and perhaps even procedurally generate new AIs. Our design facilitates model-checking and verification, and as part of future work we are developing analyses that help in identifying and avoiding some of the more intricate logical errors that may arise in combining larger and more complex AI modules. Implicit state or message dependencies, for example, are a potential concern that may be addressed through deeper analysis of functional group behaviours, as well as through more formal means of specifying Statechart interactions, such as found in protocol state-machines.

Acknowledgments. The authors would like to thank the Natural Sciences and Engineering Research Council of Canada for its support.

References

1. Apache Commons. Commons SCXML. <http://commons.apache.org/scxml/>, November 2010.
2. R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14 – 23, March 1986.
3. Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. In *Proceedings of the NODE 2002 agent-related conference on Agent technologies, infrastructures, tools, and applications for E-services*, NODE’02, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag.
4. Christopher Dragert, Jörg Kienzle, and Clark Verbrugge. Reusable components for artificial intelligence in computer games. In *to appear in Games and Software Engineering Workshop*, May 2011.
5. Max Dyckhoff. Evolving Halo’s behaviour tree AI. Presentation at GDC, 2007. <http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf>.
6. 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.
7. Sunbir Gill. Visual Finite State Machine AI Systems. Gamasutra: <http://www.gamasutra.com/features/20041118/gill-01.shtml>, November 2004.
8. David Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Programming*, 8:231–274, 1987.
9. Damian Isla. Handling complexity in the Halo 2 AI. *Game Developers Conference*, page 12, 2005.
10. Jörg Kienzle, Alexandre Denault, and Hans Vangheluwe. Model-based design of computer-controlled game character behavior. In *MODELS*, volume 4735 of *LNCS*, pages 650–665. Springer, 2007.
11. Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth: A Massively Multiplayer Game Research Framework. In *4th International Conference on the Foundations of Digital Games (ICFDG)*, pages 308 – 315, New York, NY, USA, April 2009. ACM.
12. Philipp Kollhoff. Level up for finite state machines: An interpreter for statecharts. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 317–332. Charles River Media, 2008.
13. John Krajewski. Creating all humans: A data-driven AI framework for open game worlds. http://www.gamasutra.com/view/feature/1862/creating_all_humans_a_datadriven_.php, 2 2009.
14. Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of Evolutionary Computation*, volume 6024 of *LNCS*, pages 100–110. Springer, 2010.
15. Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.
16. 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*, pages 33 – 38. Eurosis, August 2005.
17. Unreal Technology. The Unreal Engine 3. <http://www.unrealtechnology.com/html/technology/ue30.shtml>, 2007.