Automated Design for Playability in Computer Game Agents

Scott Watson, Wolfgang Banzhaf and Andrew Vardy

***Abstract*—This paper explores whether a novel approach to the creation of agent controllers has potential to overcome some of the drawbacks that have prevented novel controller architectures from being widely implemented. This is done by using an evolutionary algorithm to generate finite state machine controllers for agents in a simple role playing game. The concept of minimally playable games is introduced to serve as the basis of a method of evaluating the fitness of a game's agent controllers.**

I. Introduction

Considerable research has been carried out into how to make better control architectures for agents in computer games [1], [2], [3], [4], [5], [6], [7]. Some of these architectures are very sophisticated and boast impressive capabilities. Despite these achievements, very little of this research has made its way into commercial computer games [1], [5], [8], [9]. This paper seeks to address some similar goals from a perspective on which relatively little work has been done - improving the process of how existing agent controllers are made rather than improving the agent controllers themselves.

The most fundamental requirement of a set of controllers to control the NPCs in a game is that the controllers will lead to a game that is possible for the player to complete. We call a game which it is possible to complete *minimally playable*. A simple game specification language is used to establish whether an evolutionary algorithm can generate controllers for agents in such a way that desirable gameplay properties are obtained. The contributions made by this paper are the introduction of the concept of minimally playable games and the description and preliminary validation of the concept of evolving agent controllers that satisfy minimal playability.

II. Background

A. Computer Role Playing Games

Role Playing Games (RPGs) are a subset of computer games that place emphasis on the development of the character controlled by the player, their importance in the game world and the influence they have on the world. These games often place a lot of importance on carefully crafted storylines that the player's character plays a central role in. The game worlds in RPGs can be extremely large in scope and complexity. The player often has a great deal of freedom to explore the world in whatever manner they see fit. RPG game worlds can be inhabited by thousands of non player characters (NPCs).

Scott Watson and Wolfgang Banzhaf are with the Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland and Labrador, Canada. (e-mails: saw104@mun.ca, banzhaf@mun.ca). Andrew Vardy is jointly appointed to the Departments of Computer Science and Electrical and Computer Engineering, Memorial University, St. John's, Newfoundland and Labrador, Canada (email: av@mun.ca).

B. Agents in Role Playing Games

Typically a great deal of the game experience in RPGs is based on the player's interaction with NPCs. This can vary from very simple interactions based on fighting and defeating a character to more complicated interactions such as conversation, trade or negotiation. In many cases, the social landscape of the game environment influences the interaction. For instance a friendly agent will behave differently towards the player than an antagonistic agent. Because of the variety of interactions that are potentially required to be handled by the agent controllers in RPGs, and the scope for future experimentation that this offers, RPGs were chosen as the genre to focus on.

C. Finite State Machines

Finite State Machines (FSMs) are models of computation defined by a finite list of states and a finite list of transition rules. Each transition rule controls which state the machine moves to for a given input. FSMs can be applied in a large variety of domains. Their strengths include conceptual simplicity, fast execution speeds, and ease of implementation.

Finite State Machines are used extensively in computer games [3][6][8][10][11][12]. FSMs can achieve good results but are rigid and cannot deal with situations not explicitly prescribed for by the developer. Human players are becoming adept at predicting behaviour by learning the rules of the FSM [8]. FSMs are easy to test, modify and customize [13]. FSM were used in Doom and Quake, among many others.

Efforts have been made to augment the classic FSM to increase its functionality. Fuzzy State Machines (FuSMs) have come into fashion to give less binary behaviours [12][13]. Fuzzy logic allows unpredictable behaviours to be generated based on traits of the agents which are modeled as decision thresholds. FuSMs were used in Unreal, S.W.A.T.2 and Civilization: Call to Power [13]. Gruenwoldt et al. attempted to use a dynamic relationship graph to modulate basic FSM behaviour [14][15][16].

D. Related Work

To the best of our knowledge there is no related work pertaining to the evolution of FSMs to control game agents. Certainly there is no mention of it in Fairclough et al's 2001 discussion of research directions for AI in computer games [8], Johnson and Wile's 2001 survey of AI in computer games [13], Lucas's 2006 overview of evolutionary computation in games [9], Togelius et al's 2011 survey of procedural content generation [17], Hocine and Gouaich's 2011 survey of agent

978-1-4799-3547-5/14/1.00

programming in serious games [10] or Hendrikx et al's 2011 survey of procedural content generation [18].

There are however a number of works that have analagous approaches. One interesting example among many is the work of Spronck et al. that seeks to recombine low level behaviour units into progressively more effective controllers in an iterative fashion [19].

E. Motivation

Creating, testing and maintaining finite state machines are time intensive processes. Being able to automate the creation of these machines will be advantageous for game developers in terms of the labour saved and accelerated development time [6], [10], [17], [20]. If the creation of a FSM for an agent is not associated with a significant cost in terms of labour, it may become feasible to handle larger populations of agents and use fewer duplicate controllers among the agent populations of a game. It is also quite feasible that larger, more complicated finite state machines could be viably produced, potentially leading to superior results in agent behaviour [10], [17].

III. Controller Generation Model

A. System Overview

The high level view of the system and its operation is given in figure 1. A developer provides an input and output specification and an evolutionary algorithm uses these to produce a set of controllers to be used by the agents in the game.

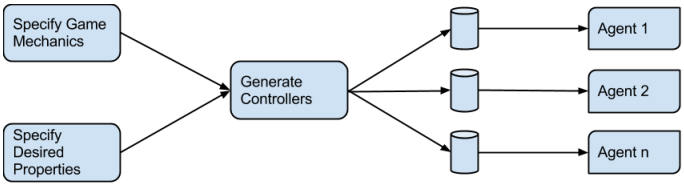

Fig. 1. System Overview

As described in Section I, the most important constraint of a set of controllers is to satisfy minimal playability. In this context, completion is taken to mean the achievement of some pre-defined victory objective. Examples of victory objectives might include slaying a particular NPC, obtaining a particular item or learning a particular fact. If it can be demonstrated that a set of candidate controllers can form a game, we consider that controller set to be *viable*.

For every game victory objective, there may be multiple viable controller sets. For games featuring even a moderate number of agents, actions and items this number may be extremely large. It is therefore useful to consider how to compare viable controller sets in terms of desirability. The desirability of a given controller set can be explored by assessing the desirability of the game produced by using that controller set to control its agents. Attributes of a game that allow this assessment can be inferred from the information gathered when checking if a controller set is viable.

To check if a controller set is viable, it is sufficient to check the actions the player can carry out in order to achieve the victory objective. A sequence of such actions constitues a *path* to victory. A controller set with at least one path is viable. A controller set may have many paths. The path(s) associated with a controller set can be used to compute certain attributes of the game-play experience a player would get by playing a game formed using this controller set. The precise nature of the attributes that could be calculated depends on the description of the game itself. Examples might include the number of actions the player must carry out to satisfy the victory objective, the number of agents the player must interact with to satisfy the victory objective or the types of actions that the player must carry out to satisfy the victory objective.

B. Agents

Each agent belongs to a social group. The groups are respectively $friendly$, $neutral$ and $hostile$ towards the player. Each agent has a set of items and a set of facts when the game starts. These sets may be empty. Items are unique and can only be held by one agent at a time. Facts can be duplicated and can be held by many agents at a time.

The agents in the game are always in one of the declared states. The agents only change states in response to actions by the player. Their transitions between states are produced by the generation process. The inputs which trigger transition table lookup are of the form $<currentAgentState, playerAction, socialRelation>$ where $currentAgentState$ is the state the agent is currently in, $playerAction$ is the action the player carried out that the agent is reacting to and $socialRelation$ is the relation between the player and the social group the agent belongs to. The state an agent is in controls the outcome of any actions the player carries out involving that agent. Figure 2 shows a representative controller for a limited set of states and actions.

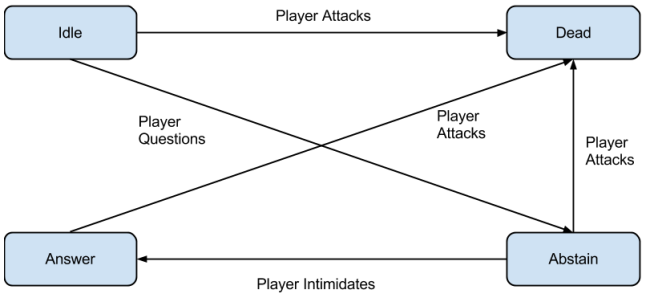

Fig. 2. Example of agent controller for a limited set of states and actions

C. Input Specification

- List of Items - A list of items present in the game world is specified by the ID number of each item.
- List of Facts - A list of 'facts' present in the game world is specified by the ID number of each fact. Facts in this scenario are highly abstract, effectively serving as boolean flags to enable certain things to happen such as

being able to see an invisible agent or being able to use a certain action. Facts differ from items in that facts can be duplicated. Unlike items, facts can also be 'lost' if all agents possessing the fact are killed. Finally, the actions that can lead to the player gaining a fact are different from those that lead to the player gaining an item.

- List of Agents - A list of agents present in the game specified by the ID number, social group, facts and items of each agent. These agents can be in one of three social groups which are respectively friendly, neutral, and hostile to the player. This relation forms a passive input to state transitions.
- Item Mappings - A mapping of items to the agents that have possession of them, specified by their respective ID numbers.
- Fact Mappings - A mapping of facts to the agents that have possession of them, specified by their respective ID numbers.
- Trade Mappings - A mapping of items to items that should be accepted in exchange as part of the trade action, specified by their respective ID numbers. This mapping is consulted when evaluating valid chains of actions to achieve victory. If a trade action is evaluated as part of a search for how to acquire an item a, a recursive search for how to acquire the item b specified in the mapping such that $a \rightarrow b$ must be performed, and those actions prepended to the action chain.
- Action Descriptions - A list of actions available to the player. Each action is represented by a name, a set of states an agent can be in to initiate the action, a set of states an agent can be in if the action is succesful, a flag to indicate whether the action can result in the acquisition of an item, a flag to indicate whether the action can result in the acquisition of a fact, the (possibly null) ID of an item required to perform the action and the (possibly null) ID of an fact required to perform the action.
- States - A list of states that agents can be in.
- Victory Objective - The 'end goal' of the game is specified by the type of objective (obtain item or fact) and the ID number of the item or fact that is to be obtained.

D. Output Constraints

Output refers to the game created by using a given set of generated controllers. Constraints on this output are therefore constraints on the game-play produced. This allows the fitness function to direct the search towards specific game-play objectives. The developer has the freedom to specify as many or as few constraints as they are interested in. There is a trade-off between the expressiveness and conciseness of the specification format.

1) Required Transitions:
A list of state transitions required to be generated for the agent controllers. These are specified by the input tuple of $< currentAgentState, playerAction, socialRelation >$.

For example, $<idle, attack, hostile>$ indicates that a transition must be generated to handle the event when a player attacks an agent that is hostile to them and is currently in the idle state.

If no transitions are given, transitions will be generated for all transitions possible from the lists of states and actions. Having the ability to restrict the transitions generated is useful if some transitions are going to be manually created or if certain transitions will be shared accross multiple agents. For example if all agents in a certain social group should respond to the player attacking them by running away, that transition need not be generated. This reduces the size of the search space.

2) Valid Transition Mappings:
These mappings define valid and invalid transitions by specifying a set of valid states that can be transitioned to from a given input tuple of the form <currentAgentState, playerAction, socialRelation>. This mapping can be as exhaustive or sparse as desired. If no mapping exists for a required transition, an implicit mapping to the set of all states is used. This allows the developer to maintain plausibility by forbidding undesirable transitions. An example of an undesirable transition might be an agent moving from a 'dead' state to any other state.

For example, $<idle, attack, hostile> \rightarrow < flee, dead, attack>$ indicates that if the player attacks an agent that is hostile to them and is currently in the idle state, that agent must change its state to one of the dead, flee, or attack states. If a transition is not mapped, it indicates the agent can change to any state.

3) Desired Actions:
A set of actions that the developer wishes the player to use. There is at least one sequence of actions that leads to satisfaction of the victory objective for every action in the desired action set.

E. Solution Encoding

If the transition table is considered to be a mapping from input tuple of the form $<currentAgentState, playerAction, socialRelation>$ to output state $newState$, the solution is encoded as the $newState$ component of each mapping designated as required in the input specification.

States are mapped to integers and the solution is represented as a list of integers.

For example, given:

- A set of required transitions
  - $<idle, attack, hostile>$
  - $<idle, question, hostile>$
  - $<idle, move, hostile>$
- A mapping of valid transitions
  - $<idle, attack, hostile> \rightarrow < flee, dead, attack>$
  - $<idle, question, hostile> \rightarrow$ $< flee, answer, lie, abstain, attack>$

Note that there is no entry in the mapping of valid transitions that corresponds to the entry $<idle, move, hostile>$ in the set of required transitions. This indicates that a transition

to any state is acceptable for this input. A solution would conceptually be modeled as shown in figure 3.

Current State	Player Action	Social Relation	New State (Integer representation)
Idle	Attack	Hostile	Dead (10)
Idle	Question	Hostile	Answer (5)
Idle	Move	Hostile	Flee (1)

Fig. 3. Example of agent controller for a limited set of states and actions

This would be encoded as: $10, 5, 1$

F. Evolutionary Algorithm

The evolutionary algorithm used to generate the agent controllers is as follows:

1) Initialize a population P of N candidate solutions. For each agent, a state is randomly selected from the set of valid states mapped to the input tuple of each required transition.
2) Evaluate the fitness of each individual S in P. If any solution S has a fitness of 0, return S and terminate.
3) Select a population P_P of N_P 'parent' solutions using binary tournament selection with tournament size K_p.
4) Create a population P_O of N_O 'offspring' solutions by randomly selecting two 'parent' solutions P_1 and P_2 with replacement from P_P and combining them to create a solution O.
5) Mutate each solution S_O in P_O.
6) Evaluate the fitness of each individual S_O in P_O. If any solution S_O has a fitness of 0, return S_O and terminate.
7) Select a population P_N of N 'survivor' solutions using binary tournament selection on $P \bigcup P_O$with tournament size K_N.
8) Set $P = P_N$
9) Go to 2.

In this experiment the following parameters were used:

- N = 20
- N_P = 20
- K_p = 10
- N_O = 20
- K_N = 10

1) Recombination:

Given two parent solutions P_1 and P_2, a child solution C is generated by recombination as follows:

1) Create C a solution with the same number of agents N_A and states N_S as P_1 and P_2.
2) For each agent A
   a) For each state s
      i) $C.A.s$ = $P_1.A.s$ with 50% probability and $C.A.s$ = $P_2.A.s$ with 50% probability.

2) Mutation:

Each state of each agent of each candidate solution is mutated with mutation rate MR. Mutation randomly selects a state from the set of valid states mapped to the input tuple.

In this experiment $MR = 2\%$.

G. Fitness Function

A minimisation fitness function is employed. Solutions are evaluated by attempting to find all chains of actions that will lead to the completion of the specified victory objective that are possible under the agent configuration represented by the solution. Each distinct chain represents a different way of winning the game. It is desirable to obtain all such chains so that properties of the game can be inferred. For example, if all valid chains are known and in each chain a certain action is used, that action will clearly be used in the game. Alternatively if no chain ever features an action involving a particular agent, it can be seen that the agent is not utilised in the game. Knowing these facts helps evaluate the quality of the game produced by using a candidate solution as the controllers in the game and hence the quality of the candidate solution itself.

Valid chains of actions are determined by searching backwards from the the victory objective in a depth-first manner. Objectives can either be to acquire a fact, acquire an item or attain a state. A conceptual illustration of how this process works for objectives to acquire an item is shown in Figure 4. Objectives for changing an agent's state or learning a fact are handled in a very similar fashion, the only difference is to check whether an action can lead to the desired result in each case. Starting from a condition required for victory, actions are checked to see if they can achieve the objective. Then any-prerequisites necessary to carry out the action are checked by recursively making these pre-requisites into objectives and checking them. A detailed listing of the algorithms is included in Section IX.

Once all the valid chains of actions have been identified, the properties of the chains are evaluated and penalties applied as appropriate. Thus, an optimal solution will score 0.

H. Example

To illustrate how this process works, consider the following scenario. For the sake of brevity, only a subset of the full specification has been shown.

- Victory objective is to obtain the item with $I_{ID} = 1$
- Agent A is neutral towards the player
- Agent A has possession of item $I_{ID} = 1$
- Agent A has transitions $<idle, offerTrade, neutral> \rightarrow acceptTrade$, $<idle, attack, neutral> \rightarrow dead$ (among others)
- Agent B is neutral towards the player
- Agent B has possession of item $I_{ID} = 2$
- Agent B has transitions $<< idle, attack, neutral >> dead >$(among others)

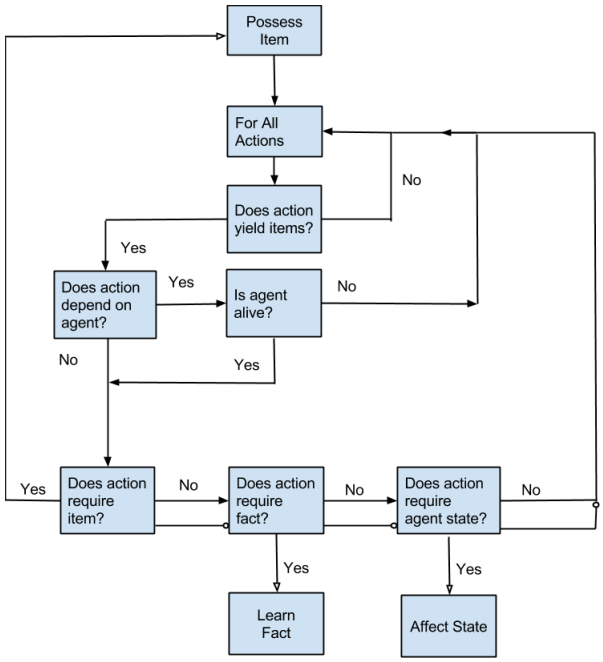

Fig. 4. Illustration of solution checking process

- The player can attack, intimidate, or trade with agents

The list of valid chains of actions to achieve victory would be found as follows:

- Agent A (ID=2) has item O(ID=1)
  - Action $attack$ can result in possession of items
    - A has a transition that can be satisfied by $attack$
      - L_{States} = {}
      - L_{Facts} = {}
      - L_{Items} = {}
      - L_{Joins} = {{attack}}
      - L_{Chains} = {{attack}}
  - Action intimidate cannot result in possession of items
  - Action trade can result in possession of items
    - A has a state transition that can be satisfied by trade
      - L_{States} = {}
      - L_{Facts} = {}
      - L_{Items} = getItemChains(2) = {attack} Perform a search to acquire an item with $I_{ID} = 2$
      - L_{Joins} = {{attack, trade}}
      - L_{Chains} = {{attack},{attack,trade}}

Hence the valid chains of actions to victory are:

1) Attack agent A.
2) Attack agent B. Trade item 2 with agent A for item 1.

IV. Experiment

A. Game Scenario

A simple experiment was devised as a first step in demonstrating the viability of the concept. In the experimental scenario, an 'evil wizard' agent Z has possession of a magic sword. The player's victory objective is to acquire possession of the magic sword. The FSM controlling agent Z is fixed in advance so that the only way to acquire the magic sword item is to attack and defeat Z. Initially the player cannot see Z. The player must obtain the knowledge of how to see Z and also obtain an item that will allow the player to defeat Z. Agents are created that possess the knowledge of how to see Z, a 'Crossbow', the item required to defeat Z and an item that can traded for the 'Crossbow'. In this example, a viable controller set is any set that allows the player to obtain the knowlede of how to see Z and the item required to defeat Z. Desirable properties are for every action the player can perform to appear on at least one path and for every agent to be interacted with in at least one path.

B. Input Specification

- List of Items -
  The items used in the experiment are listed in Table I.

TABLE I
ITEMS FEATURED IN THE EXPERIMENT

Item ID	Item Name
1	Magic Sword
2	Crossbow
3	Gold
4	Invisibility Cloak

- List of Facts -
  The items used in the experiment are listed in Table II.

TABLE II
FACTS FEATURED IN THE EXPERIMENT

Fact ID	Fact Name
1	Z's location
2	How to spot Gold

- List of Agents -
  The items used in the experiment are listed in Table III.

TABLE III
AGENTS FEATURED IN THE EXPERIMENT

Agent ID	Fact	Item	Note
1	N/A	Magic Sword	Z, The Evil Wizard
2	Z's location	N/A	N/A
3	N/A	Crossbow	N/A
4	N/A	Gold	N/A
5	How to spot gold	N/A	N/A

- Item Mappings - This mapping can be observed in Table III.
- Fact Mappings - This mapping can be observed in Table III.
- Trade Mappings -
  The items that agents in the experiment will accept in exchange for items they hold are listed in Table IV.

TABLE IV
TRADE ITEM MAPPING

Accept Item	Give Item
3	2

- Action Descriptions -
  The actions the player can carry out in the experiment are described in Table V.

TABLE V
ACTION DESCRIPTIONS

Action	Valid Start States	Valid Post States	Gains Fact	Gains Item	Requires Fact/Item
Attack	Idle Fight Flee	Dead Slain	No	Yes	-1/-1
Ask	Idle Fight Flee	Answer	Yes	No	-1/-1
Trade	Idle	Accept Trade	No	Yes	-1/-1
Intimidate	Abstain	Answer	Yes	No	-1/-1
Pick Up	N/A	N/A	No	Yes	2/-1

- States -
  - Idle
  - Flee
  - Attack
  - Fight
  - Lie
  - Answer
  - Abstain
  - Accept Trade
  - Refuse Trade
  - Invisible
  - Dead
- Victory Objective - Obtain item 'Magic Sword'.

C. Output Constraints

1) Required Transitions:
In this experiment, transitions are required for all combinations of states, actions and social relations.

2) Valid Transition Mappings:

- $<?, Attack, ?> \rightarrow <Flee, Dead, Attack>$
- $<?, Ask, ?> \rightarrow <Abstain, Answer>$
- $<?, Trade, ?> \rightarrow <AcceptTrade, RefuseTrade>$
- $<?, Intimidate, ?> \rightarrow <Abstain, Answer>$

Note that $<?, attack, ?> \rightarrow <flee, dead, attack>$ denotes that for an agent in any state and in any social group, the valid states that they should transition to upon being attacked by the player are flee, dead or attack.

3) Desired Actions:

- Attack
- Ask
- Trade
- Intimidate

D. Fitness Function

The penalties applied to a solution's fitness are listed in VI.

TABLE VI
OUTPUT CONSTRAINTS

Reason	Penalty
No valid chains found	1000
A specified fact is not used in any chain	50
An agent has exclusive possession of more than one fact	10
An action in the required actions set is not present in any chain	10
An agent is incapable of being involved in any chain	200

V. RESULTS

A. Controllers

In all (1000) experimental runs, valid solutions were discovered. Figure 5 shows selected state transitions from the controllers generated in one run of the experiment. Using these controllers there were two valid paths to win the game.

1) a) Attack agent 4 to gain item Gold
   b) Trade item Gold for agent 3's item Crossbow
   c) Ask agent 2 to learn fact Z's location
   d) Attack agent 1 to obtain item Magic Sword
2) a) Ask agent 5, who abstains
   b) Intimidate agent 5 to learn fact How to spot Gold
   c) Pick up item Gold
   d) Trade item Gold for agent 3's item Crossbow
   e) Ask agent 2 to learn fact Z's location
   f) Attack agent 1 to obtain item Magic Sword

From these paths it can be seen that agents 1-5 are involved in at least one path. Actions 'attack', 'ask', 'intimidate', 'trade' and 'pick up' are all used in at least one path. This shows that as well as being viable, the controller set is optimally desirable for the given constraints.

B. Observed Output

The following capabilities have been demonstrated:

- Controllers can be generated that form a simple but coherent game where victory is possible.
- A developer can specify actions they wish the player to use and the controllers generated will feature at least one valid path to completion that features those actions.
- Multiple valid paths to victory can be generated and detected in a single solution.

C. Performance

No systematic testing has been carried out to measure complexity so far and the emphasis on testing has been on small scale situations so that results can be manually inspected and verified. In these examples, optimal solutions (fitness 0) have been generated at an average time of 271ms on an i5-2500K processor.

Current State	Player Action	Social Relation	Agent 1	Agent 2	Agent 3	Agent 4
Idle	Question	Friendly	Lie	Lie	Answer	Abstain
Idle	Question	Neutral	Answer	Lie	Lie	Abstain
Idle	Question	Hostile	Answer	Answer	Abstain	Answer
Abstain	Intimidate	Friendly	Answer	Flee	Idle	Answer
Abstain	Intimidate	Neutral	Flee	Flee	Idle	Idle
Abstain	Intimidate	Hostile	Idle	Idle	Answer	Answer
Idle	Trade	Friendly	Refuse Trade	Refuse Trade	Accept Trade	Accept Trade
Idle	Trade	Neutral	Accept Trade	Refuse Trade	Accept Trade	Accept Trade
Idle	Trade	Hostile	Refuse Trade	Accept Trade	Refuse Trade	Refuse Trade
Idle	Attack	Friendly	Idle	Flee	Dead	Flee
Idle	Attack	Neutral	Dead	Dead	Flee	Idle
Idle	Attack	Hostile	Idle	Idle	Dead	Idle

Fig. 5. Snapshot of generated agent state transitions

VI. Future Work

In this paper the motivation for the work has been to assess whether this concept has the potential to automate the design of agent controllers for RPG style games. The experiment described in this paper is on a very small scale. To test the validity of the concept in general, experimentation on a scale representative of real games will be required. We are in the process of examining scenarios from the World of Warcraft game to judge the number and type of interactions which will be required.

If our planned larger scale experiments prove successful, work will be carried out to determine whether additional benefits can be gained from this approach such as creating very large and complex FSMs that offer advantages over those that can be feasibly crafted by hand or used to guarantee minimum levels of variety over large populations of agents.

Further work is also required to ascertain exactly how and where the system would most effectively be deployed in the development process. For instance, the system could be deployed as described in this experiment to generate an entire set of agent controllers or it could be used to fine-tune the lower-level controllers in a hierarchical FSM template architecture or to provide bounded variations from a core template.

VII. Conclusions

The experiment described in this paper has shown that it is possible to evolve a set of controllers for agents in a computer game such that the gameplay resultant from using those controllers adheres to desired constraints. It has also been shown that it is possible to quantify certain gameplay properties and evaluate them from a candidate set of controllers. Both these findings have been made in a primitively small and simple scenario and further research is required to ascertain under which circumstances they will hold.

VIII. Acknowledgements

This work was supported by NSERC's Discovery Grant Program under RGPIN 283304-2012 and by a doctoral award from the Dean of the School of Graduate Studies at Memorial University.

IX. Appendix

1) Acquire Item:
For a given item I:

Algorithm 1 Psuedocode for the algorithm to construct a sequence of actions to acquire a given item

INPUT: $A \leftarrow$ agent that has possession of the target item I
INPUT: $actions \leftarrow$ set of all actions the player can perform
for each action a in $actions$ **do**
  if action a can result in acquisition of items **then**
    if A has a transition T that can be satisfied by a **then**
      $L_{STATES} \leftarrow$ a list of of all valid chains of actions that can result in A attaining the pre-requisite state T_{PRE} of T.
      $L_{FACTS} \leftarrow$ a list of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{FACT} necessary to execute T.
      $L_{ITEMS} \leftarrow$ a list of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{ITEM} necessary to execute T.
    end if
    $L_{JOINS} \leftarrow$ a list of all valid chains of actions found by joining a to all the combinations made by combining one chain each from L_{STATES}, L_{FACTS} and L_{ITEMS}.
    $L_{CHAINS} \leftarrow L_{CHAINS} + L_{JOINS}$
  end if
end for
Output L_{CHAINS}

2) Acquire Facts:
For a given fact F:

Algorithm 2 Psuedocode for the algorithm to construct a sequence of actions to acquire a given fact

INPUT: $A \leftarrow$ agent that has possession of the target fact F
INPUT: $actions \leftarrow$ set of all actions the player can perform
for each action a **do**
  if action a can result in acquisition of facts **then**
    if A has a transition T that can be satisfied by a **then**
      $L_{STATES} \leftarrow$ a list of of all valid chains of actions that can result in A attaining the pre-requisite state T_{PRE} of T.
      $L_{FACTS} \leftarrow$ a list of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{FACT} necessary to execute T.
      $L_{ITEMS} \leftarrow$ a list of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{ITEM} necessary to execute T.

end if

$L_{JOINS} \leftarrow$ a list of all valid chains of actions found by joining a to all the combinations made by combining one chain each from L_{STATES}, L_{FACTS} and L_{ITEMS}.

$L_{CHAINS} \leftarrow L_{CHAINS} + L_{JOINS}$

end if

end for

Output L_{CHAINS}

3) Attain State:

For a given agent A and state S:

Algorithm 3 Psuedocode for the algorithm to construct a sequence of actions that results in a given agent being in a given state

$L_{CHAINS} \leftarrow null$

for each transition T that has a post-state S **do**

if If the social relation of T is the same as the relation between the player and A **then**

Create a List L_{FACTS} of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{FACT} necessary to execute T.

Create a List L_{ITEMS} of all valid chains of actions that can result in the player obtaining the pre-requisite fact T_{ITEM} necessary to execute T.

Create a list L_{JOINS} of all valid chains of actions found by joining a to all the combinations made by combining one chain each from L_{STATES}, L_{FACTS} and L_{ITEMS}.

Add L_{JOINS} to L_{CHAINS}.

end if

end for

REFERENCES

[1] N. Afonso and R. Prada, "Agents that relate: Improving the social believability of non-player characters in role-playing games," in *Proceedings of the 7th International Conference on Entertainment Computing*, 2008, pp. 34–45.

[2] S. C. J. Bakkes, P. H. M. Spronck, and H. J. van den Herik, "Opponent modelling for case-based adaptive game ai," *Entertainment Computing*, vol. 1, no. 1, pp. 27–37, January 2009.

[3] D. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in *Proceedings of the 5th International Conference on Intelligent Games and Simulation*, 2004, pp. 36–40.

[4] P. Lankoski and S. Bjork, "Gameplay design patterns for social networks and conflicts," in *Proceedings of the Computer Game Design and Technology Workshop*, 2007, pp. 76–85.

[5] J. Miles and R. Tashakkori, "Improving believability of simulated characters," *Journal of Computing Sciences in Colleges*, vol. 25, pp. 32–39, 2010.

[6] P. G. Patel, N. Carver, and S. Rahimi, "Tuning computer gaming agents using q-learning," in *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2011, p. 583590.

[7] P. Sweetster, D. Johnson, J. Sweetster, and J. Wiles, "Creating engaging artificial characters for games," in *Proceedings of the International Conference on Entertainment Computing*, 2003, pp. 1–8.

[8] C. Fairclough, M. Fagan, B. M. Namee, and P. Cunningham, "Research directions for ai in computer games," in *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*, 2001, pp. 333–344.

[9] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *Computational Intelligence Magazine*, vol. 1, pp. 10–18, 2006.

[10] N. Hocine and G. A. Gouaich, "Agent programming and adaptive serious games: A survey of the state of the art," 2011.

[11] C. Thurau, G. Sagerer, and C. Bauckhage, "Imitation learning at all levels of game-ai," in *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence,Design and Education*, 2004, pp. 402–408.

[12] I. Szita, M. Ponsen, and P. Spronck, "Effective and diverse adaptive game ai," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, pp. 16–27, 2009.

[13] D. Johnson and J. Wiles, "Computer games with intelligence," in *Proceedings of the 10th IEEE International Conference on Fuzzy Systems*. IEEE, 2001, pp. 1355–1358.

[14] M. Katchabaw, D. Elliott, and S. Danton, "Neomancer: An exercise in interdisciplinary academic game development," in *Proceedings of the DiGRA 2005 Conference*, 2005.

[15] S. D. L. Gruenwoldt and M. Katchabaw, "Creating reactive non player character artificial intelligence in modern video games," in *Proceedings of the 2005 GameOn North America Conference*, 2005.

[16] M. K. L. Gruenwoldt and S. Danton, "A realistic reaction system for modern video games," in *Proceedings of the DiGRA 2005 Conference*, 2005.

[17] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 3, 2011.

[18] M. Hendrikx, S. Meijer, J. van der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 9, no. 1, pp. 1–22, 2013.

[19] P. Spronck, M. Ponsen, and E. Postma, "Adaptive game ai with dynamic scripting," in *Machine Learning*. Kluwer, 2006, pp. 217–248.

[20] N. Cole, S. J. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bot," in *Proceedings of the International Congress on Evolutionary Computation*, 2004, pp. 139–145.