AIIDE 2012 Doctoral Consortium
AAAI Technical Report WS-12-18

Model-Driven Development of Game AI: Research Plan

Christopher W.A. Dragert
McGill University
Montreal, QC, H3A 2A7, Canada
christopher.dragert@mail.mcgill.ca

Abstract

The field of game AI is largely industry driven, lacking an agreed upon formalism for AI representation. Ad-hoc scripting languages, simple finite state machines, behaviour trees, and planners are employed, but not in a fashion adhering to any standard. As a result, reuse is sparse between games and formal analysis techniques are undeveloped. As research for a Ph.D. thesis, we propose to show that a *layered Statechart-based AI* is a suitable formalism for Game AI, enabling the use of model-driven development techniques such as reuse and high-level analysis including model-checking.

The fundamentally modular nature of this approach leads naturally to reuse as a fundamental component of the design process. Supported by a clearly defined formalism, useful behavioural analyses become possible, such as testing reactions to various inputs at design time. We also explore transformations at the modelling level to enable procedural generation, allowing rapid deployment of varying AIs. Additionally, such a model allows for the generation of efficient code that can be directly inserted into games. Tool support for reuse, generation, and analysis will be developed, then employed in creating an industrial scale AI, proving that this formalism is appropriate for industrial use.

Game AI typically focuses on the control of non-player characters (NPCs) such that they exhibit the behaviours required to fill their role within the game context. Most frequently each NPC is controlled by a simple reactive agent that translates input from the game state into NPC actions. The AI is typically developed for a single game, and portability of implemented AI is largely non-existent. This type of narrow game-by-game focus is a source of consternation for game developers. At GDC 2011, well known AI developer Kevin Dill argued this point, saying that the lack of behavioural modularity was stymying the development of high quality AI (Schwab, Brian and Mark, Dave and Dill, Kevin, and Lewis, Mike and Evans, Richard 2011). With no standard, there is no agreed-upon format for behaviour, and so 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. A model-driven development approach would directly address this, allowing for modularity through component reuse, and brings along additional benefits by permitting high level analysis including model-checking, and code generation.

We believe the fundamental cause for the absence of modularity and reuse in game AI is the lack of a formalism suitable for the application of software engineering techniques. Scripting approaches are very context specific and lack a high-level model that would form the framework for higher-level reasoning. Games that do employ a formalism tend to use finite state machines (FSMs), which tend to become overly complex, or behaviour trees, which intertwine behaviours inhibiting any push towards modularization.

An appealing alternative is offered by Kienzle et al. (Kienzle, Denault, and Vangheluwe 2007), who introduce an AI based on an abstract layering of Statecharts. Here, each Statechart acts as a modular component implementing a single behavioural concern, such as sensing the game-state, memorizing data, deciding upon high-level goals, and so on. Unfortunately this formalism lacks exposure and is not currently employed by industry, but we believe that this formalism is appropriate for exploring model-driven development of game AI, and intend to demonstrate its suitability for game AI through the development of a large scale AI.

The fundamental goal of this research is to bring model-driven development to game AI through the use of layered Statechart-based AI. We will show how such an approach leads easily to reuse by allowing for modular design. Statechart models will allow us to perform useful behavioural analyses, allowing game developers to test and verify their creations at design time. To show appropriateness to the domain, we will explore interesting transformations at the modelling level that will permit procedural generation of varying AIs. Implementation will take the form of *SkyAI*, a tool to manage model-driven development of game AI. The remainder of this report will address each of these aspects in turn.

Modular Reuse

Development of computer games would benefit tremendously from a standardized format for game AI. Behavioural modules could then be reused across games, reducing development time and effort, and freeing AI developers from reimplementing basic functionalities. Open source modules

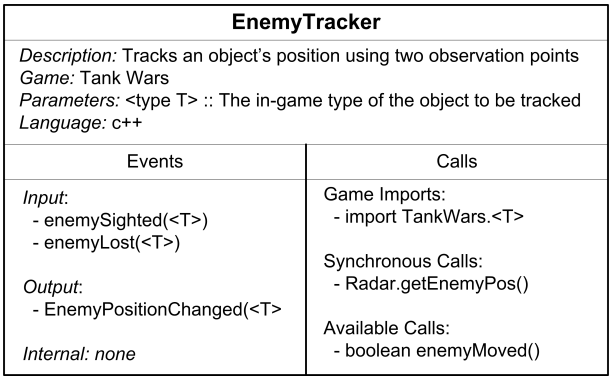

Figure 1: *A sample AI-module interface*.

could be shared, enabling independent and nascent developers to download AI behaviours, streamlining development. By modularizing the logic that underlies a specific behaviour, such as fleeing from an enemy or picking up an item, future AIs can be constructed by integrating existing behavioural modules.

This type of reuse approach places specific demands on the architecture. Monolithic structures become impractical, as they lack clear behavioural separation. For instance, deciding if an NPC should flee should be thought of a different behaviour than deciding how to flee, which in turn is a different behaviour than deciding where to flee, and so on. As well, nothing in these behaviours should refer to external behaviours, meaning that none of the flee behaviours should reference combat, for instance. Behaviour trees tend to be monolithic, and often have cross-cutting concerns, making them ill-suited to such an approach.

The layered Statechart-based AI approach clearly suggests the Statechart as the reuse component. This approach was first explored in (Dragert, Kienzle, and Verbrugge 2011). In that position paper, we investigated how Statecharts communicate and interact with each other. Primarily, this is by generating events and broadcasting them, triggering transitions in receiving Statecharts. However, the associated classes of the Statecharts can make synchronous method calls between one another, creating a second type of interaction. Defining an *AI-module* as a Statechart and its associated class, the unit encapsulates both types of communication and led to the creation of an interface for AI modules, detailing how a given AI-module interacts with other AI-modules. Furthermore, it proved possible to compose AI-modules as a *functional group* with an interface identical in form to the interface for a single module. A sample interface is given in Fig. 1, showing how an enemy tracking module communicates using input and output events, synchronous calls, and also noting game imports and run-time parameters.

We further investigated modular reuse by showing how this work is applied in practice. In recently published work (Dragert, Kienzle, and Verbrugge 2012a), we applied this approach to reuse significant portions of the AI controlling a tank to create a new AI controlling a squirrel. The two unrelated roles were selected deliberately, showing that even unrelated NPCs have behavioural similarities. Several behaviours (such as pathfinding, sensing enemies, and fleeing) were shared between the two NPCs, and thus we were able to reuse 10 modules from the tank. The resulting squirrel had a total of 19 modules, meaning less than half of the behaviours for the squirrel had to be newly coded. The integration process involved renaming events, redirecting synchronous calls, and modifying run-time parameters.

As well, the modular approach simplifies porting between games by listing game dependencies clearly in the module interface. Some AI-modules, through their associated classes, include classes from the game at-large, enabling tasks such as examining specific properties of an in-game object, or analyzing terrain. These modules can be ported by working through and updating all game dependencies listed in the interface. However, other AI-modules work purely based on communication from other AI-modules and include nothing from the game. These we call *game-agnostic* modules, and note that they can be freely reused between games sharing an implementation language and AI structure.

Analysis

One significant advantage of model-driven development is the ability to perform high-level analysis. For a computer game AI, the appropriate level of abstraction is at the behaviour level as this is the same level as the design goals. Most usefully, an analysis should be able to answer questions such as 'can the NPC move?', or 'will the NPC engage enemies?'. In model-driven development, these questions are treated as formal specifications, and could be encoded in *Linear Temporal Logic* (LTL) or *Computational Tree Logic* (CTL). Then, a systematic analysis of the model can verify that the model meets these specifications. We intend, as a key part of this research, to create a comprehensive approach to the analysis of a layered Statechart-based AI.

In the case of AI module reuse, specifying NPC behaviour is problematic because the NPC behaviour is emergent. Each NPC will behave based upon the combined behaviours of the modules present, and there is no general 'correct' behaviour. While it would be possible to verify individual modules (for example, verifying that when a `flee` module receives a *flee* event, a *move* event is eventually generated), in our experience AI modules are only a few states in size and such verification would be of questionable value.

One approach might be to have specifications for each module, and compose these as modules are composed. However, this runs the risk of unwieldy specifications, and it is unclear how exactly these specifications would relate to design goals. A more appropriate approach might be to specify a few basic behaviours for a constructed NPC that could be verified, such as the ability to move or react to other players.

A potential technique for verification is what we call *capability testing*, which is a determination of the behaviours the AI could possibly exhibit. At the modelling level, this is a *reachability* analysis where we ask if certain transitions are reachable given the events generated by the Statecharts present. A typical property to verify would be to ensure that

an actuator in the system can act. Here, CTL seems appropriate as this could be encoded as $EF(actuate)$, meaning there exists a path where $actuate$ eventually happens.

Verification of such specifications requires model-checking, a complete exploration of the state-space of the model. Model-checking of Statecharts was explored by Schäfer et al. (Schäfer, Knapp, and Merz 2001), where they created the tool HUGO that accepts a Statechart as input and generates PROMELA code suitable for use with the SPIN model-checker. UML collaborations are used as specifications, which are transformed into Büchi automata (never claims) accepted by SPIN. As a complicating factor, our formalism allows synchronous message passing permitting communication without events. The transformation in HUGO explicitly assumes this is not the case, so extra work will be required to generate PROMELA code. Other analysis methods, such as (Pap et al. 2001), seek to map out the expected behaviour for all sets of input, addressing completeness and consistency of the model. This does not require a full exploration of the state-space and may provide more timely analysis. It is unclear at this point how such tests would be valuable in verifying behaviour.

This work will be evaluated through testing. First off, if given a flawed AI (e.g., one that will never flee), and a requirement that the AI should flee under certain conditions, can our analysis identify the known error? In formulating these questions, both liveness and safety conditions will be tested for. If the AI is thought to be good and verifies as such, does it in fact work correctly in the game, or are there implementation concerns? Since the analysis is intended to be a development aide, it must complete in a reasonable time, and specification creation should not be overly complex.

Model-based Variation

Rapid prototyping and development of AI is an important problem in the game AI domain. To facilitate this using the layered Statechart formalism, we investigated how a single AI could be used to generate new AIs. Our research (Dragert et al. 2011) has explored the generation of new variations of an AI by performing modifications at the model level. This is well-suited for creating interesting populations of recurring characters, such as townspeople, basic enemies, and wildlife, enriching the gameplay experience by giving variation and personality to existing AIs.

Starting with a Statechart-based layered AI, variations at three level of abstraction are possible: parameter modification; addition, removal, or swapping of Statecharts; and rule-based transformation of Statecharts. The goal is the creation of many different versions of a AI, each able to fill the original role, but with variations adding flavour and life. Generation occurs by applying a subset of alterations to an existing AI, while variation arises by using different subsets of alterations to generate each AI. Semantic correctness can be ensured by using only safe transformations, or by allowing any transformation but then culling unacceptable variations.

These three levels allow for a wide range of variation. Parameter-based variations change simple properties of the AI, for example, changing the threshold at which the AI enters a low-health state, or switching the item that an NPC collects. Adding and removing Statecharts allows for the introduction or removal of behaviours, such as removing a flee behaviour to get a brave AI, or adding in a low-morale module that causes fleeing at additional times to get a cowardly AI. As well, Statecharts can be swapped, meaning that behaviours can be replaced to get a different expression of the same behaviour. As an example, a targeting module could be replaced, changing the priority in which an NPC attacks enemies. Finally, modification of the Statechart itself through a rules-based graph transformation fundamentally alters the behaviour of a module. For instance, a reset behaviour can be added to initialize a behaviour, new transitions can be added to make the Statechart respond to different events, and so on.

Implementation

Implementation will take a two-pronged approach. In order to effectively explore the design and usage of a layered-Statechart based AI, we need to have a large-scale reference AI. With that in hand, the next step is to develop a tool to manage the model-driven development process. Our tool is called SkyAI, and is designed to manage layered Statechart-based AIs and handle module reuse, along with model-level analysis.

Case Study

Too often, academic research produces toy examples as proof of concept, but these are ultimately unconvincing of adaptability to a full scale implementation. By producing an AI similar to state of the art industrial AIs, we would provide convincing proof that a Statechart-based modular AI can be the basis of a high-quality AI. As guidelines, we look to the highly publicized and influential AI featured in Halo (Isla 2005). The Halo AI uses the behaviour tree formalism. Since the AI starts with a small set of high level goals that eventually lead to low level actions, the structure can be imitated in a modular fashion using our layered approach, yielding similar functionality. Work on this is largely complete (Dragert, Kienzle, and Verbrugge 2012b).

We were able to create an AI using a total of 49 AI-modules that together replicates many of the behaviours in the Halo AI. During this process, we discovered and described several recurring Statechart patterns. Using the reuse approach described above, we were able to easily reuse these modules thereby reducing design efforts. These patterns represent a highly useful tool in the creation of future AIs, and are a valuable contribution to the community.

The resulting AI represents, at the behaviour level, our best attempt to replicate the Halo AI based upon various presentations of the Halo. While this sacrifices accuracy, it ensures our work can be used outside of any proprietary context. One major drawback is that many of the attendant algorithms, such as finding cover and pathfinding, were not implemented in any meaningful way and thus the AI falls short of being an actual implementation. This presents issues with regards to verifying memory usage, for example. Regardless, with the work complete at the modelling level, it provides an excellent basis for research and testing into various aspects of model-driven AI development.

The SkyAI Tool

As a tool to enable model-driven development of game AI, SkyAI will be a major contribution of this research. The end goal is to allow a user to grow a library of AI modules, and build layered AIs by adding and connecting modules. Much of the basic functionality of SkyAI has already been developed (Dragert, Kienzle, and Verbrugge 2012a). While the tool itself is in a pre-alpha state, it already shows promise.

SkyAI uses an abstract representation of the AI module, building each module from their source files with guidance from the designer. Right now, 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. The module itself is stored in an XML format, and managed along with the source files by SkyAI.

As a support feature, SkyAI has a complete error and warning system. A number of potential issues arise when building a new AI through module reuse, primarily related to connections between modules. These are classified as *errors* if they will prevent the AI from compiling. An issue is merely a *warning* if it is a potential source of behavioural error, but will not prevent the AI from functioning. These are listed in the main display, similar to IDE warning systems found in Visual Studio or Eclipse.

The end state of SkyAI with respect to this research is to be a comprehensive implementation of the ideas in this thesis. As an AI is being constructed, SkyAI will provide model-level analysis to guide the designer. Once an AI is completed, SkyAI will provide the ability to export an efficient optimized version for insertion into the target game. This will accommodate event renaming and other modifications arising from module connection. Procedural generation of AI variations will be also be enabled through Sky AI.

Conclusions

The proposed research seeks to apply model-driven development practices to computer game AI. By using the Statechart and associated class as the fundamental module of reuse, and constructing AIs using the layered Statechart formalism, we have already developed a reuse strategy for game AI, and investigated procedural generation of varying AIs. We intend to complete this work and show the value of the model-driven approach by developing useful analysis that can guarantee correctness at the behavioural level. The overall viability of the approach will be demonstrated by developing a large scale AI suitable for industrial use, all within the tool SkyAI that will enable AI-module reuse and analysis. On the whole, this research provides a balanced, comprehensive approach to the exploration of model-driven development techniques to game AI.

References

Dragert, C.; Kienzle, J.; Vangheluwe, H.; and Verbrugge, C. 2011. Generating extras: Procedural AI with statecharts. Technical Report SOCS-TR-2011.1.

Dragert, C.; Kienzle, J.; and Verbrugge, C. 2011. Toward high-level reuse of Statechart-based AI in computer games. In *Proceeding of the 1st international workshop on Games and software engineering*, GAS '11, 25–28.

Dragert, C.; Kienzle, J.; and Verbrugge, C. 2012a. Reusable components for artifical intelligence in computer games. In *Proceeding of the 2nd international workshop on Games and software engineering*, 35–41.

Dragert, C.; Kienzle, J.; and Verbrugge, C. 2012b. Statechart-based AI in practice. In *The Eighth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (to appear)*.

Isla, D. 2005. Managing complexity in the Halo 2 AI system. In *Proceedings of the Game Developers Conference*.

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

Pap, Z.; Majzik, I.; Pataricza, A.; and Szegi, A. 2001. Completeness and consistency analysis of UML statechart specifications. In *Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'2001*, 83–90.

Schäfer, T.; Knapp, A.; and Merz, S. 2001. Model checking uml state machines and collaborations. *Electronic Notes in Theoretical Computer Science* 55(3):357 – 369. Workshop on Software Model Checking (in connection with CAV '01).

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