Scythe AI: A Tool for Modular Reuse of Game AI

Christopher Dragert and Jörg Kienzle and Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada

Abstract

Scythe AI is a tool designed for game AI developers to enable modular reuse of NPC AI. In this demonstration, we show how Scythe AI can be used to compose and manage statechart-based AI-Modules to produce new AIs for a target game. The tool includes a streamlined importing process to introduce new modules, library management of modules, an error and warning system to assist in new AI construction, and can export directly to a defined target game.

Introduction

Modular reuse of game AI provides numerous benefits to the game developer, chiefly by reducing development time and cost. Without effective reuse, AI for non-player characters (NPCs) must be built from the ground up. Instead of perfecting high level behaviours, time is spent reimplementing basic functionalities. With Scythe AI, we seek to provide enable game developers with the ability to reuse behavioural modules encapsulating components of AI logic.

The past several years we have explored how finite state machine AIs (FSMs) can be tailored for effective reuse, in particular through the use of modular statecharts. One major shortcoming of FSMs is the rapid increase of complexity as state machines grow in size. However, we have found this complexity growth can be limited by constructing AIs as a composition of many small statecharts, each modularizing an element of behaviour. We organize our AIs using a layered statechart model, and have found that this design approach leads naturally to reusable components.

Unlike other game AI tools, such as Unreal Kismet or pathfinding middleware, Scythe AI directly addresses the reuse problem. Instead of authoring AI components, Scythe AI focuses on building interfaces for existing FSM-based AI modules, and integrating modules in new game contexts. Developers using Scythe AI are freed from reimplementing basic behaviours, and can instead focus their efforts on implementing and perfecting new high-level behaviours.

Scythe AI: The Tool

Scythe AI is a tool specifically designed for AI developers to manage AI reuse. At a high level, Scythe AI allows users to build a library of AI-modules by importing statecharts and associated classes. These modules are then reused to build new AIs. Like a standard IDE, warnings and errors are generated to guide users through the process of connecting modules to ensure proper interaction. When an AI is completed, it can be exported directly from Scythe AI into to a target game. The workflow is illustrated in Figure 1.

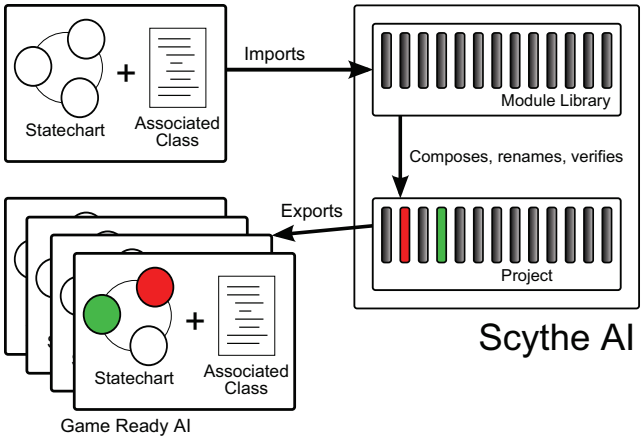

Figure 1: Scythe AI workflow

As a formalism, FSMs are loosely defined. Some allow hierarchy (HFSMs), some execute actions when changing states, some use guards on transitions, and so on. All of these approaches are generalized by statecharts. While we have based our tool on statechart-based AI, and explicitly use statechart formalisms, it is appropriate to consider Scythe AI as a tool to manage reuse of state machine AI.

Importing AI Modules

After extensive research into the nature of AI-module communication (Dragert, Kienzle, and Verbrugge 2012), we have developed an *AI module interface* that covers all aspects of module communication, including input and output events, synchronous event calls, and module parameters. This interface is the primary artifact used by Scythe AI to manage modules. Much of the logic of an AI-module lies in the functions called by that module. A pathfinding module would receive events telling it when to find a path, but the

logic to do so (an A* algorithm for instance) would reside in an associated class. Occasionally functions are shared between classes using synchronous calls. To capture both logic and implementation interactions, we define an *AI-module* as a statechart and an associated class, and communication between modules includes both event-based message passing and synchronous calls.

Importing an AI module into Scythe AI consists of building the AI interface by working through a highly guided process. Event-based communication details are recovered directly from the statechart itself, supplied in the SCXML format. Scythe AI scans the statechart for all events that are generated or transitioned upon and suggests these as output or input events respectively. Any event that is both created and consumed is an internal event, and is inappropriate for connection of any type.

The remainder of the AI module interface comes from the associated class. Currently, Scythe AI supports Java files, and could be extended to support C++. When importing, the associated class is parsed and information relevant to reuse is extracted. The user is given the list of class imports, where the user determines which imports come from, and thus link the module to, the source game. As well, any parameters in the class are extracted, so that values can be specified when that AI module is reused. Synchronous calls are addressed next, covering calls made to other classes, and calls made available for other classes to utilize. When this guided process is complete, the new AI module is added to Scythe AI's module library, and copies of the associated class and statechart are made and stored.

Building a new AI

The building process consists of adding modules from the library to the new AI. Assuming that a module's logic is internally correct, i.e., the statechart correctly does what it claims to do, the task for the user is to coordinate module communication such that the modules in the new AI will cooperate as desired. Most frequently, this means ensuring that events sent by a statechart are correctly received by target statecharts, while not being incidentally received by non-target statecharts. This is done by event renaming as described in (Dragert, Kienzle, and Verbrugge 2012). Once a module is added, values for parameters should be specified.

A number of issues can arise while constructing a new AI. These are classified as *errors* if they will prevent the AI from running (e.g., a synchronous call is not satisfied), and must be corrected before completion. An issue is classified as a *warning* if it is a likely source of behavioural error, but would not prevent the AI from running. Each issue is listed in the main interface so that the user has a continual description of potential problems. Table 1 gives Scythe AI's current listing of warnings and errors.

Output an AI

Once the AI is complete, the final step is to output the AI. For a typical game, this will be in the form of a class or .xml file defining the AI. Scythe AI automatically builds this file from the constructed AI. Output also includes both the .scxml statechart and the source Java file for each AI module. Scythe AI will modify output .scxml files to account for any event renaming that has occurred. The source game must be able to accept these files as input.

Table 1: Warnings and errors generated by Scythe AI

Severity	Problem	Description
Error	Event Interference	Event e is private in 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	Game Conflict	Module x has input event e sourced by the game, but module y generates e
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.
Warning	Unused call	Module x provides method m which is never called.
Warning	Null Parameter	Parameter p in module x is null.
Warning	No Actuators	Project has no actuators.

In our experimentation, we output to the game Mammoth, an MMO research framework. Importantly, Mammoth allows NPCs to be defined externally using XML files, is written in Java, and has an SCXML execution environment. Thus, our output profile for Mammoth writes the external XML definition, which delineates the exact SCXML/Java pairings and provides specific values for parameters.

Conclusions

Based on our experience, Scythe AI is an effective tool to manage reuse of statechart-based AI. We have modelled the AI for several NPCs within Scythe AI, and found it to be an excellent environment for the rapid composition and deployment of game AI.

Future work on Scythe AI will explore validation of the constructed AI. When an AI is assembled and all modules are communicating correctly, there is still no guarantee that the emergent behaviour will meet specifications. We are currently investigating how Scythe AI can be used to effectively validate an AI, specifically through the use of model-checking. Along the same lines, we would like to explore the ease of use of the software, and perform user studies that validate our positive experiences so far.

Additionally, many quality of life features can be added to further improve the usage experience. Specific wish-list features include tagging modules with intended purposes to make a searchable library, adding visualization of module connections, adding output profiles for new game environments such as Unity, and adding statechart compilation to remove the need for SCXML support in the target game.

References

Dragert, C.; Kienzle, J.; and Verbrugge, C. 2012. Reusable components for artificial intelligence in computer games. In *Proceedings of the 2nd International Workshop on Games and Software Engineering*, GAS '12, 35–41.