Reusable Components for Artificial Intelligence in Computer Games

Christopher Dragert, Jörg Kienzle, Clark Verbrugge
School of Computer Science, McGill University
Montreal, QC, Canada, H3A 0E9
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.

***Keywords*-AI, Computer Games, Reuse, Statecharts**

I. Introduction

Complex and ubiquitous artificial intelligence (AI) has become a staple of modern computer games; players expect non-player characters (NPCs) to intelligently react to player actions while exhibiting appropriate behaviours depending on their role within the game context. Developing a good AI for a real-time game environment, however, is a difficult task. While a variety of formalisms are employed, practical AI designs typically resort to strongly customized approaches closely connected to 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.

Previous work has argued the reason for this lies with the formalisms employed [1]. The applicability of software engineering practices becomes limited due to the use of non-modular custom approaches. As an alternative, a layered Statechart approach [2] provides an inherent modularity with nesting capabilities. 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 yields a faster development process with the ability to employ a library of AI behaviours to construct complex AIs, while at the same time simplifying 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. At a high-level, however, there are many behavioural similarities, and by expressing and reusing AI behaviours at a suitable level of abstraction 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 the tool *SkyAI*. This software framework directs and facilitates the development workflow, taking in Statecharts and associated classes, providing an interface for producing novel AIs from a library of behaviours, and exporting code that can be directly incorporated into a game. 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 is well-constructed. SkyAI represents a useful illustration of the practical value and general feasibility of our approach.

Major contributions of our work include:

- Extending the reuse strategy for developing game AIs based on Statecharts presented by Dragert et al. [1]. The work further demonstrates the feasibility of Statecharts for game AIs [2], 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, *SkyAI*. As part of a practical workflow for AI generation, it simplifyies component reuse and allows for non-trivial analysis helping to ensure that AI components are properly composed and integrated.

978-1-4673-1768-9/12/$31.00 © 2012 IEEE

II. Background and Related Work

Artificial intelligence in modern computer games focuses upon controlling NPCs such that they exhibit behaviours relevant to that character's role in the game. This type of AI is referred to as *computational behaviour*, distinguishing it from classical AI approaches. In the context of game development, efficiency and testability are paramount, strongly constraining design approaches. The easiest approach is to employ arbitrary code expressed through a custom scripting context [3], [4] or a relatively simple tree or graph structure, such as a decision tree. This type of narrow game-by-game focus is a source of consternation for game developers. At GDC 2011, Kevin Dill raised this exact issue arguing that the lack of behavioural modularity was stymying the development of high quality AI [5]. With no agreed-upon formalism for AI behaviour, there is no clear path towards the creation of open-source behaviour resources like there are for 3d models, animations, and so on. Development time is spent again and again crafting the same basic behaviours.

Industrial research into AI reuse focuses on modularizing code artifacts [6] or on modifying an existing AI system [7], [8] with incremental improvements rather than modularising and porting behaviours to a fundamentally new context. Our approach aims to shed light on this relatively unexplored space, by demonstrating how behaviours themselves can be modularized and reused in new game contexts.

Finite state machines (FSMs) are the oldest and most commonly used formalism to model game AI, wherein states represent behaviours and transitions are triggered to change the behaviour exhibited [9], [10]. Hierarchical FSMs (HFSMs) incorporate aspects of Statecharts [11] by allowing states to contain substates with internal transitions. In the context of reuse, superstates in HFSMs can be treated as modules and exported to new AIs [12]. This approach is valuable, but omits important details such as code portability, and has no provision for interaction with internal states. The strict hierarchical nature of HFSMs can be limiting as it places restrictions on how transitions between states can be modelled. This limitation is shared by behaviour trees [13], which although they more clearly delineate how the system chooses behaviours, are strongly hierarchical, and further suffer from a lack of modal states encapsulating different behavioural groupings. Modularizing behavioural components is based upon pruning and reusing branches, an idea which has been previously been explored [14]. In a practical sense, the extent of reuse in behaviour trees 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.

In theory, planning approaches seem ideal for reuse. Goal-oriented action planners, popularized by the game F.E.A.R. [15], choose behaviours using a heuristic search through a library of behaviour modules with pre- and post-conditions. In practice, getting the planner to select appropriate behaviours requires a proliferation of variables encapsulating basic knowledge, along with considerable tweaking of weights and heuristics. As a result, the modules become highly customized to the game context and general purpose reuse becomes more difficult as a result.

A. Layered Statechart-Based AI

Our work adopts the formalism developed by Kienzle et al. [2], who introduce an AI based on an abstract layering of Statecharts. This approach is inspired by the *sense-plan-act architecture* common in robotics. Here, each Statechart implements 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 Statecharts are ideal for reuse. An AI is built from multiple Statecharts, each embodying a specific concept, with the exhibited behaviour being a function of the superposition of states. Importantly, communication can occur at lower levels without involving higher levels, meaning the formalism employs a subsumption architecture [16]. This allows for improved modularity; higher levels of abstraction do not require a full set of knowledge to make decisions, which also reduces cross-cutting concerns.

Under this model, the lowest layer contains *sensors*, which read the game state typically through listeners or observers that generate events as changes are detected. Events are passed up to *analysers* 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*, most typically a single Statechart, which reacts to analysed and memorized data to decide upon a high level goal. Becoming less abstract, the goal triggers a *tactical decider* to determine how it will be executed. 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, which are corrected by *coordinators*. The final layer contains *actuators*, which execute actions by modifying the game-state.

In Fig. 1, a sample `FoodMemorizer` Statechart 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. 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 [17].

B. Reusing Statecharts

Our prior work proposed a strategy for reusing Statechart-based AI [1]. The fundamental reuse component was defined to be the *AI Module*, encompassing a Statechart and an

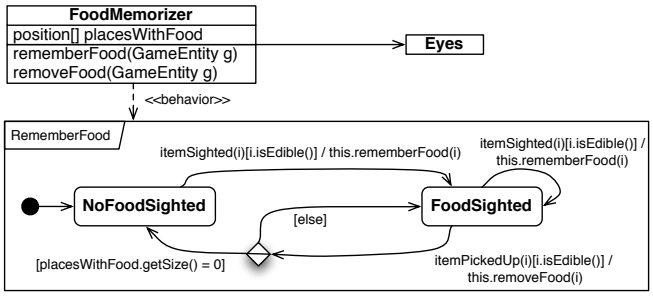

Figure 1. A Sample AI Statechart

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

Figure 2. The interface for the `EnemyTracker` AI module.

associated class. Next, an interface for the reuse of AI modules was designed, capturing the properties of a module relevant to reuse. Modules communicate primarily through *event-based message passing*, with events being marked as input, output, or internal, depending on the source module and target module for each event. As well, *synchronous calls* between associated classes allow for direct communication, for example retrieving information from a memorizer. Finally, modules may access the game directly, and thus the interface shows these interactions as well. Figure 2 gives a sample interface for an AI module that tracks enemies.

Reuse of AI modules allows for behavioural aspects of an AI to be exported, but the typical module is too fine-grained to capture a higher-level behaviour. *Functional groups* are thus defined to provide a structure for the composition of multiple modules, allowing behaviours to be reused as a single component. Importantly, a functional group can be given a composite interface identical in form to the interface for individual modules, and thus inserted interchangeably. As an example, the Tank Wars AI [2] contains several modules that together comprise the fuel management system. Thus, the `FuelTank` sensor, the `FuelStation` map, and the `RefuelPlanner` can be combined into a single `FuelManagement` functional group, ready for reuse in a new AI as a single group.

III. AI Module Reuse in Practice

Creating a new AI through reuse begins similarly to regular AI construction: the role of the AI within the game context is considered, then translated into a set of behaviours fulfilling that role. The thinking and planning that leads to the expression of these behaviours must then be implemented. At this stage, a reuse development approach diverges as it replaces implementation by importing existing AI modules wherever possible. In the absence of a sizable library of existing behaviours, it is unrealistic to assume that new AIs can be created purely through reuse, but as the library grows, the amount of reuse will increase, proportionally decreasing the amount of new development required.

A. Component Integration

Integrating components correctly and easily is the primary problem in AI reuse. Since AI modules work together in a layered architecture, integration demands the establishment of intra-module communication. In the case of Statecharts, communication typically occurs via event-based message passing, but could also be through a synchronous call. Thus, *connecting* an AI module can be defined as adding an AI module to a system such that it communicates with that system, or acts orthogonally to all other components. Since functional groups use the same interface as AI modules, they do not present a special case and the following integration strategies apply equally to both groups and modules. In our experience the complexity of inter-module communication is quite limited, obviating the need for module protocol management through constructions such as protocol state machines.

For event-based connections, modules communicate by pairing an input and output event. If a broadcast model is assumed, event renaming is typically sufficient, allowing connections to be formed by renaming the output event so that it matches the input event (or vice versa). To ensure correctness, existing names must be respected. If an already used event name is shared by a new module, events should be renamed to prevent the formation of unintended connections. Internal events, by definition, should never be used to form a connection. Since renaming an internal event can never break connections, it is always safe to rename an internal event when it conflicts. In the case of an event being used as both an internal communication and an output event, we recommend classification as an output event, with the goal of keeping those events classified as internal purely internal.

In some special cases, where multiple Statecharts already interact on the same event, renaming may create unintended connections or break existing connections. Additionally, there are boundary cases where renaming fails, typically in situations where events are used both internally and externally by multiple Statecharts. In these cases, targeted usage of narrowcasting will allow for safe event renaming. As an alternative, narrowcasting could be employed exclusively, though this creates more effort at the modelling level as connections must be managed.

Integration relating to the associated class is less forgiving, as an unsatisfied game import or synchronous method call will prevent compilation or cause 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.

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 is easily addressed there. 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 at an undetermined point in the future, then a synchronous call is appropriate.

Module reuse across different games is both possible and encouraged. The simplest case is when an AI modules is purely behaviour driven and has no game imports; these modules are *game-agnostic* and may be freely moved between games. If a module has game imports, then reuse in a new game will require updating all game imports to matching classes in the new game. This may not always be trivial or possible, and thus designing for reuse implies that modules be made game agnostic whenever possible. The process is much more complicated if the target game is coded in a programming language different than that of the module. Here, only the Statechart could be reused; the associated class would require a total rewrite.

B. Preconditions and Correctness

An AI module, when situated in an AI, can be highly dependent on the other modules within the AI. Reuse of such an AI module without all dependent modules could prevent that module from fulfilling its expected behaviour. For example, if a Statechart requires an input α or β before it can receive an event γ, but the target AI does not have Statecharts producing α or β, then it can never receive γ and the Statechart behaves as though it is unconnected. Alternatively, the acceptability of a system that can only produce one of α or β is unclear.

A solution to this would come in the form of a reachability analysis, where a thorough examination of the new AI at the model level can detect problems of this type. Specifics of this are out of scope of this paper, but facility for such analysis is integral to our design and future work.

IV. 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 the Tank Wars game in [2] 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.

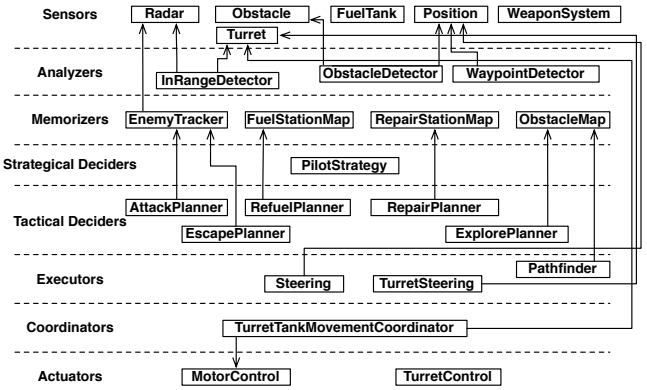

Figure 3. Detailed Tank Architecture, where lines represent synchronous calls between modules.

A brief overview of the Tank AI, developed for the Tank Wars simulation by Electronic Arts, is presented here; readers interested in full details are referred to the original paper [2]. The AI consists of 24 modules in total, divided across the layers as shown in Fig. 3. Each of the connecting arrows represents a synchronous call; connections arising from event-based message passing are not shown on the diagram. Behaviourally, the tank explores the game world looking for enemies, gas stations, repair stations, and obstacles, all the while memorizing locations of objects spotted. 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 [18]. Mammoth is written in Java, and uses SCXML Commons as the Statechart execution environment [19]. This version of the tank was written such that it 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 many of the 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.

The target AI seeks to model a squirrel, which in the Mammoth game world is intended to be a minor background character, moving through open spaces and collecting food

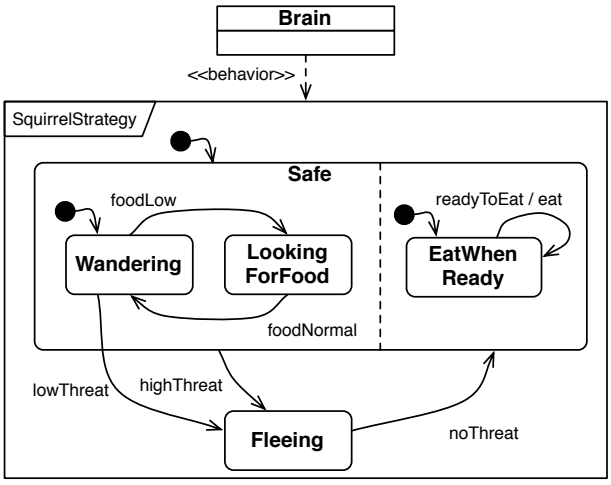

Figure 4. `SquirrelBrain` Module.

when it can. Characters in Mammoth are restricted by energy, which is consumed as the AI acts. For squirrels, it is restored by eating gathered food. 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.

A. Reuse in Practice

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, the `SquirrelBrain` shown in Fig. 4, was developed. The SquirrelBrain uses four high level goals: wander, look for food, flee, and eat. 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. 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 I. 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

Table I
MODULE MODIFICATIONS

Tank Module	Squirrel Module	Modifications
`EscapePlanner`	`Flee`	Synchronous call to `EnemyTracker` pointed at `NearbyThreats`
`RefuelPlanner`	`LookForFood`	*look_for_fuel* input renamed to *look_for_food* *move* output renamed to *take_item(Item)* *FuelStation* import replaced with *Item* import
`ExplorePlanner`	`Wander`	*explore* input event renamed to *wander*
`Waypoint-Pathfinding`	`Waypoint-Pathfinding`	All functional-group events reclassified as internal events.
`FuelStation-Map`	`SeenFood`	*FuelStation* import replaced with *Item* import Parameter *KeyItemType* set to type *Item*
`FuelTank`	`Energy`	Parameter *fuel* set to *energyLevel*
`Position`	`Position`	*Tank* import replaced with *Squirrel* import

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 through a functional group. The `Pathfinder`, `ObstacleMap`, `WaypointDetector`, and `Obstacle` sensor together perform waypoint-based pathfinding.

At the level of concrete actuators, the squirrel is quite different to 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

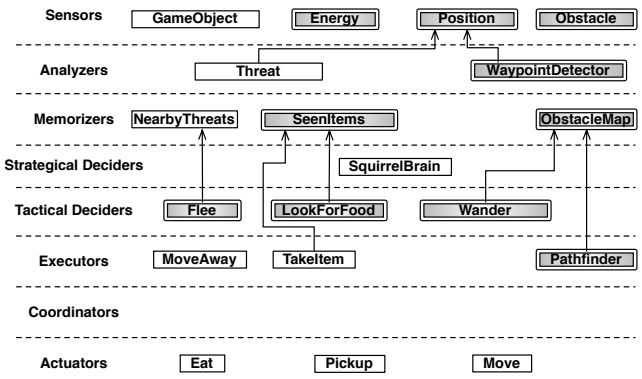

Figure 5. Detailed squirrel architecture; shaded modules are reused from the Tank.

`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` 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.

B. The New Squirrel

The resulting squirrel AI is shown in Fig 5. 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 will result in an improvement in development time, with more than half of the AI being reused. More 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 provide attacking behaviours.

V. 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. Development is ongoing and the tool will be released to the community upon maturation.

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

A. SkyAI Workflow

Usage of SkyAI begins by creating new modules. The user specifies the source SCXML Statechart and Java associated class, whereupon SkyAI reads the source file and extracts information about events, methods, and imports, finally adding the module to the library. Some information cannot be determined programmatically, such as internal event classification and some synchronous calls, so SkyAI displays the interface for the new module, allowing modification as necessary. When complete, the new module is added to the SkyAI module library and made available for reuse.

A new AI built from SkyAI is handled as a project, and must be assigned to a specific game. Modules are selected from the library listing and added to the project. Selecting a module here will allow the designer to make changes necessary for reuse, such as renaming events, or other modifications as found in Table 1. While the designer works, errors and warnings are generated, supporting the design process. When the new AI is complete, the AI is exported and reuse modifications are saved into new source files, ready for insertion into the target game.

B. 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 module connection. 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 covers issues at the interface level, and is shown in Table II.

VI. 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

Table II
THE LIST OF WARNINGS AND ERRORS GENERATED BY SKYAI

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 $class$, 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.

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.

ACKNOWLEDGMENT

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

REFERENCES

[1] C. Dragert, J. Kienzle, and C. Verbrugge, "Toward high-level reuse of statechart-based AI in computer games," in *Proceedings of the 1st International Workshop on Games and Software Engineering*, 2011, pp. 25–28.

[2] J. Kienzle, A. Denault, and H. Vangheluwe, "Model-based design of computer-controlled game character behavior," in *MODELS*, ser. LNCS, 2007, vol. 4735, pp. 650–665.

[3] Unreal Technology, "The Unreal Engine 3," http://www.unrealtechnology.com/html/technology/ue30.shtml, 2007.

[4] 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*. Eurosis, August 2005, pp. 33 – 38.

[5] Schwab, Brian and Mark, Dave and Dill, Kevin, and Lewis, Mike and Evans, Richard, "GDC: Turing tantrums: AI developers rant," http://www.gdcvault.com/play/1014586/Turing-Tantrums-AI-Developers-Rant, 2011.

[6] Laming, Brent and McGinnis, Joel, and Champanard, Alex, "Creating your building blocks: Modular component AI systems," http://www.gdcvault.com/play/1014573/Creating-Your-Building-Blocks-Modular, 2011.

[7] M. Dyckhoff, "Evolving Halo's behaviour tree AI," Presentation at GDC, 2007, http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf.

[8] Walker, John and Zubek, Robert, and Carlisle, Phil, "Little big AI: Rich behavior on a small budget," http://www.gdcvault.com/play/1012483/Little-Big-AI-Rich-Behavior, 2010.

[9] D. Fu and R. T. Houlette, "Putting AI in entertainment: An AI authoring tool for simulation and games," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 81–84, 2002.

[10] S. Gill, "Visual Finite State Machine AI Systems," Gamasutra: http://www.gamasutra.com/features/20041118/gill-01.shtml, November 2004.

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

[12] J. 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.

[13] D. Isla, "Handling complexity in the Halo 2 AI," *Game Developers Conference*, p. 12, 2005. [Online]. Available: http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml

[14] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game DEFCON," in *Applications of Evolutionary Computation*, ser. LNCS. Springer, 2010, vol. 6024, pp. 100–110.

[15] J. Orkin, "Three states and a plan: The AI of F.E.A.R." in *Proceedings of the Game Developer's Conference*, 2006.

[16] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14 – 23, Mar. 1986.

[17] P. Kolhoff, "Level up for finite state machines: An interpreter for statecharts," in *AI Game Programming Wisdom 4*, S. Rabin, Ed. Charles River Media, 2008, pp. 317–332.

[18] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker, "Mammoth: A Massively Multiplayer Game Research Framework," in *4th International Conference on the Foundations of Digital Games (ICFDG)*. New York, NY, USA: ACM, April 2009, pp. 308 – 315.

[19] Apache Commons, "Commons SCXML," http://commons.apache.org/scxml/, November 2010.