Statechart-Based AI in Practice

Christopher Dragert and **Jörg Kienzle** and **Clark Verbrugge**
McGill University
Montreal, QC, H3A 2A7, Canada
chris.dragert@mail.mcgill.ca
{*joerg.kienzle, clark.verbrugge*}*@mcgill.ca*

Abstract

Layered Statechart-based AI shows considerable promise by being a highly modular, reusable, and designer friendly approach to game AI. Here we demonstrate the viability of this approach by replicating the functionality of a full-featured and commercial-scale *behaviour tree* AI within a non-commercial game framework. As well as demonstrating that layered Statecharts are both usable and amply expressive, our experience highlights the value of several, previously unidentified design considerations, such as sensor patterns, the necessity of subsumption, and the utility of orthogonal regions. These observations point towards simplified, higher-level AI construction techniques that can reduce the complexity of AI design and further enhance reuse.

Introduction

As recently as GDC 2011, it was argued that the lack of behavioural modularity stymies the development of high quality AI (Dill 2011) for non-player characters (NPCs). Improvements to modularity and reusability tend to focus on either game or engine specific properties, staying away from general investigations at an architectural level. Moreover, little attention has been paid to the actual development process and how large-scale, complex AIs can be constructed using a fundamentally modular approach.

In part, this follows from the proliferation of agent-based AI architectures used for NPC AI. Since none are universally superior, a wide range of formalisms are employed and studied. Historically, scripting approaches and various finite-state machines (FSMs) have been used, but more recently *behaviour trees* (Isla 2005) and *goal-oriented action planners* (GOAP) (Orkin 2006) have emerged as viable options. All have drawbacks: behaviour trees have a fundamental difficulty in expressing common stimulus or event-driven behaviours; scripting approaches tend to be game specific; FSMs can become unmanageably complex due to state-space explosion; and planners are notoriously fickle, requiring encapsulation of basic knowledge as heuristics in an effort to tease out intended behaviours.

Recent work on *Statechart-based AI* (Kienzle, Denault, and Vangheluwe 2007; Kolhoff 2008) offers significant improvements to the complexity problems encountered by FSMs approaches. Through the use of substates, history states, and orthogonal regions, transition proliferation is limited, while modular layering controls state density. At the same time, full Statecharts are more portable and reusable than FSMs or hierarchical FSMs, since the feature set is consistent. Finally, the layered Statechart approach is fully compatible with subsumption approaches. That being said, Statecharts have not at this point been explored deeply by the video game industry, with criticisms focusing on comprehensibility (Schwab 2008), the existence of only small examples, and concerns as to scalability.

In this work we develop an extensive Statechart-based AI model for a non-trivial and commercially relevant game AI, derived from the behaviour tree implementation described for the *Halo* series of computer games (Isla 2005). As well as providing a practical demonstration that Statecharts have sufficient and appropriate expressiveness for such a large-scale and complex AI, our efforts reveal useful and interesting design considerations. This includes the classification of several behaviour patterns, best-practices for efficiency, as well as an examination of the complexity of the resulting AI model. Specific contributions include:

- The development of a non-trivial Statechart implementation of a commercially relevant computer game AI. Ours is the first work in this area to present results based directly on using a full-scale AI design derived from a modern computer game, thereby demonstrating the suitability of Statecharts to game AI design, while providing a practical and meaningful testbed for future work in the area.
- The classification of several modular behavioural patterns for usage in AI development, along with the description of several techniques that manage complexity and efficiency of the constructed AI.
- A demonstration of how a variant of subsumption can be used to limit Statechart complexity without increasing coordination requirements.

Background and Related Work

In modern computer games, AI most frequently comes in the form of reactive agents used to control NPCs such that they exhibit behaviours relevant to the character's role

in the game context. This is referred to as *computational behaviour* or *computational intelligence*. The focus is on choosing behaviours either through arbitrary code expressed via a custom scripting context (Onuczko et al. 2005), or relatively simple tree or graph structures such as decision trees. Abstract models such as FSMs are commonly used, wherein events cause the NPC to exhibit behaviours based upon the current state in the FSM (Fu and Houlette 2002; Gill 2004). Hierarchical FSMs (HFSMs) are commonly employed in industry, encapsulating behavioural sub-tasks by allowing states to contain substates with internal transitions.

Alternative computational intelligence models are found in the robotics community where agent-based control is commonplace (Wooldridge and Jennings 1995). The *subsumption architecture* proposed by Brooks (Brooks 1986) is highly influential, wherein low layers can express behaviours independent of higher layers. This allows for basic reactivity without involving high level reasoning. Importantly, subsumption techniques are not mutually exclusive with HFSMs. It has been shown that when combined, hierarchical subsumption architectures have significantly lower representational complexity than behaviour trees, FSMs and HFSMs, and pure subsumption architectures (Heckel, Youngblood, and Ketkar 2010).

Our work adopts the formalism developed by Kienzle et al. (Kienzle, Denault, and Vangheluwe 2007), who propose an AI based on an abstract layering of Statecharts. The *layered Statechart-based* formalism describes a hybrid agent, where a reactive agent is supplemented with limited memory. 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, with techniques having been developed for reuse specifically in a game context (Dragert, Kienzle, and Verbrugge 2011; 2012). This work improves on the original architecture through the introduction of a variant of subsumption to manage state-space explosion and limit state complexity.

Statecharts themselves generalize FSM and HFSM models. Transitions are of the form $e[g]/a$, where e is a triggering event, g is a guard condition, and a is an action executed in response. States can have substates, and transitions are possible from substates and enclosing states. Multiple transition matches are resolved inside-out (i.e., child first). After leaving a nested state, history states, denoted with an H, allow for a return to the previous substate. Statecharts can have orthogonal regions, allowing for the Statechart to concurrently be in a state in each orthogonal region, and to independently make transitions within those regions.

Behaviour Trees and the Halo AI

Behaviour trees (BTs) recast HFSMs into a strictly hierarchical decision tree. While they clearly delineate how the system selects behaviour, the strict hierarchy impairs reactivity and lacks modal states that would encapsulate different behaviour groupings. Recent advances, such as event-driven and data-driven BTs improve efficiency (Champanard 2012), but sidestep reactivity issues through parallel nodes.

As the first popular commercial game to employ BTs, the AI for the Halo trilogy was well received and highly publicized (Isla 2005; Dyckhoff 2007; 2008). Many approaches derive from the Halo implementation, with the AI for the game *Spore* explicitly doing so (Hecker 2009). Halo is an FPS game, where the player fights groups of aliens with the help of an allied squadron. AI controlling the NPCs organizes behaviours under high-level functions: search, combat, flight, self-preservation, and idle. Each of these contains subtasks; combat, for instance, decides between grenade use, charging, fighting, searching, and guarding. The tree for Halo 2 has a maximum depth of four, with the bottom layer consisting of leaf-nodes that execute concrete behaviours and trigger corresponding animations. Nodes can be cross-linked acyclically allowing a single behaviour to appear under multiple nodes.

Designing the AI

The goals in developing a Statechart-based version of the Halo AI were as follows: to capture the basic behaviour of the reference AI without sacrificing key functionality, while showing how reuse practices and modularity lead to a well-constructed AI. The Halo AI was chosen due to its visibility as an example of good AI design, and its success as a commercial title. By developing a similar AI, we can credibly conclude that layered Statecharts are capable of handling industrial scale AIs.

The model is divided into 8 layers: sensors, analyzers, memorizers, strategic deciders, tactical deciders, executors, coordinators, and actuators. Events originate at the sensors, flowing up through the input layers to the strategic decider. A high level goal is selected, then communicated down through the output layers, ending up at the actuators which issue commands causing the NPC to act. Every layer contains one or more separate Statecharts that each manage a specific aspect of behaviour. During the design process, it was common to discover several isomorphic Statecharts at a single layer. Using one as a pattern, it was possible to implement other behaviours by event renaming, allowing for modular reuse of a working behaviour. Since this was a significant time saver during the design process, these patterns are valuable as a contribution to others using this approach.

A key update to the layered Statechart approach was the introduction of subsumption. Typical subsumption creates a coordination problem when low level reactions conflict with higher level behaviours. We avoid this by construction, never allowing lower levels to enact behaviours without prior permission. Instead, low level sensors and actuators communicate information about the game-state to the output layers in order to 'preset' the behaviour. For example, a *WeaponSensor* would send information about the equipped weapon to a tactical decider. When the high level goal to engage is chosen, the *CombatDecider* has already been preset to either melee or ranged combat, and immediately executes the appropriate choice. Thus, tactical details are subsumed, lower levels never take spontaneous action, and the need for coordination is largely obviated.

The number of Statecharts in the model prevents a complete exhibition; only the most interesting details are de-

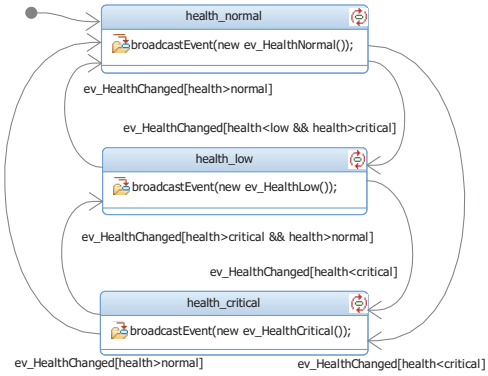

Figure 1: The *HealthSensor*.

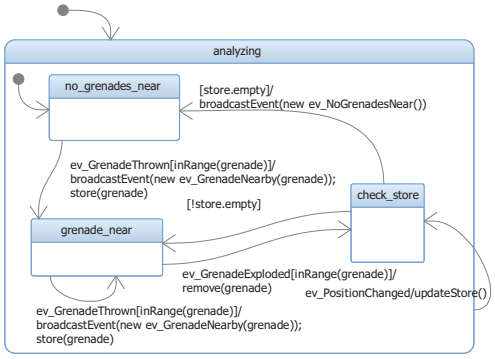

Figure 2: The *GrenadeProximityAnalyzer*.

scribed herein. Interested parties are referred to the complete set of models, freely available from http://gram.cs.mcgill.ca for external examination and experimentation.

Input Layers

The sensors are responsible for transforming game events and data into events used within the Statecharts. Sensors were implemented as listeners for efficiency, rather than actively polling. Upon receiving a callback, Sensors generate an event and pass it to the other Statecharts.

Of note, we found two Statechart patterns while creating sensors. The first we called a *discretizing sensor*, which maps a continuous value to discrete events. The number of states is equal to the number of discrete levels needed. Transitions between the states have guards constructed from desired threshold values. The *HealthSensor* given in Fig. 1 is an instance of a discretizing sensor with 3 states. Instead of relying on guards, the `ev_HealthChanged` event is generated internally in response to changes to health, thereby preventing inefficient polling.

The second behavioural pattern was a sensor mapping in-game events to AI events, creating a bridge between the game and the AI. This was typically a state-independent transformation, yielding a trivial Statechart with a single state and no. We call these *event-mapping sensors*, and use these as often as possible due to their overall efficiency. However, in a case where event-mapping is state-dependent, such as having on/off states, the event-mapping sensor can be expanded to have a second state that does not generate events, and appropriate transitions between.

Analyzers construct a high level view of the game-state using sensed data as input. For example, a `ev_PlayerSpotted` event could be analyzed and result in an `ev_EnemySpotted` event, which could in turn result in an `ev_EnemyInMeleeRange` and so on. One pattern emerged at this level, which we named the *Binary Analyzer*. An instance is shown in Fig. 2, where the analyzer creates an event for each nearby grenade, but does not give the all clear until all grenades are out of range. After each unit is removed from tracking, or the NPC moves, the binary analyzer enters a state with two outbound transitions having mutually exclusive guards, ensuring the condition is only checked when a change is possible.

Strategic Decider

At the highest level of abstraction, our *StrategicDecider* uses states to store current goals: changing states implies that a new goal has been selected. This is communicated through the creation of an event using the on-entry block of the state. When a state is exited, an on-exit action creates a stop event notifying downstream Statecharts that they should cease current behaviours.

This approach proved very practical. Since lower level Statecharts are aware of the current game-state through subsumption, it is sufficient for the *StrategicDecider* to simply give commands to lower levels, without having to deal with unnecessary details. The Statechart itself looks nearly identical to the high-level approximation of the Halo AI given in (Isla 2005). This means that our Statechart approach yields a high level strategy that is visually explicit, free of complication, and nicely conforms to the original designer's intuitive understanding of what it ought to look like.

Output Layers

At this layer the effect of our limited subsumption becomes clear. While the *StrategicDecider* could store enough information to decide on a specific tactical strategy, this would unnecessarily complicate its structure. Instead, tactical deciders directly receive relevant sensor and analyzer data so that their decisions are prepared in advance. This has the additional effect of making tactical deciders more modular, in that their logic is self-contained and readily comprehensible. To demonstrate this, one of our most complex Statecharts, the *CombatDecider*, is presented in Fig. 3. Events relating to equipped weapons and ammo level come directly from the *WeaponSensor*, while enemy location comes from the *EnemyAnalyzer*. Upon receiving an `ev_Engage` event from the Strategic Decider, the orthogonal region `activity_switch` enters the `engaging` state and permits activity, and immediately triggers an on-entry event in the main region. As new relevant inputs are received, the *CombatDecider* is free to revisit its own tactical decisions, so long as the activity switch remains on. This approach is only possible due to the orthogonal region—without it, the number of states would double

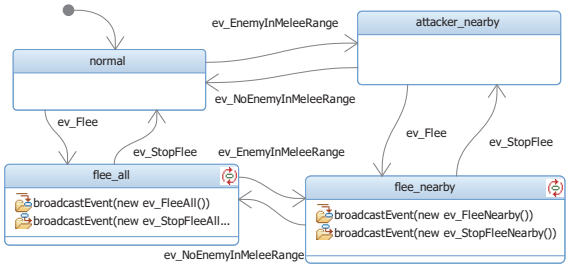

Figure 4: The *FleeDecider*.

as each state would have to have an active and non-active version.

A fourth pattern was isolated at the tactical decider level, the *Priority Decider*. These are used in the situation where a tactic A is the default activity, but always prioritizes tactic B when the triggering event is received. In Fig. 4, we see how *FleeAll* is the default behaviour, but is superseded by *FleeNearby* when an enemy is detected in melee range. Upon abatement of the threat, the decider returns to default.

Coordinators solve potential conflicts between actions and correct for changing conditions. Since subsumption does not result in action without prior permission from the *StrategicDecider*, there are no inter-layer conflicts to resolve. Instead, the only coordinator automatically transforms move actions into vehicle movements if the NPC is driving, or run movements otherwise, simplifying move events at higher levels. It also uses a movement cool down timer to prevent movement oscillation in corner cases.

While most actuators trivially receive events and execute actions, one new pattern emerged. Similar to exceptions in code, actions can fail for a variety of reasons, e.g., trying to pick up an item that has just been picked up by another player. When these failures are relevant to the AI, it is useful to have the actuator itself track the results of an event and produce appropriate feedback. We call actuators of this type *Feedback Actuators*. Upon creating a move action, the feedback *MoveActuator* show in Fig. 5 can receive a callback event after executing its `pathfind(target)` call and react accordingly. For instance, if a move fails because a new obstacle has appeared, a reasonable course of action is to retry and allow the pathfinder to calculate a new path around the new roadblock; usage of a feedback actuator allows this. On the other hand, if the failure is due to no path existing, higher levels may wish to change behaviours. This is signalled by having the sensing portion of the feedback actuator create an *ev_MoveFailed* event, notifying higher levels that a new destination should be determined, or a new behaviour chosen.

Key Features

In the various publications and presentations regarding the Halo AI, several key features were highlighted. These included a technique enabling efficient event reaction and a method to customize behaviours for individual NPCs. In the conversion to the layered Statechart formalism, it was important to ensure that none of this functionality was lost.

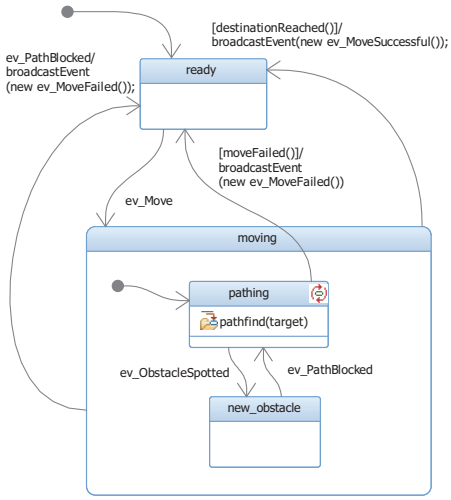

Figure 5: The feedback *MoveActuator*.

Stimulus Behaviours

The Halo AI attempted to enable reactive behaviours by the creation of *stimulus behaviours*. The framework used a stimulus system that received events from the game at-large, and reacted by inserting special stimulus nodes into the behaviour tree at run-time. Upon the next execution through the tree, the new stimulus behaviour would run as expected. The insertion point of a node was carefully chosen, so that it respected the decision making process of the tree without overriding higher level behaviours that may potentially supersede the inserted behaviour. While this provided the ability to react to rare events without repeatedly checking for a condition, it came at the price of making the behaviour tree less understandable by obscuring behaviours.

Fitting an event-based reaction into a Statechart-based approach is trivial. This can be done by adding a Statechart that reacts to the event in question and produces an output that triggers the appropriate higher level Statechart. The example of a stimulus behaviour in Halo was for the AI to flee if their leader was killed. Our Statechart approach accomplishes this by adding a new analyzer to check if the player involved in an `ev_PlayerKilled` event was actually their leader. If so, it reacts by sending a special `ev_ LowMorale` event to trigger fleeing behaviour at the *StrategicDecider*. The AI behaviour is clear at all times since dynamic behaviour modification is not required. As well, the new behaviour is modularized, with all operations related to leader tracking stored in one location, making it straightforward to comprehend.

Behaviour Masks

Halo employs a shared static structure for the behaviour tree which effectively limits memory usage, but this comes with a downside: using a single data structure prevents each character from having their own customized AI. While this can be tempered by clever design, for example by having characters that wield ranged weapons travel down a different

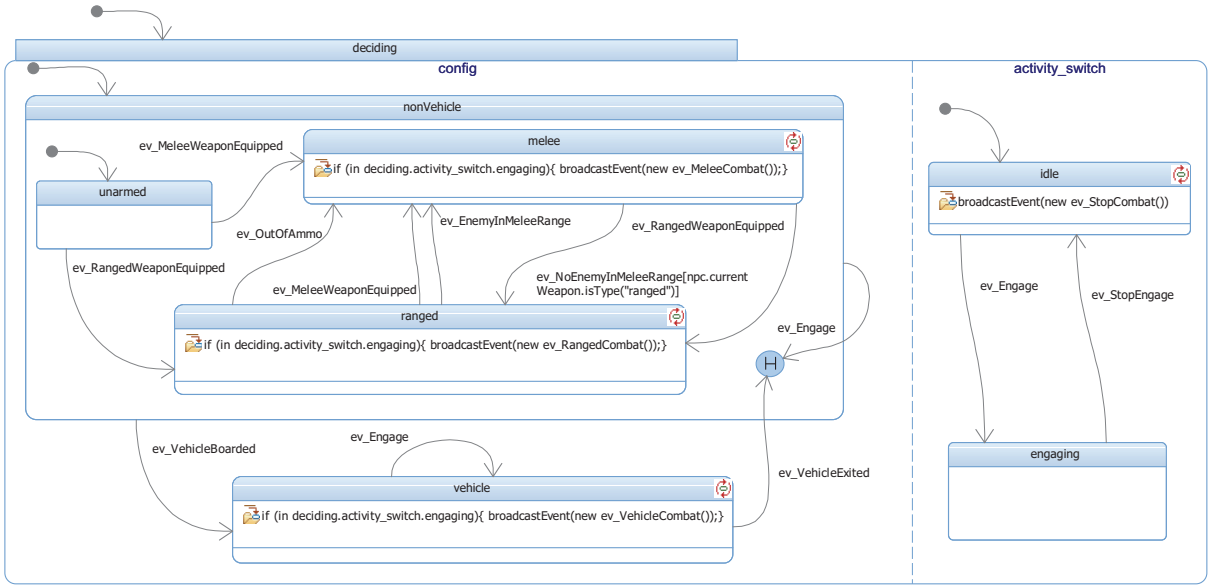

Figure 3: The *CombatDecider*.

branch of the behaviour tree than those with melee weapons, the problem remains that all AI characters use the same behaviour tree.

Halo 2 addressed this through *styles*. They provide customization for characters by providing a list of disallowed behaviours, effectively pruning branches from the behaviour tree. This evolved into *behaviour masks* in Halo 3, which gave designers the choice between 3 sets of disallowed behaviours, resulting in NPCs that were normal, aggressive, or timid. An aggressive behaviour mask, for example, disallows the branches of the behaviour tree concerned with fleeing and taking cover.

In our layered Statechart-based approach, the situation is not as simple as trimming branches, since decision making and reaction to events is distributed across modules. Regardless, if a module does not receive a triggering event, no behaviour will occur. Two approaches exist: filter events, or entirely ignore the generating Statechart. Implementation-wise, the Statechart approach can also employ a shared static structure, with the current state of each Statechart with respect to an NPC stored in the NPC. By setting this to 0, it can be communicated to the Statechart executor that NPC is not using the referenced Statechart, causing that Statechart to be skipped when processing events.

Ignoring entire Statecharts has the disadvantage of being relatively coarse-grained—a Statechart may produce more than one event, and so ignoring the Statechart may affect multiple downstream behaviours. Individual event filtering gives more fine-grained control, but suffers from larger storage requirements as well as additional computational costs in verifying individual events. Statechart behaviours that are fully filtered may also be a source of redundant computation. In our design we are able to exclusively make use of Statechart removal to customize individual behaviours, relying on use of small and relatively modular Statecharts to achieve sufficient granularity.

Validation

The Statechart model we have created is based on information made available through various presentations on the Halo AI. This approach was chosen to ensure our work could be used outside of any proprietary context, but has the disadvantage that the public description is incomplete, and so our efforts represents our best approximation of what we interpret the full AI to be. Moreover, our modelling work did not involve the creating or coding of various algorithms invoked in the AI, such as target selection, pathfinding, or determining cover, and this limits our ability to validate that our Statechart AI is a complete and faithful reproduction of the original behaviours.

Validation took place at two levels of abstraction. First, the model was analyzed to determine the effectives of the approach at the design level. Secondly, the AI was implemented to verify correctness of the logic in the Statecharts, as well as to learn how the model functions in practice.

The Model

Part of the motivation of using Statecharts was to reduce the complexity of the representation. This is born out by a statistical analysis of the resulting system. In total, there were 48 Statecharts, containing an average of 4.00 states and 6.08 transitions. Only 5 Statecharts had 10 or more states (the largest was the *ThreatAnalyzer* with 13), but all of these had orthogonal regions in the state at the top of the hierarchy. These regions can be separated into independent Statecharts, and if this was done, the averages drop to 3.28 states with 5.18 transitions. While the overall number of Statecharts is high for such a complex AI, each Statechart is small enough to be easily understood. The *StrategicDecider*, for instance, is only 8 states with 12 transitions, a comprehensible size.

The presented Statechart patterns proved useful. The quantizing sensor appeared twice, binary analyzers were employed three times, feedback actuators were used twice, event-mapping sensors were needed four times, and priority deciders were used twice. This accounts for 13 Statecharts, meaning that 28% of the Statecharts were pattern instances, varying only in state and event names.

Implementation

The AI was implemented in Mammoth (Kienzle et al. 2009), a highly extensible research framework for MMOs. While Mammoth lacks many features now common to AIs in FPS games, such as cover maps and navigation meshes, this implementation was sufficient to test core functionality of the AI, demonstrating that the various Statecharts operated correctly in both their individual and collective roles.

Our implementation also allowed for practical verification of potential performance concerns. For example, since the design is modular, event profligacy is a concern. Event generation was thus examined by looking at the number of events potentially generated in response to various inputs. The maximum number of events was 14, assuming that every guard evaluated to true and that every Statechart involved was in the appropriate state to react to an event thereby continuing the chain reaction. While this worst case is high, the number of events generated in practice was quite low. After several short executions of the AI, the AI generated a mean of only 0.3 events per execution pass, while the median number of events in a single pass was zero. Aside from the occasional burst of 5-10 events, event generation was quite limited and did not cause a significant overhead.

Conclusions and Future Work

In performing this research several interesting properties of the layered Statechart-based approach emerged. Of immediate utility are the Statechart patterns, which can easily be adopted into a variety of FSM approaches, or generated automatically through tool support. The value of modified subsumption in managing complexity was also clear. The size of the resulting Statecharts was small enough to be easily comprehended, due in large part to the simplification of high level Statecharts. The use of orthogonal states also proved to be a valuable tool in preventing combinatorial explosion of states, noteworthy because this technique is not a part of standard hierarchical finite state machines, and allows for the specialized activity switch.

In addition, this research demonstrates for the first time that a large-scale, complex game AI can be modelled using the layered Statechart-based formalism.

Future work will focus on analysis and testing of our constructed models, for which the existence of a non-trivial and realistic game AI is essential in order to demonstrate practical value, as well as for guiding the design of accompanying analysis and verification tools.

Acknowledgements

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

References

Brooks, R. 1986. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of* 2(1):14 – 23.

Champanard, A. 2012. Understanding the second-generation of behavior trees. http://aigamedev.com/insider/tutorial/second-generation-bt/.

Dill, K. 2011. GDC: Turing tantrums: AI developers rant. http://www.gdcvault.com/play/1014586/Turing-Tantrums-AI-Developers-Rant.

Dragert, C.; Kienzle, J.; and Verbrugge, C. 2011. Toward high-level reuse of statechart-based AI in computer games. In *Proceedings of the 1st International Workshop on Games and Software Engineering*, GAS '11, 25–28.

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.

Dyckhoff, M. 2007. Evolving Halo's behaviour tree AI. Presentation at the Game Developer's Conference.

Dyckhoff, M. 2008. Decision making and knowledge representation in Halo 3. Presentation at the Game Developers Conference.

Fu, D., and Houlette, R. T. 2002. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems* 17(4):81–84.

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

Heckel, F. W. P.; Youngblood, G. M.; and Ketkar, N. S. 2010. Representational complexity of reactive agents. In *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, 257–264.

Hecker, C. 2009. My liner notes for spore/spore behavior tree docs. http://chrishecker.com/My_liner_notes_for_spore/Spore_Behavior_Tree_Docs.

Isla, D. 2005. Handling complexity in the Halo 2 AI. Presentation at the Game Developers Conference.

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

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.

Kolhoff, P. 2008. Level up for finite state machines: An interpreter for statecharts. In Rabin, S., ed., *AI Game Programming Wisdom 4*. Charles River Media. 317–332.

Onuczko, C.; Cutumisu, M.; Szafron, D.; Schaeffer, J.; McNaughton, M.; Roy, T.; Waugh, K.; Carbonaro, M.; and Siegel, J. 2005. A Pattern Catalog For Computer Role Playing Games. In *Game-On-NA 2005*, 33 – 38. Eurosis.

Orkin, J. 2006. Three states and a plan: The AI of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC)*.

Schwab, B. 2008. Implementation walkthrough of a homegrown "abtract state machine" style system in a commercial sports game. In *in Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 145–148.

Wooldridge, M., and Jennings, N. 1995. Agent theories, architectures, and languages: A survey. In Wooldridge, M., and Jennings, N., eds., *Intelligent Agents*, volume 890 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 1–39.