2011

Feasibility Study of Utility-Directed Behaviour for Computer Game Agents

Colm Sloan
Technological University Dublin

John D. Kelleher
Technological University Dublin, john.d.kelleher@tudublin.ie

Brian Mac Namee
Technological University Dublin

Recommended Citation

Sloan, C., Kelleher, J. & Mac Namee, B. (2011) Feasibility Study of Utility-Directed Behaviour for Computer Game Agents. Proceedings of the 8th International Conference on Advances in Computer Entertainment, Lisbon, Portugal ,08-11, November. doi:10.1145/2071423.2071430

Feasibility Study of Utility-Directed Behaviour for Computer Game Agents

Colm Sloan
Applied Intelligence Research Centre
Dublin Institute of Technology
Ireland
colm.sloan@student.dit.ie

John D. Kelleher
Applied Intelligence Research Centre
Dublin Institute of Technology
Ireland
johnd.kelleher@dit.ie

Brian Mac Namee
Applied Intelligence Research Centre
Dublin Institute of Technology
Ireland
brian.macnamee@dit.ie

ABSTRACT

Utility-based control (UBC) hasn't been widely adopted for commercial game AI. Some of the reasons for this are that UBC is perceived to be: (1) resource intensive, (2) difficult to design complex behaviours with, and (3) difficult to scale for use in complex environments. This paper investigates these perceptions to see if UBC is suitable for controlling the behaviour of non-player characters in commercial games. The investigation compares agents using a UBC system against two control systems that are more frequently used in commercial games: finite state machines (FSMs), considered a simple control system, and goal-oriented action planning (GOAP), considered a complex control system. We present a case study which suggests that: (1) UBC is more resource intensive than FSMs and less than GOAP; (2) it was reasonably simple to create complex behaviours with UBC; (3) UBC didn't scale as well as FSMs or GOAP for use in complex environments.

1. INTRODUCTION

A *non-player character* (NPC) is a type of character in computer games that is controlled entirely through the use of a behavioural control system and without any continuous instruction from a human. A designer would ideally script all NPC behaviour to give exactly the type of gaming experience they want for the player. The scope of modern AAA computer games is so large that scripting all NPC can be impossible. A common approach to overcome this time limitation is to use a control system to carry out the bulk of NPC behaviours and to script only behaviours particular to certain situations.

A *behaviour control system* directs the actions performed by a NPC in a computer game. Some of the behaviour control systems that can be used to direct the actions of NPCs in computer games are rule-based systems [15], expert systems [12], decision trees [29], Markov systems [30], fuzzy-logic based systems [19, 22], hierarchical FSMs [7, 14] and behaviour trees [8].

Many designers of modern computer games spend little time developing sophisticated control systems. Some AAA title games being produced today still use NPCs that are predictable and perform actions that are not appropriate for the particular situation of the NPC. This behaviour could be the result of limited development time that designers have to create a behaviour control system or because the designers don't want to use much computational resources on NPCs. In this paper, we describe an implementation of utility-based control (UBC) that is used in an attempt to create NPCs that are less predictable and more capable of selecting actions than NPCs that use popular behaviour control systems. We attempt to do this without using significant computational resources or a prohibitively long implementation time.

We will compare UBC to finite state machines (FSMs) [11] and goal-oriented action planning (GOAP) [26]. These two control systems have been selected for comparison because the FSM is considered among the most basic and resource inexpensive of the behaviour control systems, GOAP is considered among the most complex and resource intensive and both are used in commercial computer games. Comparing against these two control systems should allow us to see where a UBC system falls in the spectrum of complexity and resource requirements.

Contribution: The contribution of this paper is a detailed evaluation and comparison of three control systems (FSMs, GOAP and UBC) to judge how suitable a UBC system is to drive the behaviour of NPCs in computer games. The UBC system will be compared to these two systems by: (1) quantitatively testing the resource requirements of all three control systems in terms of processing and memory; (2) an evaluation of the complexity of the behaviours afforded by all three control systems where emergent behaviours are deemed to be more complex, interesting and desirable; and (3) our personal assessment of how easily each of the three control systems are implemented and extended. The experiments described in this paper were carried out in a virtual hospital environment.

Overview: Section 2 will briefly describe the three control systems used in our experiments. Section 3 will describe the implementations used to carry out the comparison of the control systems. Section 4 will detail the results of the evaluation. Section 5 concludes and explains the directions in which we intend to take this work in future.

2. BACKGROUND

This section will discuss each of the three control systems used as part of this research: (1) FSMs, (2) GOAP and (3) UBC. It will briefly discuss each system, how it works, an example of it working and highlight some of its strengths and weaknesses.

When the word *state* is used hereafter in this document, it will refer to the values of a set of variables when referred to with GOAP, and a behaviour that a non-player character is executing when referring to an FSM.

Presented as Full Paper at ACE'2011 - Lisbon, Portugal

2.1 Finite State Machines

Adopted from academia [24, 25], an FSM in a computer game is a control system consisting of a predefined collection of states, each defining exactly how an agent should act when in that state [4]. An agent may only occupy one state at a time [1]. FSMs have been known to require few resources and to be implemented quickly. Many variations of the FSM have been developed by professionals in the games industry and by academics [9, 13, 32, 35].

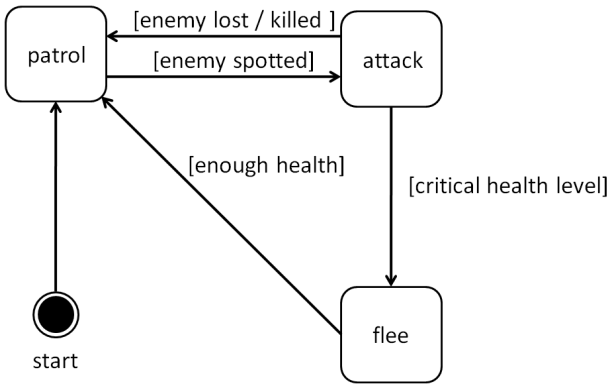

Figure 1: An example of an FSM

An example of an FSM for a typical guard NPC is shown in Figure 1. The rounded rectangles represent states, the arrows represent transitions between states and the words surrounded by square brackets represent the event that triggers a transition.

When a designer is giving a behaviour to an NPC using an FSM, the designer must specify every state and how to behave in that state before the program starts running. A transition to a state can be triggered by the satisfaction of some set of conditions. The NPC may then carry out some premeditated fixed sequence of actions determined by what state he is in, giving the impression of an intelligent behaviour, such as running away from an enemy when health is below a certain level.

FSMs are flexible and can implemented reasonably quickly when there are few states. Problems arise with FSMs as more states are added to them. All transitions between states need to be specified within the code while the FSM is being designed. This means that the designer must know all possible transitions from any state to any other at design-time. It can be very difficult to foresee all possible situations an NPC may find itself in when dealing with complex environments such as those in a commercial game. Designers may also find themselves tangled in the complex web of transitions between NPC states even if they can foresee all transitions. Furthermore, the low-level behaviour that must be specified when using an FSM is often programmed to work for a particular environment. This makes it difficult for FSMs to be transferred into a new environment, such as a new game. FSMs have a range of other drawbacks that are well recognised [6] including the difficulty in adapting them to cope with goal-directed decisions; the fact that they are not standardized and that they don't handle concurrent behaviours well.

2.2 Goal-Oriented Action Planning

GOAP [26] is a goal-oriented planning system that has been used to control game agents in commercial games [28]. GOAP goes a step further than other behaviour control systems by allowing agents to decide at run-time not just what to do but also how it should be done. Agents using GOAP periodically reevaluate their situation and choose the optimal behaviour to achieve their most relevant goal. A goal in GOAP is any set of conditions that an agent wants to satisfy, be it to get a particular item or to kill a particular opponent.

GOAP is based on an early STanford Research Institute Problem Solver (STRIPS) [10]. Actions in STRIPS have two parts: (1) the conditions that must be satisfied in order for the action to be executed, and (2) the effects that occur when an action is executed. Unlike STRIPS, GOAP also associates actions with costs, giving the planner the ability to know which actions are preferable. It also has the ability to replan, i.e. the ability to make a new plan when a previous one has failed during execution. Replanning uses knowledge of which actions have previously failed and creates a new plan to achieve the same goal that doesn't include any of the actions that have previously failed.

GOAP agents have predefined goals. Some goal selection mechanism is used to pick a goal the agent wishes to achieve. This goal is passed to a plan formulation process to determine the lowest cost sequence of actions that will achieve the goal. GOAP uses a regressive A* search [23] from the goal state to the current state. It finds the lowest cost sequence of actions by searching through a library of possible actions using action cost to guide the search. The search begins by searching for an action with the effects that satisfy the goal state and then adds new actions to the plan that are needed to satisfy any of the conditions of that action. More actions are added to satisfy the conditions of actions that already exist in the plan until all conditions are satisfied.

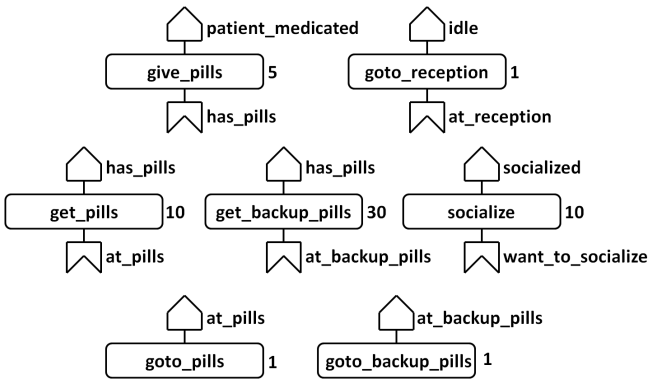

Figure 2: A representation of the GOAP plan formulation process

Figure 2 shows a representation of the GOAP plan formulation process for a nurse in a simulation, where actions are shown as rounded rectangles, action costs as the numbers next to the rounded rectangles, effects as arrowheads and conditions as inverted arrow heads. For example, the nurse has the goal of medicating a patient. This goal consists of the **patient_medicated** condition. The planner will search through the action library for an action that has the effect of changing **patient_medicated** into the desired state, which leads it to the **give_pills** action. The planner will look through all actions in search of those with effects that satisfy the **has_pills** condition. The planner will find the **get_pills** and **get_backup_pills** actions. The **get_pills** action is added to the plan as that action has a cost of 10 and is lower than the action cost of 30 for the **get_backup_pills** action. The **get_pills** action has the **at_pills** condition. The **goto_pills** action satisfies this effect and is added to the plan. The planning process then ends because there are no conditions left unsatisfied and the lowest cost plan consisting of the actions **goto_pills**, **get_pills** and **give_pills** is selected for execu-

tion.

GOAP is essentially an optimal, heuristically guided planner, like those introduced during the mid 1990s [3]. One difference between such a planner and GOAP is that GOAP agents refer to a list of previously failed actions while formulating a plan, which itself has been done before in early planning research [31]. GOAP differs from other planners because GOAP specifies not just what action should be executed but also how it is executed for the particular agent executing it. This allows actions to be performed differently for particular agents in a game. For example, an agent using the **goto** action could result in it performing the **fly** action if it is a bird, but the **walk** action if it is an elephant. GOAP combines all of these ideas in a manner tailored to computer games.

One of the strengths of GOAP is that it uses the A* search algorithm which is already well known in the games industry thus reducing the learning curve for the algorithm and allowing it to leverage optimizations made to A* for pathfinding. Another is the ease with which it can reuse planning logic and customise actions for specific agent types because of its decoupling of what actions should be performed and how they should be performed. GOAP has a number of weaknesses. Very specific behaviours are difficult to craft with GOAP as designers can only work with high-level logic. This limitation can be overcome by using some scripted behaviours in parallel with GOAP but there's nothing within GOAP that deals with this problem. Another problem with GOAP is that because it starts with the goal and works backwards, there is no way to begin executing the first action in a plan while the other actions are still being calculated [26], a problem that doesn't occur with progressive search planners [16]. GOAP also only allows for the pursuit of one goal at a time and is difficult to extend for use with partially-satisfying plans, though it is possible [5]. Perhaps the biggest complaint game designers normally have with GOAP is that it requires so many symbols, such as **has_pills**, when it scales up to the level of a commercial game. Finally, GOAP finds the lowest cost plans but these may not necessarily be the best overall plan as the planner fails to account for how beneficial an action may be to the planning agent. Unlike GOAP, the complete effect of an action can be modelled in a UBC system.

2.3 Utility-Directed Behaviour

Utility is a measure of the desirability of a state to an agent with a set of goals [33]. This desirability is represented by a utility function that maps a state to a real number. This mapping makes it possible to judge exactly how far a state is from the state preferred by an agent with goals. This mapping makes it possible to judge how an action will affect the utility of a state to an agent with a set of goals if that action was to be executed. An agent whose behaviour is driven by utility will select actions that will create states with the highest utility for the agent with a set of goals. Utility is useful for selecting actions in two types of situations: (1) when there are conflicting goals e.g. a taxi driver wants to drive as quickly as he can but safely, and (2) when there is uncertainty and weights involved in achieving multiple goals e.g. a student really wants to publish a paper soon but also wants to publish at a high-impact conference. Utility can be used to find the best action to take if weights are associated with the goals and probabilities are associated with actions denoting the chance an action will succeed.

The calculation of utility can be done a number of ways. A Markov decision process [2] is one popular type of control system that can use utility. Markov decision processes will require knowledge of reward and discount functions and will need to be trained. Training can be unsuitable for computer games because games change rapidly during development, requiring retraining of Markov process and also because it can take time to tinker with the reward and discount functions to get the desired behaviour. More simple approaches exist such as simply having a one utility function for each goal that measures the satisfaction of that goal and having another function to tally the values returned from each of these functions. Though this simple utility-directed method can find the best action to perform while working with multiple goals, it would need functions and values to calculate the utility of a state for each goal.

For example, let's imagine that there is a man who wishes to buy a watch. His first goal is to have as much money as he can where €100 is the maximum and his second goal is to be stylish, where 100 style points is the maximum. He has €100 in cash and a style level of 0. There are two watches that he's interested in buying. The first costs €100 and raises his level of style by 95. The second watch costs €50 and raises his level of style by 60. Assume there is a fictional unit of measurement called *utils* which is used to describe the utility of an action. We'll say that for every euro gained is worth 1 util and that style point gained is worth 2 utils. The utility function for each goal of the man will be used to check the utility of the state with regard to those goals if each action was executed. The action of buying the first watch will bring a change of -100 utils to the goal of having money and +190 utils for the goal of being stylish, totalling 90 utils. The action of buying the second watch will bring a change of -50 utils to the goal of having money and +120 utils for the goal of being stylish, totalling 70 utils. Therefore the action that creates the highest utility state for the man is to buy the first watch. The method described in this example will be similar to the method implemented for use in the experiments.

3. IMPLEMENTATION

A test simulation environment has been developed to compare the performance of FSMs, GOAP and UBC systems at driving the behaviours of NPCs in game environments. The gameplay in the simulation takes place within a virtual hospital where NPCs play the roles of nurses, doctors, patients, and visitors. Figure 3 shows a screenshot of this environment.

Figure 3: A screenshot of the hospital simulation

Nurses care for patients by giving them food and medicine when necessary. Nurses also socialize with nurses and doctors and perform administrative duties. Doctors occasionally check how things are going. Patients ask for food, medicine and company. Visitors chat with patients and rest when they're finished chatting. The actions executed by the NPCs are governed by either an FSM, GOAP, or utility-based controller.

Each control system was designed to be as simple as it could be. This is to try to make each as domain-independent as possible.

Each FSM implemented in the simulation has at least two states. An agent controlled by an FSM starts off in a state that has been selected as the starting state. In each state the control system checks if certain conditions have been met that would trigger the agent to move into another state. For example, a nurse performing administrative duties would switch into a state of giving medication if the medication level of any patient fall below a certain threshold.

The particular implementation of GOAP used during this evaluation uses GOAP in its simplest form without any enhancements e.g. no caching. The planning agent has a set of goals where each of which is associated with a priority. The planner will check every 0.2 seconds to see if a new plan should be made. A new plan is made if the planning agent has no plan, has discovered that his current plan has failed, or if a higher priority goal is no longer satisfied. The planner will create a plan for the highest priority goal found. The list of goals an NPC has are specified at design-time. The list of actions an NPC can execute are specified at design-time and are used to form plans at run-time.

Each agent using our UBC system has a set of goals. The UBC system can only calculate the utility of a state that is one action away for the game agent. Unlike GOAP, which dynamically builds plans by chaining actions together, our particular UBC system encodes a sequence of actions as one predefined plan, similar to a hierarchical task network [34]. For example, the nurse can attempt to improve the state of to her goal of having all patients medicated by checking how the utility of the state would be if she executed one of her predefined plans. She would then select the plan that changes the state in the best manner for the goal of medicating the patients. An example of a predefined plan is shown in Figure 4, where rectangles represent goals, rounded rectangles represent predefined plans and where arrows show which goal may be affected by a predefined plan. For example, for the goal keeping patients medicated is affected by the predefined plan that consists of the actions of going to the medicine cabinet, retrieving the medicine, going to the patient and giving the medicine to the patient.

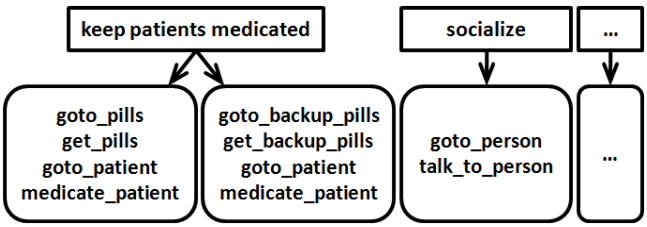

Figure 4: A representation of some goals and predefined plans used in the utility-based control systems

Each goal is associated with a weight that is used to let the NPC know which goal is more important. Each predefined-plan considers the state created when that action is performed on the best target for that action. For example, only the hungriest patient will be considered when calculating the action of the nurse feeding the patients. All utility-directed NPCs will attempt to maximize the utility of the state to their goals.

While this hospital environment isn't quite as demanding as some found in commercial computer games it is sufficient to compare the performance of the three control systems. The environment was created as a modification of the game *Half-Life 2* (www.half-life2.com). This was selected because it was necessary to run the simulation in a game engine to ensure that all control systems tested work within the computational and memory restraints placed on a commercial game.

4. EVALUATION

In order to evaluate the feasibility of UBC as an NPC behaviour control system for games we compared its performance versus FSM and GOAP in terms of: (1) the processing and memory requirements of the three control structures, (2) the complexity of the behaviours they afford, and (3) our assessment of how easily each of these control systems are implemented and extended. Comparing the FSM, GOAP and UBC approaches to control in games based on their processing and memory requirements is justified because these metrics are widely accepted as fundamentally important for real-time computational systems. Our motivation for assessing the complexity of the behaviours these systems can generate is an attempt to capture the potential of each system to generate more diverse gameplay. We think that this assessment of the control structures is relevant from the perspective of a game player because the more complex behaviours, the greater the potential for replayability and hence the more desirable the system. Contrasting with this perspective of the player, our analysis of how easily each of the control systems are to implement and extend can be seen as taking the perspective of the designer on the desirability of the system as a tool.

NPCs were made to perform a single thinking process with their control system every 0.2 seconds. The measurement concerns only how long it took their control system to finish performing whatever it needed to do to cause a behaviour in an agent. The sum of all processing time for each control system was recorded. The simulation was run five times for each of the control systems. The runs with lowest and highest average time for each type were discarded and the remaining three runs of each type were averaged over the produce a single line (shown in Figure 5) representing the performance of each type of control system. NPCs were programmed to perform the same tasks but using each of the three different control systems for each simulation run. Each simulation lasted four minutes as this was the amount of time needed for agents to have completed all of their actions at least once and return to a state where nothing needed to be done momentarily.

The computer used for all of the tests described had an Intel Core 2 CPU 6600 at 2.4Ghz, 4096MB of RAM and ran Windows 7 Ultimate 64 bit.

4.1 Processing

Figure 5 shows the processor usage for UBC agents, GOAP agents, and agents using FSMs. Because of the large number of game cycles recorded, the average of every ten processing cycle time values were taken and averaged over to create one value. The average of this value for each of the simulation runs of a particular control system represents one data point in the line representing processor usage over time for that particular control system. The very beginning of the recorded processor usage in the simulation was removed from the figure because of a massive spike in computation that occurred at the beginning of the simulation that dwarfed the other data points, making it very difficult to distinguish the processor usage of any of the behaviour control system throughout the simulation. This spike occurred because the Half-Life 2 simulation is creating the level and characters. It was removed because it was not relevant to the evaluation of the behaviour control systems. The reason for the increase in processor usage over time for each of the behaviour control systems is unknown at this time but it appears to affect each system equally.

Each control system was being used by 32 agents. The FSMs used the least processing time and although GOAP used the most processing time because of its A* search, the amount of processing wasn't prohibitive. Moreover, GOAP didn't have any visible ill

effects on the simulation except at the beginning of GOAP simulations, where there was notable lag that vanished after a few seconds. Given that GOAP uses more processing power than the UBC system and that GOAP is used in commercial games, this strongly suggests that the UBC system is a computationally viable behaviour control system for NPCs in commercial games.

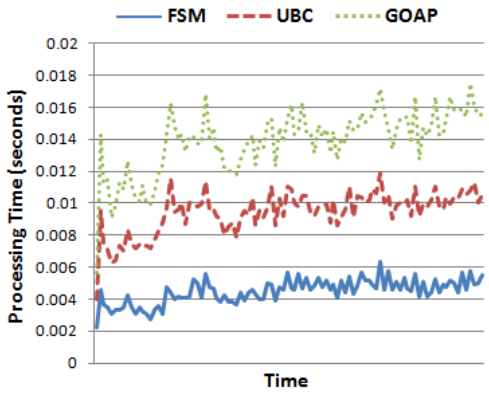

Figure 5: A comparison of processor usage for the FSMs, GOAP and the UBC behaviour control systems when running in a simulation of 32 NPCs

4.2 Memory

The memory requirements for the FSM and UBC were negligibly small. For GOAP, a database of all goals, actions, and conditions and effects used in the simulation that GOAP uses to perform search must be held in memory. Each NPC in the simulation only had a small set of actions to choose from. The data that needs to be stored for each action is quite small. Commercial games using GOAP have only needed a few dozen actions, so the memory footprint of the databases have been very small, as they were with this simulation and others [20]. The tiny memory footprint for the UBC system shows that the system is a viable behaviour control system for NPCs in commercial games in terms of memory usage.

4.3 Behaviour

Following Juul's [17] analysis of online games we distinguish between open and closed games. Open games include emergent elements that introduce variability into the gameplay. Importantly, Juul notes that emergent games "tend to be replayable", an aspect that is desirable from a players perspective.

There were no emergent behaviours or surprises of any kind observed when the simulation ran with agents controlled using GOAP. This may be because the small scope of this simulation simply left no room for anything unexpected to occur. The FSM also acted exactly as it was scripted to do, as expected. The UBC, on the other hand, proved more dynamic and interesting. During the trials the FSM and GOAP controlled nurse NPCs consistently focused on first giving everyone their medication and then giving them food as these were respectively the two most important duties of the nurse NPCs. In contrast, the UBC controlled nurse NPCs judged whether satisfying the hunger of some patient was better to the overall utility of the nurse and would give food first to a really hungry patient rather than give medication to a patient that isn't in great need of it. This variability in behaviour wasn't possible to implement in GOAP as GOAP only considered the cost of plans rather than the benefit derived from executing a plan. This behaviour would have been possible to implement in an FSM but it would have to be done by first calculating the point at which, with the consideration of the importance of each goal, it is more beneficial to feed a hungry patient than it is to medicate an patient in need. Having this calculation automated and resulting in interesting and unexpected action selection is a real advantage of using utility-based approaches.

4.4 Ease of Use

The complexity of the FSMs grew quite quickly. Each new behaviour added to a growing collection of variable checks. The simulation only required a few behaviours for each NPC, but even in this case the comprehensibility of the systems plummeted quickly. This has also been experienced in other research [21]. Because the order in which things occurred mattered in the FSM, great care had to be taken not to disturb the flow of the sequence of checks.

On the other hand, GOAP proved remarkably modular. Actions, goals etc. could be broken down into very comprehensible chunks. The things that motivate behaviour are separated from plans, that are themselves separated from how a plan may be carried out. There is no point at which a designer may feel overwhelmed by the complexity of the behaviour. As a result of this modularity, there is no code replication in planning. The modularity also meant that new actions could be added with little thought of consequences. It was difficult to understand how the whole system integrated its parts together at first but once this was established, productivity was greatly enhanced. Though, by the time the use of the system was mastered, most of the behaviours had already been implemented, owing to the very few NPC types in the simulation. However, once GOAP was set up, it was very easy to add behaviours to agents.

It was more difficult than suspected to implement utility functions for goals to make it so certain goals were more important than other to some degree e.g. making sure the patients having their medication was more important than making sure they were properly fed. When a new behaviour was added to an agent using a UBC, the utility functions had to be tweaked again to get the desired behaviour.

The code used to create behaviours for the FSM and the UBC system were very specific to their respective NPC and used domain-specific knowledge that and could not use much code previously made for other NPCs. GOAP was different in that it was able to share behaviours across agents. Visitors and nurses were given the low priority goal of socializing. Both used exactly the same planning logic but had different implementations of how they went about fulfilling that goal. Nurses would talk to doctors or other nurses, while visitors would talk to patients. There was no logic duplication for planning. Actions, such as talking, were also shared. GOAP is also flexible enough to allow goals and actions to be shared over different projects [27] because the logic may be domain-independent while the implementation of actions can be domain-specific.

5. CONCLUSIONS AND FUTURE WORK

The goal of this paper was to evaluate the feasibility of using a UBC system to drive the behaviours of NPCs in computer games. The evaluation compared the UBC system to FSMs and GOAP. The comparison considered processing and memory resource requirements of each control system, the variability of the behaviours they can dynamically generate, and our assessment of how easily each of these control structures are to implement and extend.

Our findings were that: (1a) The UBC systems needed more processing time than the FSMs and less than GOAP; (1b) The UBC system required roughly the same amount of memory as the FSM and less than GOAP; (2) NPCs using the UBC systems demon-

strated emergent behaviour whereas the other systems did not; (3) The UBC system was harder to author than the FSMs and scaled the worst out of the three control systems. In our opinion, this last point is probably the reason why UBC systems have, to the best of our knowledge, never been used to control NPC behaviour in AAA games.

Nevertheless, the UBC system showed the greatest potential to generate emergent behaviours, opening the possibility for more challenging and interesting games. In future, we plan to combine UBC with GOAP to create a system that has the sensible processing and memory requirements and the ability to plan for multiple goals, while also maintaining the simple authorship abilities of GOAP.

6. REFERENCES

[1] D. Abreu and A. Rubinstein. The structure of nash equilibrium in repeated games with finite automata. *Econometrica: Journal of the Econometric Society*, pages 1259–1281, 1988.

[2] D. Bertsekas. *Dynamic programming: deterministic and stochastic models*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987.

[3] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 714–719. JOHN WILEY & SONS LTD, 1997.

[4] D. M. Bourg and G. Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004.

[5] D. H. Cerpa and J. Obelleiro. *An Advanced Motivation-Driven Planning Architecture*, volume 4, pages 377 – 381.

[6] A. J. Champandard. 10 reasons the age of finite state machines is over, Dec. 2007.

[7] A. J. Champandard. The gist of hierarchical fsm, Sept. 2007.

[8] A. J. Champandard. Understanding behavior trees, Sept. 2007.

[9] E. Dybsand. A generic fuzzy state machine in c++. *Game Programming Gems*, 2:337–Ű341.

[10] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[11] D. Fu and R. Houlette. The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom*, 2, 2003.

[12] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co. Pacific Grove, CA, USA, 1989.

[13] E. Gordon. *A Goal-Based, Multitasking Agent Architecture*, pages 265–274.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[15] F. Hayes-Roth. Rule-Based Systems. 1985.

[16] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001.

[17] J. Juul. The Open and the Closed: Games of Emergence and Games of Progression. In *Computer Game and Digital Cultures Conference Proceedings*, pages 323–329, 2002.

[18] J. Laird and M. VanLent. Human-level AI's killer application: Interactive computer games. *AI magazine*, 22(2):15, 2001.

[19] C. Lee. Fuzzy logic in control systems: Fuzzy logic controller–part I. *IEEE Transactions on systems, man, and cybernetics*, 20(2):404–418, 1990.

[20] E. Long. Enhanced NPC behaviour using goal oriented action planning. Citeseer, 2007.

[21] B. Mac Namee. Proactive Persistent Agents-Using Situational Intelligence to Create Support Characters in Character-Centric Computer Games. 2004.

[22] B. MacNamee and P. Cunningham. Creating socially interactive non-player characters: The μ-sic system. *Int. J. Intell. Games & Simulation*, 2(1):28–35, 2003.

[23] J. MatthewsŮGenerations. Basic A* Pathfinding Made Simple. *AI Game Programming Wisdom*, page 105, 2002.

[24] G. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[25] E. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[26] J. Orkin. Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom*, 2:217–228, 2003.

[27] J. Orkin. Symbolic representation of game world state: Toward real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, 2004.

[28] J. Orkin. Three States and a Plan: The AI of F.E.A.R. In *Proceedings of the 2006 Game Developers Conference*, volume 12, pages 13–14. Citeseer, 2006.

[29] J. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[30] L. Rabiner and B. Juang. Introduction to hidden Markov models. *IEEE ASSP MAG.*, 3(1):4–16, 1986.

[31] G. Reece and A. Tate. *Synthesizing Protection Monitors from Causal Structure*. Artificial Intelligence Applications Institute, University of Edinburgh, 1994.

[32] G. Rosado. Implementing a Data-Driven Finite-State machine. *AI Game Programming Wisdom 2*, pages 307–Ű318, 2004.

[33] S. Russell, P. Norvig, J. Canny, J. Malik, and D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, NJ, 1995.

[34] A. Tate. Generating project networks. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pages 888–893. Morgan Kaufmann Publishers Inc., 1977.

[35] P. Tozour. Stack-Based finite state machines. *AI Game Programming Wisdom 2*, pages 303–Ű306, 2004.