A Comparative Analysis of Intelligent Agents and State Machines: Models for the game Domain.

Student: Arran Bartish
Supervisor: Charles Thevathayan
Second supervisor: Peter Bertok

November 30, 2001

Declaration

I here by declare that the research following is entirely the work of Arran G. Bartish, under the supervision of Charles Thevathayan and Peter Bertok and that all sources and references have been duly and fully acknowledged.

Arran G. Bartish
Signed November 30, 2001

Contents

List of Figures

List of Tables

Abstract

The Study of game related topics has long been the subject of research and development, however game Artificial Intelligence (AI) still uses classical AI simulation techniques. The AI community continues to develop concepts regarding various types of agents, genetic algorithms, and others, while older computational models such as state machines continue to be used as the back bone for game AI development. In this research, we have attempted to shed some light on how the choice of implementation may affect the important factors such as performance, complexity and estimated-effort, as the games scale up. We have used performance measures, cognitive activities of designers and software metrics to arrive at our conclusions. Our findings show that agents and finite state machines have their relative merits. We have shown that, complexity measured as a function of the number of behaviours was linear for agents and quadratic for FSM. Though run-time performance is comparable for a small number of entities, it degrades faster for agents.

1 Introduction

Over the past years games have experienced many improvements, unfortunately many of these have not been in game AI. This has lead to problems where released games are spectacular in either 3D graphics, or in user interface, but Artificial Intelligence seems neglected. This is the part that makes a game fun and challenging to play. Despite much academic research being devoted to AI, the games industry continues to use rudimentary computer science technology such as State machines to create the desired behaviour for game entities. Recently the mould has begun to crack as a number of game titles are changing tack and focusing upon AI rather than graphics and interface. This could be a sign of things to come especially given the popularity and longevity these titles have enjoyed since release.

Given this seemingly changing environment in the games industry, exotic models for AI are being explored, but few ever get incorporated into games. This is partly due to the industry's general comfort with existing models, and the ever increasing urgency to release games in a shorter space of time. To our knowledge there has not been a formal comparison of current game AI technology such as State Machines, and what is considered exotic forms of game AI like the Belief-Desire-Intention (BDI) Agent Architecture.

A formal definition of the pros and cons of both current game technology and BDI agents, will be one small step closer to providing a higher quality of entertainment. Few industry's in information technology can boast such a broad range of core computer science fields. Games provide unique environments to test new and exotic technologies that otherwise could not be thoroughly tested in real world environments.

2 Statement of the Problem

The new games appearing in the market are using more human like characters and are incorporating forms of cognitive learning. Some attempts have been made to incorporate some of the newer technologies such as BDI in game development. However no thorough work has been done about the merits of these newer technologies.

The objective of this research is to find which model is more appropriate for game AI, and under which circumstances one model might be chosen over another. Due to the frustration of the industry and of users with what seems to be behavioural simplicity of Game AI, we must define the advantages and disadvantages of both traditional state machine based and a BDI agent alternative game AI. These choices will be determined through performance, iterative modification, and software engineering criteria. The reason being that each of these issues is of key importance to game development, a major software development undertaking.

3 Background of the Problem

Intelligent Agents have been the subject of many papers in the past. For further reading on the subject of agents, please see the following [1][3][7][9][10][11][12][14][19][20][23][24][25][32][33][34]. This paper will also lead into future topics of research that could be explored in game AI. We refer the reader to the following papers on agent teamwork and negotiation [4][5][6][26][27][28].

3.1 JACK

JACK is an agent programming language developed by an Australian based company called Agent Oriented Systems. It is an extension to the JAVA API, which allows the use of several other base classes. These base classes include Plan, Event, Database as well as some others. JACK allows the simple and easy definition of simple intelligent agents and BDI agent, with very little trouble for the programmer. JACK also allows the programmer to use the normal JAVA API, a huge advantage considering the growing popularity of the language.

JACK is the base we used to create our Bomberman agent. The reason for this was two fold. The first and most obvious reason is the availability of JACK and the support from Agent Oriented Systems. The second being its relationship with JAVA to allow quick platform independent development. In addition to this, we could develop an agent model and state machine model that were both JAVA based. It would not be a justifiable comparison if the two models were based on separate languages. Using JACK allowed us to use inheritance and other object oriented techniques that gave us the flexibility to swap one model for the other as required.

In order for us to do a comparison on state machines and agents, we need to describe how a JACK agent chooses to execute plans. JACK seems to have taken inspiration from event driven programming, as JACK agent plans are dependent upon event-like goals. Our JACK agent uses rule-based evaluation (similar to the state machine) to decide which event should be fired. When an event is fired the relevant plans are initiated and can be executed. So in this way events are much like goals in that when an event is fired, plans are executed to achieve the goal represented by the fired event. JACK allows a lot of versatility in this matter, and plans can evaluate their own suitability before they execute. Many JACK features such as plan self-evaluation and teamwork, were left out to allow for a *fairer* comparison between state machine and agents.

Unfortunately JACK also has a number of problems. Some of these problems were cited by John Thangarajah [29]. He highlights a problem with JACK in its inability to represent goals in a traditional sense, a fundamental concept for BDI agents. He continues to describe his proposed addition to JACK to allow this feature. It must be noted that while this addition could be made in future research, we have used unmodified JACK as this would suffice for the purposes of this research. The version we used was JACK 3.2 released in the middle of 2001. For more information on JACK please see user guides [17, 16]. For more information on JACK and teamwork see [18].

3.2 Games

Games programming involves many core areas of computer science. Increasing the computational requirements of games pose many problems. Some of the problems being tackled range from graphics clusters, algorithms for polygon limiting, artificial intelligence, and NP complete problems like "travelling salesman". This makes the game industry a fine test bed for innovative ideas and concepts at the leading edge of current technology. There has always been constraints forced on game AI, brought about through the need for graphics and IO processing, however these are becoming less of a problem as hardware for these purposes improves.

3.2.1 Finite State Machines

State machines have long been the most popular implementation of game entity behaviour [31]. State machines are easy to understand, and are a concept most computer science graduates have been exposed to. Students commonly use state machines for the parsing of grammars. What most of these students don't understand is that a state machine can be a very powerful implementation to simulate intelligent behaviour in games. They have been the traditional implementation for game AI for many years, and it appears they may be for many more. The basic characteristics of a state machine are:

- That it has a fixed amount of memory.
- A State machine is driven by input.
- It undertakes transitions between states.
- It produces simple output.

The concept of a finite state machine has long been developed and improved over the past decades to include variations such as fuzzy state machines and fuzzy logic. However these implementations will not be the focus of this research, and a traditional finite state machine will be used during the experiments. For an introduction to finite state automata or if the reader desires more information about the topic of computation, we refer you to [15] for an in depth introduction and description of finite state automata. State machines can be used to process complex and dynamic game environments. However they are prone to software engineering issues [30, 13], such as duplicate state decision points, that threaten to break even well designed State Machines.

State machines are best described as deterministic by nature, and implementing them can lead to software engineering problems as mentioned above. There are three prominent ways to implement a state machine [13], and the choice as to which implementation was to be compared in this research was of utmost importance. The first is using a block of if, or switch statements which are executed on some condition or state. This method could easily lead to duplicate state decision points, a common flaw in this method. A duplicate state decision point can occur through poorly documented code, and results in conflicting transition conditions. In addition they blur the concept of state and transition where a transition or condition statement might contain the implementation of the state. Another implementation is a decision table, indeed a number of games have used this implementation. It is fast, however it restricts versatility and this is far from satisfactory given the dynamic environment of games. The third choice (and the one used for this comparison) is an Object State Machine, where a state and its implementation and transitions are encapsulated by an object. State transitions in the object state machine are represented by an engine that keeps track of the current state. This model has a slightly higher run-time cost, but it's conceptual advantages out weigh this cost.

3.2.2 Current Game AI Technology and Trends

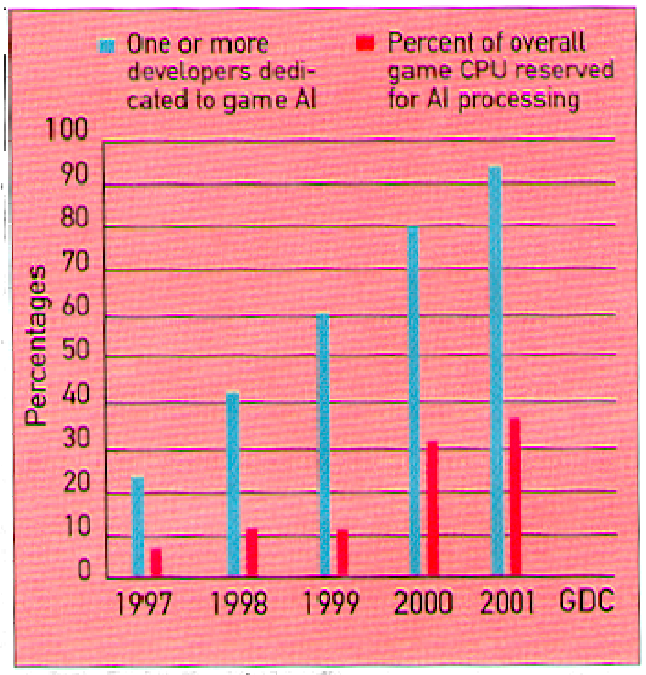


Figure 1: Current Industry trends with regard to AI resource allocation. Chart Extracted from [31]

Steven Woodcock sites at the 2001 Game Developers Conference that the majority of games being released still use a State Machine implementation [31]. This is interesting however, as the industry as a whole seems to be devoting more resources than ever before to game AI. In the same article, Steven Woodcock describes how the number of programmers per project devoted to AI has grown from 1 just 5-6 years ago, to a dedicated team of developers per project (see figure 1). For convenience figure 1 has been provided. He also discusses how greater amounts of CPU time are also being allocated to AI processing. This could be partly related to the fact that a lot of graphics processing that used to be undertaken by the CPU, is now being delegated to dedicated graphics accelerators. This would have the effect of leaving the CPU available for other tasks. Most developers feel that AI is growing in importance and will have a greater effect on games of the future [21].

State machines have a number of inherent problems. One such problem is referred to as duplicate state decision points as cited by [30, 13]. This is a software engineering issue that can be quite troublesome and could potentially break any State Machine. This problem usually occurs when code is not properly commented and documented. The result being states could be implemented that have conflicting or identical state decision points, or conflicting rules. Obviously this could cause a detrimental effect if your in-game tank suddenly finds itself in an incorrect state due to one of these conflicting rules. The second problem is more of a problem inherent in all state machines. Because state machines are deterministic by definition, the behaviour produced by a state machine is also deterministic. This leads to behaviour that is predictable. To avoid predictability, most developers are either using fuzzy state machines or fuzzy logic, even so these are still based on a model state machine model. Indeed most of the AI scripting that is being implemented for most games gets abstracted into some form of State Machine. While more exotic models for game AI are being looked into, developers with ever decreasing schedules are finding that the time required researching these other models are beyond what they can afford. So in the tradition of commercial industry, many new and exciting techniques are being put into the too hard bin as developers resort to the tried and tested techniques such as the finite state machine.

As the average bandwidth a user has available for communication increases with connections to the internet such as ASDL and Cable, games are starting to show a shift away from single user environments. Instead users are finding that games can be more challenging and generally be more fun due to multi-user aspects of the games. These games particularly are aimed at the multi-player environment and yet we find that AI could be just as important even though AI might not be required. Often users logon to a game server to find it empty just to leave and find another server. Another situation that might arise is where users find themselves in unbalanced teams, where the team of greater numbers dominates the game. What would be the result if believable BDI agents could be introduced to add some numbers to a realm or even out mismatched teams dynamically during game play and of course be removed when no longer needed? Such features would improve the experience of online gaming in general. The team aspect is of utmost importance when talking about games dependant on teamwork. In most cases AI is not required, however there are those times like those mentioned above, when plausible AI opponents would be of great value to the game. It must be noted that AI opponents can already be included most games, however these opponents are either ordinary in skill, or not plausible as a human player, ie too hard, or too easy.

3.3 A State Transition in Thinking

Recently there has been a real break from the established mould of game AI by the game Black & White by Lionhead Studios [21, 8]. The use of the BDI architecture as well as a number of cognitive learning skills show just how appropriate agent concepts are to these highly dynamic virtual environments. Included in Black & White are other agent concepts such as agent emotions, where game characters can become depressed and exhibit human reactions to these emotions such as binge eating. Obviously this adds whole new dimensions to the game experience. It could be that this is the start of things to come in game AI, citing games such as Black & White, The Sims, and others with sophisticated user interaction as proof of concept. Just as graphics have been a draw card for most games in the past, sophisticated AI could be a draw card for games being produced in the future. It is hoped that by producing games with greater intelligence, game characters will challenge the user and keep their attention for a greater period of time than they do currently. This change in focus shows that the game industry is beginning to mature from the 3 friends in a garage to an industry that is relying heavily on research and creative thinking coming from the academic world.

The way in which Game development is changing also reflects the growing focus of AI in games as figure 1 shows. There is a trend that most development teams are allocating at least 1 or more personnel specifically to AI which is in contrast to only a few years ago, where game AI was hacked together in the last month or so of the project by anyone who had time. In addition to this, as mentioned before, AI is being allocated more CPU time than it ever has been previously. This, coupled with the increased allocation of human resources to AI development, makes it clear that developers are giving AI a much higher priority than it has enjoyed in the past. We can therefore infer that we will see more of the sophisticated AI prevalent in Black & White in up future games.

3.4 Simulation Vs Game

There is a difference between a simulation and a game. However sometimes that line blurs, making the distinction difficult to identify. We find that simulations are often based on real world environments and physics. A game however, will embellish those truths in the interest of user entertainment. When someone decides to create a simulation, the object is to model something as close as possible as it is in the real world. The line that distinguishes simulations and games blurs during the design of a game when designers decide to make a game both enjoyable to the user and keep behaviour in the game realistic. Past research has seen Agents used in battle simulations [27, 28] for teamwork, and in games such as Pengi [1]. Even with research such as this, it seems that very little investigation has been done on direct comparisons between developing agent architectures such as BDI cognitive agents and the classic game AI implementations of state machines.

3.5 Atomic Bomberman

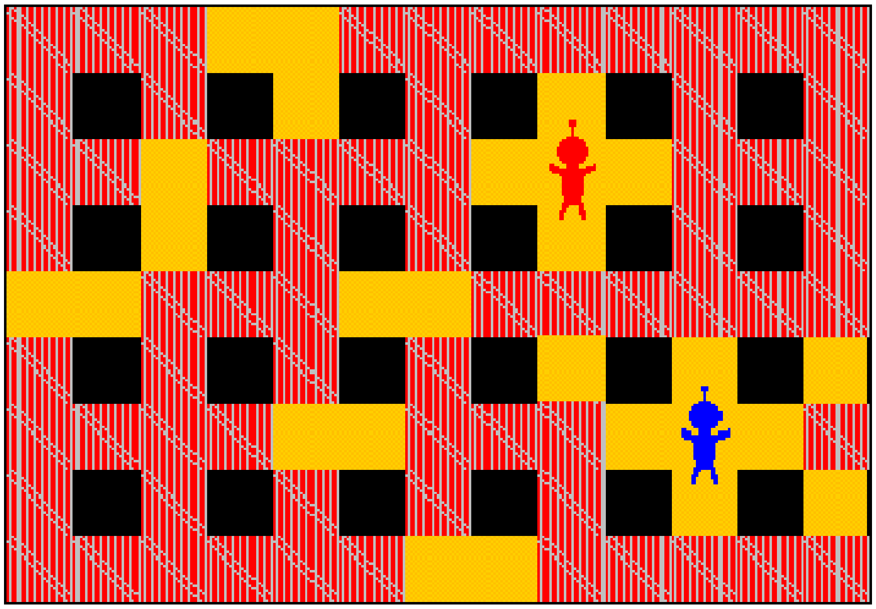


Figure 2: Blue and Red Bombermen at initial Starting locations.

Atomic Bomberman was a game produced by Interplay in the mid 1990's. Atomic Bomberman is a recreation of a much older game that has long been popular and was first released around the same time as Pac Man. It is a 2D game with an obvious state machine implementation, which often leads to predictable and simple behaviour. Figure 2 shows a possible start of game situation for a blue and red Bomberman. The objective of the game is to remove crates and create a path toward the enemy. Following this, they must try and trap their enemy and blow them up. A problem that faces the Bombermen is to move toward the enemy, without blowing themselves up. When contact has been made they must attempt not to get trapped themselves while endeavouring to trap their opponent. Bomberman is a dynamic environment that is changing constantly. This fact makes Bomberman a perfect candidate to test Agents. It must be noted that this research is not about Bomberman, but a direct comparison of the two models. Hence the game is subject to change in order to produce more accurate experimental results. Indeed changing the game by adding new game entities, and analysing how it effects the software development process, is an important part of this research.

4 An Experimental Comparison

The question was raised early in this research whether it would be a fair comparison to compare a state machine with an intelligent agent. Given the complexity originally intended for the agent model, and the relative simplicity of the state machines, the obvious answer was no. For this reason it was decided to attempt to keep the capabilities of the state machine and the agent reasonably even in the interest of a fair comparison. Because an experimental approach was used in order to make an accurate comparative study, the following chapter is broken up into types of comparison, eg. performance, software engineering issues, and iterative modification effort comparisons.

4.1 Performance Comparison

4.1.1 Average Decision Time

Hypothesis

It is expected we will find that the agent model will be slower than the state machine model. This is not a hard conclusion to come to when you consider the agent model has some JACK components included. These JACK components can be considered an overhead that the agent model must contend with. The question is just ***how fast*** is the Agent model in relation to the State Machine. Both implementations are similar and have the same capabilities, so this will allow us to test the architectures themselves. During development it was noted that decision time seemed comparable. To rule out any abnormal behaviour, this experiment ran the game 4000 times, and used the results produced to calculate the average.

Apparatus

To do this experiment we used the following items.

- 1 Bomberman game which will be used as the test environment.

- 1 State machine implementation of a Bomberman with 2 instances in the game.
- 1 intelligent agent implementation of a Bomberman with 2 instances in the game.
- 1 parser to parse the results of the experiment.
- 1 PIII 900 Computer with 128 MB of RAM.

Method

The method we used to setup and run the experiment is as follows.

1. Firstly we found a static environment on the fastest isolated PC we had available. This was so that we would see just how fast each model was at it's best.
2. We set up the experiment to run for 4000 games.
3. After this we broke the experiment into two sets so that 2000 games would be run in single player mode, while the following 2000 would be in team mode.
4. We gave each Bomberman (2 State Machines and 2 Agents) a starting location an equal distance from all other Bombermen.
5. To be sure that starting positions did not effect the results we had all Bombermen change to direct opposite starting positions half way through each set. This way we could be sure neither model could possibly be disadvantaged. You could probably compare this to changing ends of the court in tennis.
6. We let the experiment run until completion, which took approximately 4 days.
7. We put all results together and used a parser to add up the time taken for each decision and then divided the total by all the number of decisions made. The parser then produces an average for each model.
8. The parser was run twice, once on the team results and once on the single player results.

Results

The results of these experiments can be seen in Figures 3 and 4. The charts show the performance in milliseconds of both models over 2000 runs each. The results depict both the Agent model and the state Machine model. It is obvious from these figures that the state machine implementation has outperformed the agent model as expected. While both models had been programmed with similar

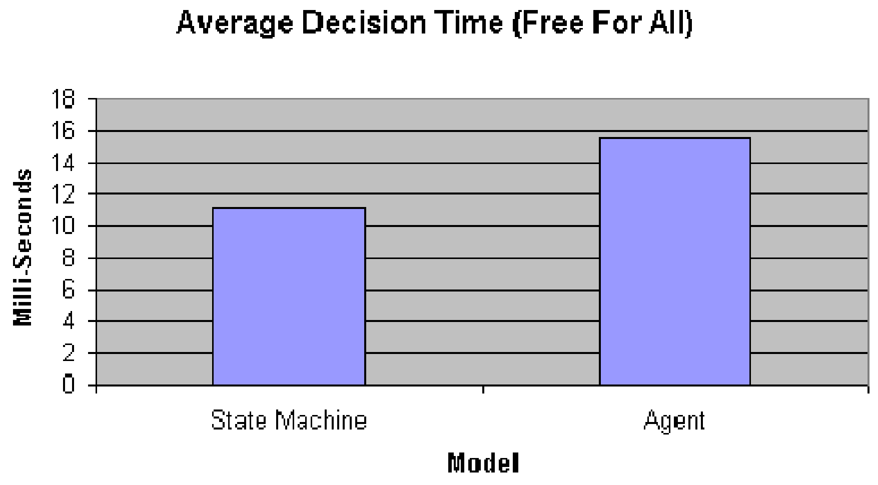


Figure 3: Performance of both models after 2000 runs in a 1 Vs Many situation

features, it appears the JACK overhead has had an effect on the agent implementation that the state machine model avoids. Over the 2000 runs of each experiment, we found that the agent model was slower only by an overall margin of 26%.

It must also be considered that the current Game Engine and Environment being used to run these experiments is reasonably simple. It is expected that as the required behaviour of the game AI gets more complicated and less specialised, we may find that the gap between the FSM and Agent run time could grow. However because the JACK overhead would remain constant, it is also entirely possible that the margin that separates these two models would remain the same. This could be a future experiment that could be run to see how performance is affected by increased complexity. Also this experiment would show whether the margin between the state machine implementation and the agent would grow or remain constant. If the margin were to remain constant, this would mean the agent models extra cost would be insignificant given a larger set of behaviours, making it a prime candidate for game AI. If the reverse is true then we could expect an agent implementation to quickly become unsustainable.

The final sets of the experiment were run with team play enabled for 2000 games. In team mode, the Bombermen are divided into two teams and will only attack Bombermen on the opposite team. In the current experiments, the behaviour of these models is relying on emergent team behaviour. Emergent team behaviour is due to the Bombermen distinguishing between friend and foe. As figures 3 and 4 show, there is little difference when emergent team behaviour is

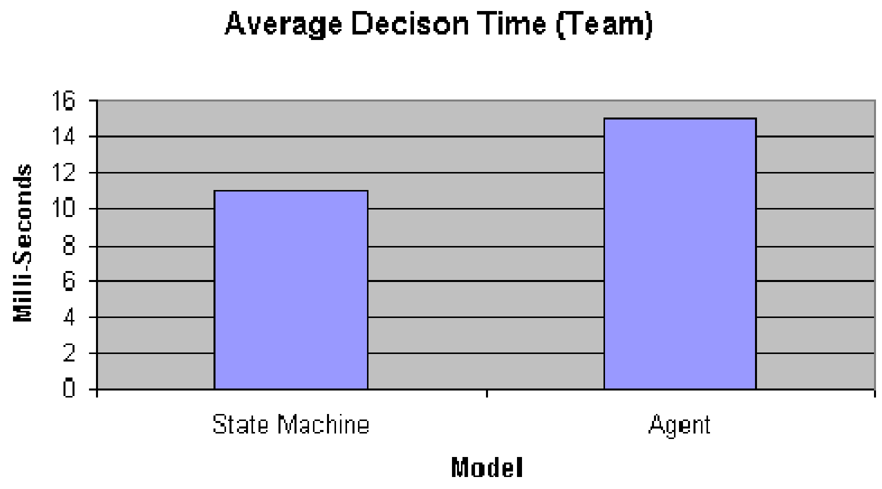


Figure 4: Performance of both models after 2000 runs in a Many Vs Many situation

introduced.

4.1.2 Performance Scalability

Hypothesis

It is thought that the experiment will show the agent to run slower than the state machine and that this margin will grow slightly over time.

Apparatus

The following items were used to under take this experiment

- 1 Bomberman game which will be used as the test environment.
- 1 State machine implementation of a Bomberman ranging from 2 instances to 11 in the game.
- 1 intelligent agent implementation of a Bomberman ranging from 2 instances to 11 in the game.
- 1 parser to parse the results of the experiment.
- 1 PII 350 Computer with 256 MB of RAM.

Method

The method used can be described as follows.

1. Firstly we found a static environment on the slowest isolated PC we had available. As the objective was to find a trend over time, it was decided

that a trend would be more obvious during worst case performance for both models.

2. We set up the experiment to run for 2000 games. and decided to leave the emergent team behaviour enabled. We did this so that fewer Bombermen would have to die to reach the final state. If you have 22 Bombermen, in team play, at least 11 need to die to end the game, in single play, at least 21 must die.

3. After this we broke the experiment into 20 sets. Games 1 - 100 were 2 State Machines Vs 2 State Machines. Games 101 - 200 were 2 agents Vs 2 agents. Games 201 - 300 were 3 State machines Vs 3 State Machines. Games 301 - 400 were 3 agents Vs 3 Agents. This went on until the final sets which were Games 1801 - 1900 for 11 State Machine Vs 11 State Machines. With the final set being 1901 - 2000 for 11 agents Vs 11 agents.

4. We attempted to give each additional pair of Bombermen equal spacing from every other Bomberman.

5. Because we were timing a complete cycle of decisions for a group of the one model, there was no need to swap positions.

6. We let the experiment run until completion, which took approximately 2.5 days.

7. We put all results together and used a parser to add up the total time for decisions for an entire set of games. It then added up the total number of cycles for the same set, and divided the total decision time by the total number of cycles. This gave us the average decision time for an entire cycle of a particular implementation.

8. The parser produced a separate average for every hundred games.

Results

Based on the last experiment we had a clue that the total cycle time required for a group of agents would increase over time. We did not expect this to increase so drastically as it does in figure 5. We also notice that the state machines performance while slightly more scattered than the agent model still shows a trend upwards which is natural. What we can gather from these results is that the overhead that the agents were displaying in experiment 1 has had an accumulative and devastating effect. So in other words, when you have 2 agents running per cycle, you have twice the overhead of just 1 agent.

What we are really interested in is the rate of increase shown in figure 5. For each additional bomberman agent, approximately 31 ms are added to the average cycle time. Whereas the FSM only

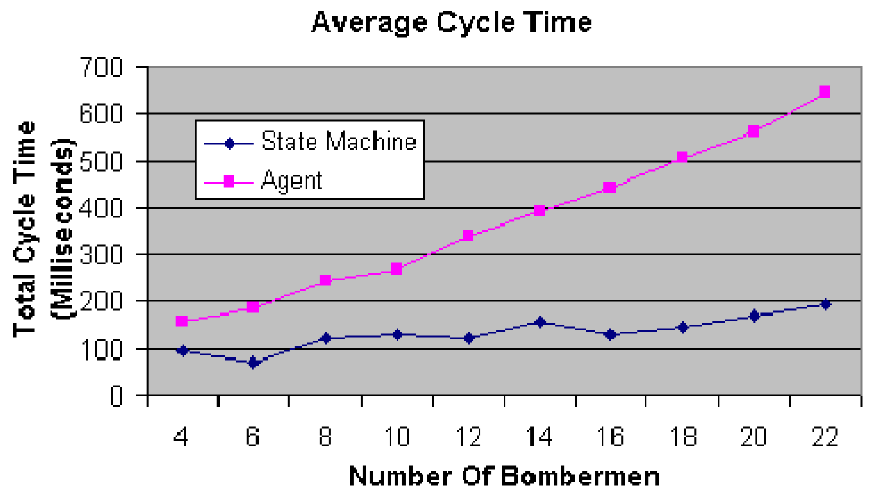


Figure 5: Performance of both models after 2000 runs, as the number of Bombermen increase by 2's from 4 to 22

adds approximately 7 ms. Both implementations show a reasonably linear rate of increase, though the FSM appears more erratic. The linear increase in the agent model could be partially due to the use of identical agents used in this study. It would be interesting to study the trend when many different agents involving many interactions among them were used.

The question remains as to why the massive increase for the agent model and the almost constant performance for the state machine. Obviously the agent is doing something that is fairly expensive, but we kept the agent definition and the state machine definition almost identical. Where could this expensive process have come from. The Answer to this question comes from JACK. When a JACK agent is built, it takes the definition and reduces it to a state machine. However as we have seen in our state machine, this isn't the problem, as we have a state machine that is running very quickly. Understanding that a lot of code has obviously been generated that was not defined by the programmer. By definition each JACK agent is a software component consisting of beliefs, events, goals and plans and the threading of some of these are taken care of internally within JACK's generated code. The State machine we defined may initialise one new state object and execute it, but we do not know exactly how many objects are being instantiated by JACK, or how many threads. Keeping in mind that thread initialisation is a notoriously expensive operation.

4.2 Comparison of effort and duration

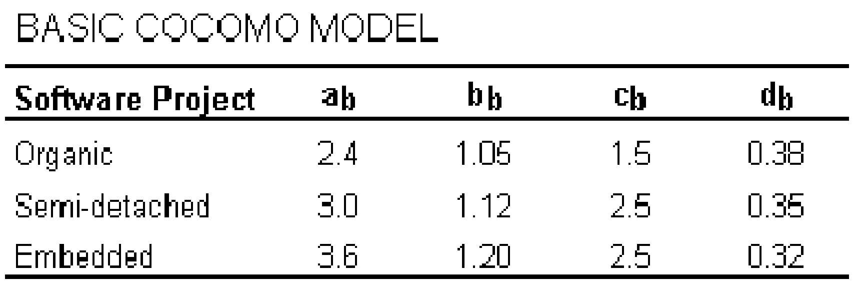
Correctly estimating the effort involved often determines the success or failure of any software project. The COnstructive COst MOdel (COCOMO) [22] is used by thousands of software project managers. It estimates the number of person months required based on the source lines on code (LOC). Though the advanced models take into consideration other factors such as precedence, team-cohesion and architecture. In our research we use a simple model based purely on LOC as other factors are common to both implementations. As the game developed is a simple one, we are particularly interested in the incremental effort involved as more and more behaviours are added. Thus in the second section we are measuring the incremental effort. This would help us to predict the effort required as a function of complexity.

In order to estimate and compare effort and project duration, we will need to collect a number of *tools* that we can use. The characteristics we will measure is the *Effort* required to develop each module, and the *Duration* required. These might be the deciding factors in which model to undertake. Software engineering methods exist, such as COCOMO and other empirical estimation models that are designed to measure exactly these code properties. Through these techniques we can get estimates on effort and duration in person months which we can use to determine development duration. While these methods are usually applied at the beginning of projects to produce estimates on staff and resource requirements, we will apply the same models to the completed Bomberman model we have created. Having the exact measure of LOC brings a certain amount of accuracy to these estimation techniques resulting in a more exact measure of effort and duration.

One tool we will use is an empirical technique known as the Boehm simple model. As a preliminary comparison metric this is a very broad-brush estimate of effort only. However it is valuable to be able to use a couple of estimation techniques and be able to verify the results produced. The formula to calculate the Boehm simple model is described in equation 1, where E is equal to effort in person months and KLOC is a reference to every thousand LOC. The initial values are constants that are usually calibrated with experience, and based on the type of project being measured. Given our case we will use the values verbatim for the Boehm simple model.

$$E = 3.2 \times (\text{KLOC})^{1.05} \tag{1}$$

To get a better indication of effort, we will apply another Software engineering tool based on KLOC, known as COCOMO. COCOMO is considered to be a more accurate technique as it has predefined


BASIC COCOMO MODEL

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	1.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 1: Basic COCOMO figures given a type of project. Values used have been extracted from [22]

constants based on the type of project as table 1 shows. We will evaluate both modules under the BASIC COCOMO model using organic mode. Organic mode being a small project with a small and experienced team working with requirements that are not very rigid. This description best suits the two modules we are comparing. In addition to these weights COCOMO uses the equations 2 to estimate effort and 3 to estimate duration. E remains estimated effort in person months, KLOC is still ever thousand LOC, and D is the estimated duration in calendar months.

$$E = a_b \mathrm{KLOC}^{b_b} \tag{2}$$

$$D = c_b E^{d_b} \tag{3}$$

4.2.1 Estimated Development Effort

Hypothesis

It is hypothesised that the State machine model would have a higher estimated effort than that of the agent model due to its size and its distribution of state transition logic. To be sure we are going to test this in two parts using two estimation techniques. Both these techniques will be based on the lines of code (LOC), under the premise that the number of logical errors can be directly related to how many LOC are present. While this is not an overly accurate experiment, this coupled with later experiments will add credibility to the comparison we are making. By giving us a measurement of effort for the agent and state machine implementation, we can compare these efforts to the effort of the modified models later.

Apparatus

To undertake this experiment we used the following items.

- The written code for the state machine module .
- The written code for the agent module.
- The cat and wc functions under Unix.
- The Boehm Simple Model of effort estimation. (See Equation 1)
- The Simple COCOMO Model of effort and duration estimation. (See table 1 and Equations 2 and 3)

Method

The method we used to do this experiment are as follows.

1. We isolated both modules so that we would only be dealing with the code for that module.
2. The code was prepared for the experiment by removing all commented lines and unwanted blank lines.
3. We executed the command *"cat * | wc -l"* on each module separately which concatenated the module together and counted every line. For the agent model we got a result of approximately 900 LOC and approximately 1000 LOC for the State machine model.
4. Now we used these values to calculate The Boehm Simple Model (Equation 1) where we must use every thousand lines of code (KLOC), and E is equal to the effort required in person months.
5. Simple estimation using Boehm model for the State Machine.
   $E = 3.2 \times (\text{KLOC})^{1.05}$
   $E = 3.2 \times (1)^{1.05}$
   $E = 3.2 \times (1)$
   $E = 3.2$ Person Months
6. Simple estimation using Boehm model for the Agent.
   $E = 3.2 \times (\text{KLOC})^{1.05}$
   $E = 3.2 \times (0.9)^{1.05}$
   $E = 3.2 \times (0.9)$
   $E = 2.9$ Person Months
7. To confirm the result we then used the COnstructive COst MOdel (COCOMO) to get a more accurate estimate effort and in addition to this an estimated duration (table 1 and Equations 2 and 3). KLOC and E retain the same meaning, however D is the duration
8. Before we could continue we had to evaluate the type that these models could be characterised as. We decided upon the organic mode.

9. COCOMO for the State Machine
   E = a_bKLOC^{b_b}
   E = $2.4(1)^{1.05}$
   E = 2.4 Person Months

   D = $c_b E^{d_b}$
   D = $2.5(2.4)^{0.38}$
   D = 3.5 Months

10. COCOMO for the Agent
   E = a_bKLOC^{b_b}
   E = $2.4(0.9)^{1.05}$
   E = 2.2 Person Months

   D = $c_b E^{d_b}$
   D = $2.5(2.2)^{0.38}$
   D = 3.4 Months

Results

Model	Boehm Effort	COCOMO Effort	COCOMO Duration
State Machine	3.2 Months	2.4 Months	3.5 Months
Agent	2.9 Months	2.2 Months	3.4 Months

Table 2: Boehm and COCOMO Estimation of effort based on KLOC

We find the State Machine implementation has a slightly higher degree of effort in addition to this it also has a slightly longer duration. It must be noted that these are only estimates and are almost totally based on lines of code. Neither COCOMO or the Boehm simple model take into account design of the module, or the time required to trace bugs and refactor. Even so, these comparisons are valuable in showing that the agent model may at least save anywhere from a few days to week in development time that could be crucial to debugging and testing.

Another observation is the ~10% more lines of code of the state Machine than that of the Agent Model. This is interesting considering the relative simplicity of these implementations of the AI in Bomberman. If one then consider that logical errors are dependent on an organisations usual error, yet will remain proportional to LOC [2], we can infer that the state machine implementation would have ~10% more logical errors than the agent module. If we look at this from the other angle, the converse of this is to extrapolate that an Agent implementation would have ~10% *fewer* logical errors when compared to a state Machine implementation and would still

produce the same behaviour. While both of these modules are relatively small, that ∼10% difference can make a huge difference when KLOC is increased from 1 thousand LOC to ∼7 thousand LOC.

4.2.2 Estimated Effort of Modification

Hypothesis

It is hypothesised that the State machine model would have a higher estimated effort than that of the agent model. This is assumed from the results of the last experiment, but the question is whether or not the margin that separates the two models will remain constant, or if we will notice a significant difference emerge. We will rerun the *Estimated Development Effort* on the modified code based on a design modification from *Incremental Modification* in the next section.

Apparatus

To undertake this experiment we used the following items.

- The written code for the modified state machine module .
- The written code for the modified agent module.
- The cat and wc functions under Unix.
- The Boehm Simple Model of effort estimation. (See Equation 1)
- The Simple COCOMO Model of effort and duration estimation. (See table 1 and Equations 2 and 3)

Method

We followed the same method we did for *Estimated Development Effort.*

1. We isolated both modules so that we would only be dealing with the code for that module.
2. The code was prepared for the experiment by removing all commented lines and unwanted blank lines.
3. We executed the command *"cat * | wc -l"* on each module separately which concatenated the module together and counted every line. For the agent model we got a result of approximately 1000 LOC and approximately 1200 LOC for the State machine model.
4. Now we used these values to calculate The Boehm Simple Model (Equation 1) where we must use KLOC to estimate E.

5. Simple estimation using Boehm model for the State Machine.
   $\text{E} = 3.2 \times (\text{KLOC})^{1.05}$
   $\text{E} = 3.2 \times (1.2)^{1.05}$
   $\text{E} = 3.2 \times (1.2)$
   E = 3.8 Person Months

6. Simple estimation using Boehm model for the Agent.
   $\text{E} = 3.2 \times (\text{KLOC})^{1.05}$
   $\text{E} = 3.2 \times (1)^{1.05}$
   $\text{E} = 3.2 \times (1)$
   E = 3.2 Person Months

7. To confirm the result we then used COCOMO to get a more accurate estimate of effort and an estimated duration (table 1 and Equations 2 and 3).

8. As in Experiment 3 we keep the same type of project and the same weights for organic mode.

9. COCOMO for the modified State Machine
   $\text{E} = a_b \text{KLOC}^{b_b}$
   $\text{E} = 2.4(1.2)^{1.05}$
   E = 2.9 Person Months

   $\text{D} = c_b E^{d_b}$
   $\text{D} = 2.5(2.9)^{0.38}$
   D = 3.7 Months

10. COCOMO for the modified Agent
   $\text{E} = a_b \text{KLOC}^{b_b}$
   $\text{E} = 2.4(1)^{1.05}$
   E = 2.4 Person Months

   $\text{D} = c_b E^{d_b}$
   $\text{D} = 2.5(2.4)^{0.38}$
   D = 3.5 Months

Results

Model	Original Effort	Effort After Modification	Effort Of Modification
State Machine	3.2 Months	3.8 Months	0.6 Months
Agent	2.9 Months	3.2 Months	0.3 Months

Table 3: Boehm estimation of effort before and after design modification in Person Months

Model	Original Effort	Effort After Modification	Effort Of Modification
State Machine	2.4 Months	2.9 Months	0.5 Months
Agent	2.2 Months	2.4 Months	0.2 Months

Table 4: COCOMO Estimation of effort before and after design modification in Person Months

Model	Original Duration	Duration After Modification	Duration Of Modification
State Machine	3.5 Months	3.7 Months	0.2 Months
Agent	3.4 Months	3.5 Months	0.1 Months

Table 5: COCOMO Estimation of Duration before and after design modification in calendar months

When we compare the results from the last two sections, we begin to see a trend that is very interesting. We have spread this comparison over three tables to make this relationship clearer. Table 3 describes the Boehm results from this experiment in comparison to the results from *Estimated Development Effort*. Table 4 shows the same information on effort as table 3, except that it is the result of COCOMO effort estimation rather than Boehm. Table 5 represents duration for both models according to effort.

In table 3 we notice that after the design modification, we have a much higher degree of effort for the state machine module. This is confirmed by the difference between effort for both modules, where the additional effort for the state machine modification is twice that of the agent modification. When we now compare the results in table 4 we again note a similar result. Although the estimations are far more conservative, we notice that the difference in effort according to COCOMO is more than double. While the actual estimations vary in some degree the margins by which the models differ shows a distinct trend that can be expected to continue.

In confirmation of the results above, table 5 shows a similar trend as the other tables, where the additional duration for the state machine model is at least double that of the agent module. When you consider that the LOC for the additional behaviour made up 20% of the original code, one realises why the effort and duration is so increased. Given that every new logical path creates added complexity, the few LOC one has to add to make a modification the better. Comparing this with the agent model the additional code makes up only 11% of the original code. In other words the percentage of code added to the state machine is almost double that of the agent, possibly explaining the increase in effort and duration.

4.3 Software Engineering Comparison

Game development typically follows a prototyping approach where new behaviours and game entities are added until a playable game that can sustain a users interest is created. Thus it is of utmost importance that the methodology used lends itself to such an approach. At the design stage this may involve adding new behaviours and changing the interactions among the entities. At the programming level this involves adding more logical paths which naturally adds to the complexity. To measure the design effort involved objectively, we have devised a control experiment where a design diagram had to be changed to reflect a new game entity, which required new behaviour for the Bombermen to handle this new entity. To measure the programming effort objectively, we have measured the additional logical paths created.

In the following experiments, it will be important for us to try and compare complexity. We have tools available to us that are in most cases accurate and helpful in this endeavour. One such tool is McCabe's complexity metric, which is capable of estimating a designs code complexity based on the number of logical paths. When McCabe's is applied it will produce a number which will correspond to a range in table 6. The higher the complexity of a design, the harder it is to maintain, update, and the higher risk of introducing bugs.

Cyclomatic Complexity	Risk Evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
greater than 50	untestable (very high risk)

Table 6: McCabe's cyclomatic complexity values. [22]

Cyclomatic complexity is a widely used metric for measuring soundness and confidence of a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly independent paths through a program or module.

4.3.1 Incremental Modification

Hypothesis

It is hypothesised that the State machine model would be highly coupled and therefore take much more time to extend. We have attempted to simplify the agent definition so that designers with next to no experience with agent concepts can evaluate the design

and make appropriate changes. We have kept the state machine design however, as all subjects have had some experience with this definition.

Apparatus

To undertake this experiment we used the following items.

- An agent Design.
- A State Machine Design.
- A clock of some kind
- Some volunteer agent and state machine designers

Method

The following method was taken during this experiment

1. First we distributed the designs to the subjects.
2. We gave a brief overview of both designs and a description of Bomberman.
3. We asked half the subjects to start with the agent definition, and the other half to start with the state machine definition.
4. We timed how long each took to modify, and noted it.
5. We compiled the results and found the average time of modification and then identified a common design implemented by the subjects.
6. The design modification was then added to the source code for both models

Results

Model	Total Classes Modified	Total Changes
Agent Module	3	7
State Machine Module	6	10

Table 7: A summary of all modifications required to add a simple change to behaviour. (See appendix for more detail

We identified the design change produced by the different participants and came up with a unified design that was the most representative. Figures 7 and 9 show the new design for both the state machine and the agent (Changes Highlighted).

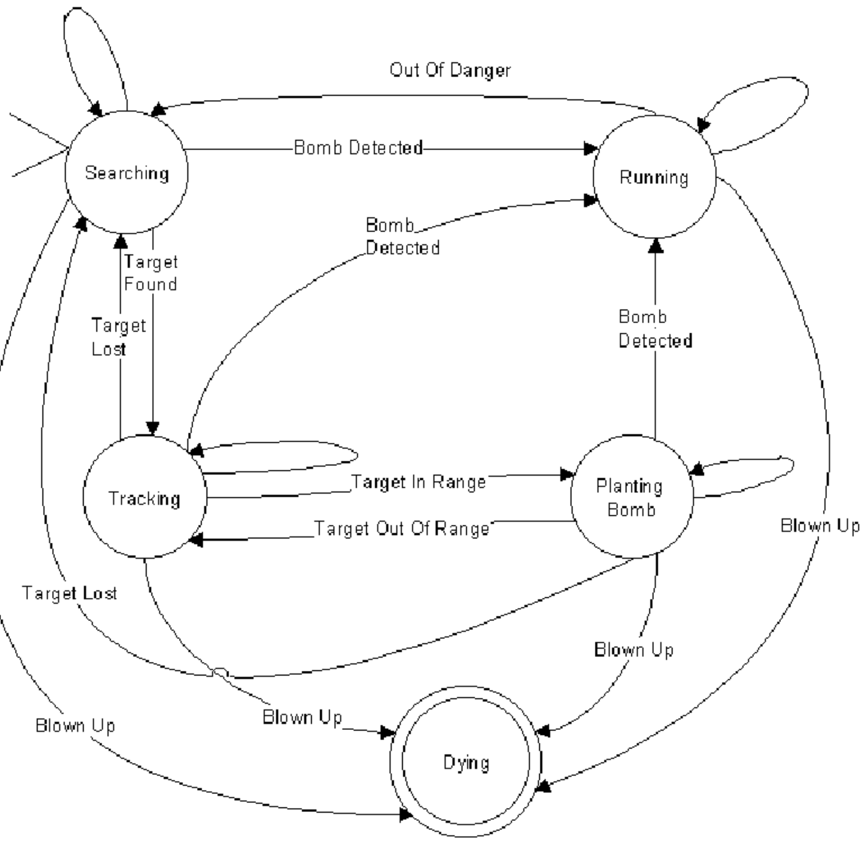


Figure 6: Initial State Machine definition for Bomberman before modification

Figure 6 is the original state machine definition while figure 7 is a new state machine definition, which introduces additional behaviour. Already we see that the state machine definition is becoming complex with transitions from one state to almost every other state. This means that a relatively minor change to the design could result in a new state and many new transitions between this state and others. We see exactly this situation in Figures 6 and 7 which is a simple state machine, modelling simple behaviour. If we consider a typical game using a complicated state machine, we might be dealing with tens of new transitions.

While we were developing the two models, it quickly became obvious that extending the original design of an agent equivalent to the State machine was going to be trivial. We were attributing this to the fact that the agent was designed and implemented after the state machine. However its ease of design is confirmed during the redesign experiment, where designers were able to redefine the agent in far less time than it took them to modify the state machine as table 8 shows. To ensure us of a fair comparison, half the participants were required to extend the agent version first, and the remaining

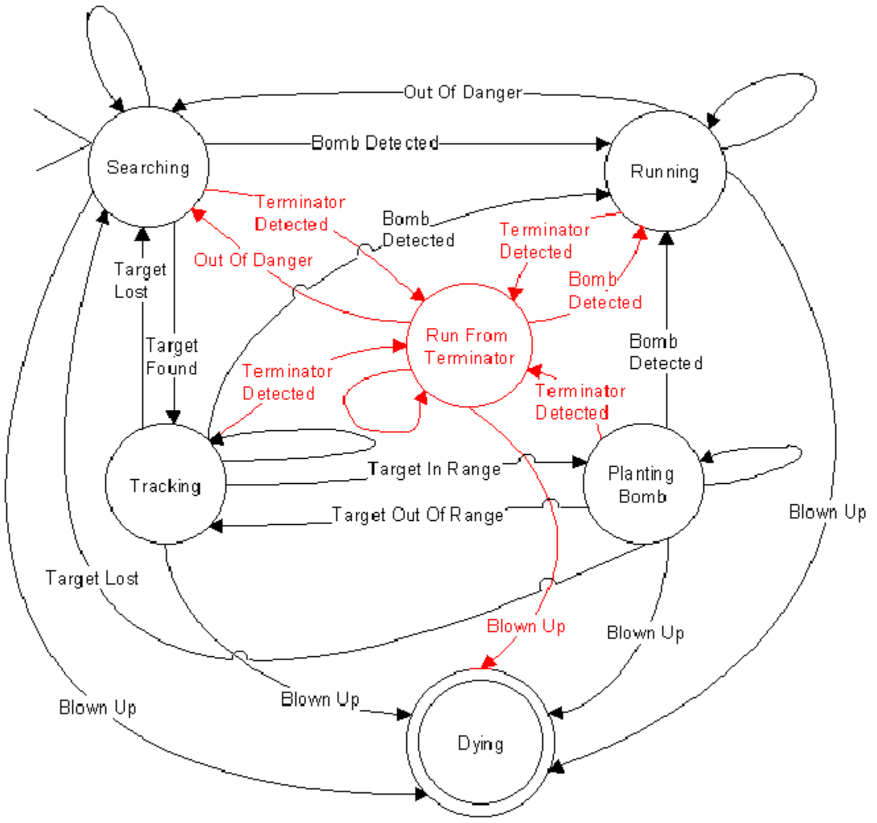


Figure 7: State Machine definition for Bomberman after modification (Changes shown in red)

vice-versa. If one compares the original specifications of the state machine (Figure 6) and the agent (Figure 8), one sees that both methods adequately describe the AI behaviours. After modification of the design, we see in figures 7 and 9 that the state machine definition becomes complicated. However the agent re-definition remains simple and understandable in comparison.

	State Machine First			Agent First		
	P1	P2	P3	P4	P5	P6
State Machine	7	8	4	4	2	2
Agent	1.5	1.5	1	1	1	0.5
Ratio	4.7:1	5.3:1	4:1	4:1	2:1	4:1

Table 8: The time taken to make modifications in minutes

In order to compare software-engineering aspects of these two models we decided to use software engineers in the experiment. We collected a number of individuals of varying skills, some with experience with agents, others with experience with State Machines. We

Agent Goals	Corresponding JACK events	Agent Plans To Execute
Escape Bomb	Bomb Detected	Escape Bomb
Move To Target	Target Out Of Range	Track Target
Find New Target	Target Not Found	Find Target
Blow Up Target	Target In Range	Place Bomb

Figure 8: A Simple agent definition that novice Agent designers can quickly understand.

Agent Goals	Corresponding JACK events	Agent Plans To Execute
Escape Bomb	Bomb Detected	Escape Bomb
Move To Target	Target Out Of Range	Track Target
Find New Target	Target Not Found	Find Target
Blow Up Target	Target In Range	Place Bomb
Escape Terminator	*Terminator Detected*	*Run From Terminator*

Figure 9: A common design modification produced by experimentation

also included individuals who have a high degree of software engineering experience but with little or no experience with agents or state machines. We provided these subjects with design specifications for both modules, and a short description of Bomberman and how to read these specifications. Finally we asked all those involved with the experiment to expand the behaviour of both the agent and the State Machine.

We noted that the time required to make the design modifications to the agent specification, was much shorter, and that the time taken to modify the state machine design was larger by a factor of ∼5 times. In addition to this many reported that they found the Agent design much easier to change. In contrast to this, they found the State machine design could quickly become complicated and required a much higher degree of fore thought as to the correct states and transitions. Some remarked that the agent design was a more human way of defining the behaviour.

4.3.2 Complexity of Incremental Modification

Hypothesis

It is hypothesised that the State machine model would have a higher complexity and that this complexity will grow drastically compared to the agent model. From our experience during development of the state machine, we noticed a single modification of behaviour would result in potential changes in every other state. It therefore makes sense to measure the complexity of both models to get an understanding of just how complex they are, and which is easier to

maintain and test. The agent model is expected to have much less complexity due to the concentration of its event generation compared to the relative high coupling of the state machine and its state transitions.

Apparatus

To undertake this experiment we used the following items.

- The original design for the agent
- The original design for the state machine
- The original source for the agent
- The original source for the state machine
- The modified design for the agent
- The modified design for the state machine
- The modified source for the agent
- The modified source for the state machine
- McCabe's complexity metric

Method

The following method was taken during this experiment.

1. We collected the designs and implementations that we had available from the last experiment.
2. Then we applied McCabe's to each model separately to try and determine the complexity of each models design.

Results

	McCabe's Complexity	
Model	Original	After Modification
State Machine	17	25
Agent	6	7

Table 9: Complexity of State transitions and Agent plan initiation

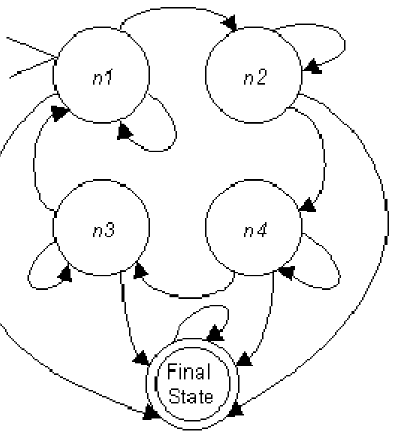


Figure 10: A small and simple State Machine

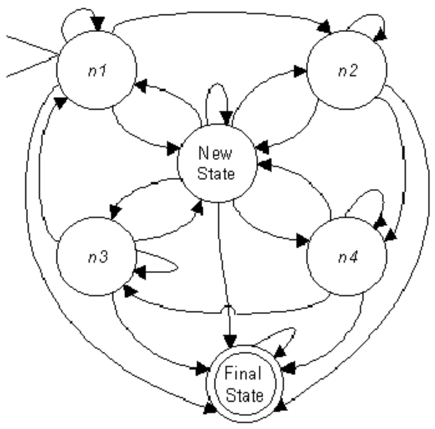


Figure 11: A worst case addition to the small and simple state machine in figure 10

Many tools are at our disposal when we need to get some understanding of complexity and design issues. We used McCabe's cyclomatic complexity [22] metric to gauge the complexity of state transitions and plan generation for both modules. Table 9 shows the results of McCabe's on both models for the original implementation and the complexity after the modification to behaviour. If we apply McCabe's to these modifications we just described, and we compare them to the original McCabe's shown in table 9 we find that the two sets of complexity show some very interesting and provocative results.

We see in table 9 that already there is a huge difference in complexity between the Original implementation and the modified version of the state machine. In addition to this we notice a difference

between the two models even though both models exhibit the same behaviour. To add further meaning to these values with regard to testing and maintainability, we use table 6 and notice that original implementation of the state machine module was already in the moderate risk of bugs range. This can be attributed to the greater number of if/else if statements that manage the state transitions. We could design a very simple state machine to reduce this complexity, however this would sacrifice behavioural complexity. We can say that the higher the number of state transitions directly affects the inherent complexity from McCabe's. Adding more transitions will result in new logical paths, adding to the modules complexity. In short, complexity both code and behavioural will be affected given either of these 2 cases:

1. We increase the number of states.
2. or, we increase the number of transitions.

While both 1 and 2 will increase complexity, we realise that by increasing the number of states, we must increase the number of transitions as a natural consequence, so these two are related. When we compare figures 6 and 7 to table 9, we notice that McCabe's produces a value of complexity that is equal to the number of transitions. This demonstrates that any additional behaviour introduced will increase the state machines complexity. In a worst case scenario, a new state may require a transition to every other state, and transitions from every state to our new state, as shown in figures 10 and 11. We can describe this relationship with equation 4, where AC is the worst case Additional Complexity, and n is the number of states.

$$AC = 2n \tag{4}$$

We have now identified the relationship between state machines, state machine behaviour, and the resulting complexity from McCabe's. If we are to have a proper understanding of the wide discrepancy McCabe's produces in table 9, we need to identify the relationship between the Agent definition, Agent behaviour, and it's resulting complexity. McCabe's essentially relies on the number of logical paths to gauge complexity. These paths often come from the any comparison ie condition statements, loops etc. We define our state machine through rule based transitions, which translate into condition statements, the more rules of behaviour, the higher the complexity. However in the case of agent design we use more of a rule based evaluation, where the agent evaluates its situation and posts a goal to achieve. The agent reduces its number of logical paths by concentrating it evaluations into one block of evaluation statements, simplifying the design.

5 Discussion

In this research, the well-known game Bomberman was produced using both a FSM and an agent implementation. The behaviour for both implementations were deliberately kept similar and simple with a limited number of states and plans. Our motivation was primarily to objectively measure how the performance and the software engineering metrics vary as the game scales up with more behaviours and characters. The rules of the game were changed where necessary, to facilitate objective measurements. The study produced some interesting and surprising results with regard to complexity, effort, and performance.

5.1 Complexity

As game development often uses a prototyping approach, it is essential that the design and code created be easily maintainable and extendable.

(a). *Design Complexity*
Though we expected the design to be more extendable with the agent model, the results far exceeded our expectations. It was evident from the experiments, that adding behaviour to the FSM design was taking approximately four to five times that of the agent design. In addition to this greater time required, many participants failed to synthesise a correct extension to the FSM design. Considering the simplicity of the game produced and the small number of behaviours being dealt with, this is a significant finding. This could be attributed to the high coupling of the FSM. Adding a new state to a state machine which has n states would introduce up to $2n$ *new* transitions, in the worst case. Hence we expect the difference in design complexity to become significant as the game is scaled up.

(b). *Code Complexity*
An important factor in maintainability and testability is that of code complexity. Through McCabe's complexity metric we were able to show that the implementation of the state machine was far more complex than that of the agent. We did not measure the total complexity of each module as a lot of the complexity was shared. What we measured was the *rule based evaluation* processes as this was very much the core difference between the implementations. The *rule based evaluation* for the state machine is distributed between the states, and represents transitions between states. The *rule based evaluation* for the agent is concentrated in the agent definition and is concerned with event generation which initiates plans.

To measure the additional complexity of a design modification, the proposed changes by the test group were implemented in both models. We compared the results of the first McCabe's results to those of the second and found that the state machine moved from moderate risk into a high risk range. In contrast to this, the agent definition remained within the simple module range and only increased by 1 point of complexity compared to the state machine's 8. This higher complexity can be attributed to the distributed and highly coupled nature of the state machines *rule based evaluation.*

With design complexity, every additional behaviour results in $2n$ new transitions, in the worst case. Summing it over t behaviours, we end up with a number of new transitions, equal to the result of the equation:

$$\sum_{n=1}^{n=t} 2n = t^2 + 1$$

Adding new behaviour with BDI involves adding a new plan. Hence the complexity for t new behaviours is just t. Code complexity too is directly related to the number of transitions. As can be seen, the complexity for FSM and agent are of a different order of magnitude, quadratic and linear respectively. We can safely conclude that the agent model lends itself to more *complex* and *intelligent* systems, when compared with the FSM.

5.2 Effort

There are essentially two types of effort that we are concerned with in this work, effort to develop, and effort to extend. Due to the prototyping approach taken up by most game developers, the effort to extend is just as important as effort to develop.

(a). Effort to Develop
We didn't know which of the two methods would involve less *effort*, nor how we would objectively measure effort. It would be impossible to time the development for both models as whichever model was developed first would be at a disadvantage. We decided to use empirical estimation based on lines of code and were rewarded with reasonably accurate estimates made in an objective manner. The results were not overly exciting as the difference in effort of development was in the Agents favour by only a matter of days to a couple weeks.

(b). Effort to Extend
We decided to extend the behaviour of our Bombermen and then applied the changes to both versions. The state machine implementation had an increase of effort that was more than double that of the agent. This allows us to predict that as the game scales up with more behaviours, the effort will be far greater for the state machine.

The results clearly show that the effort involved in FSM is likely to grow *significantly* as the number of behaviours increase. Hence from the effort point of view, the agent methodology will lend itself better for the newer type of games emerging in the market.

5.3 Performance

Game developers usually try to target the widest audience in terms of computational power. The choice of model would have a great bearing on the run-time performance and hence the computational power required. Therefore we are interested in comparing the performance of a single agent with a state machine and how the performance degrades with increasing scale.

(a). Singular Performance (comparing a single instance)
We were expecting the agents performance to be far worse than the state machine. We were excited to find that the speed of the two implementations were quite comparable when the number of game entities were few. We also found through our experiments, that emergent team behaviour did not contribute significantly towards performance degradation. However, based on all the results we were able to conclude that the agent had a small additional overhead associated with it.

(b). Accumulative Performance (comparing a group of instances)
We decided to time the cycle duration as the number of entities increase, first with the agent, and then for the state machine. These results were the most useful of the performance results as it showed that both models degraded in a linear manner in proportion to the number of Bombermen. While both displayed a linear decrease in performance, the agent model degraded at a far steeper rate. This showed that while the additional overhead was manageable when the number of agents is small, the overhead quickly accumulated, restricting the number of agents that could be run in real-time. Due to the linear nature of the degradation, it would also be possible to predict the performance degradation given n Bombermen.

The linear increase in cycle time for both models allows game developers to easily work out the limiting factor in terms of scale. Given the higher rate of increase, the state machine will naturally result in a lower threshold. Though we measured the rate of decay to be much greater for the agent, it is obviously dependent on the specific game. However we could safely say, any highly interactive game produced using FSM can be run on a wider rage of computers with varying CPU speeds, when compared to the equivalent game produced using an agent.

6 Conclusion

The conclusions we can draw from this research can be distilled down into three categories, design and code complexity, effort with regard to development and maintenance, and finally run-time performance.

6.1 Complexity

- The design complexity of the state machine (quadratic) and the agent (linear) are of a different order.
- Code complexity for the state machine increases *significantly* as the game scales up with more behaviours. For the agent model, the complexity increases linearly.
- We conclude that the agent model is more suited to the new type of *intelligent* games emerging in the market, from a complexity point of view.

6.2 Effort

- The effort required to produce the same behaviour was higher for FSM.
- As the game scales up with more behaviours the effort increases substantially for FSM.

6.3 Performance

- The run-time performance for FSM and agent model is comparable when the game entities involved are few.
- There is a linear increase in cycle time for both models, though the overhead is much higher for the agents. This makes FSM the natural choice when the number of entities are large and the speed is critical.

7 Further Work

7.1 Team Behaviour

Team behaviour was not thoroughly tested during this research, as time constraints prevented any significant progress. This makes team behaviour a natural extension of this research. Initial investigations showed that an agent implementation of team behaviour would not be overly difficult. Pursuing this line of research would not be possible without a state machine equivalent. It would be of particular interest to measure how the team behaviour affects complexity and performance issues.

7.2 Hybrid Approach

Some research must be done to see if a Hybrid model could be produced that take advantage of a state machines performance and an agents ease of design and coding. To our knowledge there has been little investigation into this topic. Some possible hybrids could be.

- An agent which controls the state transitions for *all* state machines. In this way only a single agent would ever be created, reducing the accumulative overhead, while taking advantage of a state machines performance and centralising its *rule based evaluations* in the agent.

- An agent definition that could only post specific events, given the evaluation of an internal state.

This is by no means an exhaustive list of configurations, yet could serve as a starting point in any research into a hybrid approach.

Acknowledgments

Special thanks must be extended to a person who came to a confused undergraduate and upon hearing the Student's intentions to undertake an honours course volunteered to be my supervisor. Charles Thevathayan and his tireless efforts, his understanding, and his keeping me on the straight and narrow through the year will always be appreciated. Another Special thanks must go to Peter Bertok for his support and guidance in fine tuning the direction of the research and for the benefit of his many years of experience as a research supervisor.

I would also like to thank the Agents at RMIT group, for their indirect and direct support through the year especially James Harland and Michael Winikoff. Thanks must also go to those who participated in the design complexity experiments, whose participation came at exactly the right moment.

In addition I must thank my family and friends for their understanding, support, and encouragement. Particularly my parents Norma and Robert, and a true friend Rein Van Noppen.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity.

[2] Eric J. Braude. In *Software Engineering An Object-Oriented perspective*, pages 110–114, John Wiley & Sons, Inc, 2001.

[3] Rodney A. Brooks. Intelligence without reason. 1991.

[4] Cristiano Castelfranchi. Modeling social action for ai agents. 1997.

[5] Lawrence Cavedon, Anand Rao, Liz Sonenberg, and Gil Tidhar. Teamwork via team plans in intelligent autonomous agent systems.

[6] Simon Ch'ng and Lin Padghim. From roles to teamwork: a framework and architecture. 1997.

[7] Philip R. Cohen and Hector J Levesque. Intention is choice with commitment. 1990.

[8] Richard Evans. The future of ai in games: A personal view. *Game Developer*, pages 46 – 49, August 2001.

[9] Jacques Ferber and Alex Drogoul. Using reactive multi-agentsystems in simulation and problem solving. 1992.

[10] James R. Firby. An investigation into reactive planning in complex domains. 1987.

[11] Michael Fisher and Michael Wooldridge. Executable temporal logic for distributed a.i. 1993.

[12] Stan Franklin and Art Graesser. Is it an agent, or just a program: A taxonomy for autonomous agents. 1996.

[13] Chis Hecker and Zachary Booth Simpson. State machine aka (non)deterministic finite state machine, finite state automata, flow chart. *Game Developer*, page 8, January 2001.

[14] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. 1998.

[15] Harry R. Lewis and Christo H. Papadimitriou. In *Elements of the Theory of Computation Second Edition*, pages 55–111, Prentice-Hall, inc, 1998.

[16] Agent Oriented Software (AOS) Pty Ltd. Jack intelligent agents, practicals. 2001.

[17] Agent Oriented Software (AOS) Pty Ltd. Jack intelligent agents, user guide. 2001.

[18] Agent Oriented Software (AOS) Pty Ltd. Simpleteam technical brief. 2001.

[19] Pattie Maes. Modeling adaptive autonomous agents. 1994.

[20] Jorg P. Miller. Control architectures for autonomous and interacting agents: A survey. 1998.

[21] Peter Molyneux. Postmortem: Lionhead studios' black & white. *Game Developer*, page 8, June 2001.

[22] Roger S. Pressman. In *Software Engineering A Practitioner's Approach*, pages 120–125, McGraw-Hill Companies, inc, 1997.

[23] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. 1991.

[24] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. 1995.

[25] Yaov Shoham. Agent-oriented programming. 1992.

[26] Yoav Shoham and Moshe Tennenholtz. On the emergence of social conventions: modeling, analysis, and simulations. 1995.

[27] Milind Tambe. Executing team plans in dynamic, multi-agent domains. 1996.

[28] Milind Tambe. Towards flexible teamwork. 1997.

[29] John Thangarajah. Representation of goals in the belief-desire-intention model. 2000.

[30] Dave Weinstein. State decision and consequence separation aka duplicated state decision points. *Game Developer*, page 13, December 2000.

[31] Steven Woodcock. Game ai: The state of the industry 2000 - 2001. it's not just art, it's engineering. *Game Developer*, pages 36 – 44, August 2001.

[32] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. 1995.

[33] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. 1997.

[34] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. 1999.

Appendix

A List Of Modifications

Below are all the changes that were required to make a change of behaviour to both the agent model and the state machine. These were also used to produce table 7

A.1 State Machine

Class RunFromTerminator
Create New Empty State
Define Internal State Transitions to Running, Searching, & Dying
Define RunFromTerminator State processing
Define helper method pickRandom(int[][] totalMoves)
Define helper method isAwayFromTerminator(int[] currentLocation, int[] legalMove, World w) in RunFromTerminator state

Class State
Define helper method terminatorClose(int[] currentLocation, World w)

Class PlantingBomb
Add Internal State Transition to RunFromTerminator

Class Running
Add Internal State Transition to RunFromTerminator

Class Searching
Add Internal State Transition to RunFromTerminator

Class Tracking
Add Internal State Transition to RunFromTerminator

A.2 Agent

TerminatorClose event
Create new Event TerminatorClose

EscapeTerminator plan
Create new Plan EscapeTerminator
Define helper method pickRandom(int[][] totalMoves)
Define helper method isAwayFromTerminator(int[] currentLocation, int[] legalMove, World w)

BombermanAgent agent
Allow TerminatorClose Event to be fired
Define helper method terminatorClose(int[] currentLocation, World w)
Add to Handle and Post definitions