An Object Model for Behavioural Planning in a Dynamic Multi-Agent System

[*]Alex Whittaker, Tiziano Riolfo and Neil Rowlands

A object oriented design template is described for the behavioural control of a multi-agent system.

Introduction **3**
Platform 3
Architecture 3
Performance 4
Finite State Automata **4**
Finite State Transition Networks 4
Finite State Transducers 5
Recursive Transition Networks 5
Augmented Transition Networks 5
Implementation **6**
The State Machine 6
Percepts 7
Simple Percepts 8
Compound Percepts 9
Team Percepts 9
Actions 9
Effectors 11
Register Operators 11
Communicators 11
State Changers 11
Application **12**
Actions 12
Percepts 12
States 13
Editor 13
Performance **13**
Conclusions **14**
Acknowledgements **14**
References **15**

[*] Psygnosis Ltd.

Introduction

The template presented is being developed at Psygnosis Camden studio for the game Team Buddies (working title). Broadly speaking, the game presents the player with a team of up to four agents one of which is under their control, the remainder being autonomous but responsive to commands. This team is pitted against up to three other teams, which may be controlled by the computer or by other players across a network.

The game world exists as a three dimensional terrain about which the agents operate. The agents are anthropomorphic and their actions are limited to movement in two dimensions, jumping, carrying crates, firing weapons and driving vehicles. As well as static obstacles within the terrain there are three other features of note:

Crates:	These enter the world in designated zones at a constant rate, they represent the resource with which agents are able to build toys.
Stacking Pads:	By delivering crates to the stacking pads, agents are able to build larger crates which when opened reveal toys whose value is broadly proportional to the number of crates put into the pad.
Toys:	Revealed from the crates, toys can be weapons, vehicles and new team members. Agents need to stack more crates in order to get more powerful toys, which will increase their ability to win the game.

There are also neutral agents within the game world, animals and civilians, with which the player may interact. These must also be controlled with some degree of intelligence, bringing the maximum number of agents in the game world to approximately forty.

The presence of vehicles in the game world means that the agents must be able to use different control systems depending on the vehicle type. Certain toys can give the agents special abilities that should change their behaviour. Furthermore there are several different types of agents with different abilities, strengths and weaknesses.

The player must be able to control the agents in their team, however they will expect them to behave intelligently without orders. Because the control interface must be kept quite simple, agents must be able to interpret player instructions according to their condition and that of their team.

Platform

A major constraint on the game design is the target hardware platform - a Sony PlayStation. Whilst this represents a powerful tool for the manipulation of graphical images, it is not an ideal platform on which to tackle the large search spaces of classical AI. The machine has 2Mb of main memory, of which one might expect to be allocated 500Kb for data, which is manipulated by a 33Mhz processor. If the system were to be compared to a PC, the equivalent computing power would be a 486-generation processor with a high-end graphics card and no floating-point operations.

Architecture

The majority of work involving finite state autonoma (FSA) has been centred on applications to language parsing; we demonstrate the use of FSA to drive intelligent agents using limited memory and processing.

We have implemented a behavioural model using an augmented transition network (ATN) at the highest level, a partial planner to execute some operations such as route planning, and a model for representation of a database of completed and partial plans. The model could be extended to allow a completely implemented partial planner for all actions, however we have so far avoided developing this because of the restrictions of search.

The ATN is driven by percepts from the agents embodied in the game world. Percepts are either member variables (The registers of the ATN) or calculations made within the game world from the agent's perspective, e.g. Distance from agent to nearest crate.

The ATN, partial plan database and percept database are described in data files and generated within an editor that allows the agent behaviour to be described by a designer rather than the programmer. The editor allows us to visualise and manage the complex nature of the finite state machines.

Performance

The system development is ongoing and the scheduled for completion to beta level on the PlayStation platform by summer 1999. We have implemented the ATN on a PC platform and can demonstrate acceptable performance, both in terms of agent intelligence and compute cycles.

The general nature of the solution makes it applicable to a wide range of problems where there are a large number of agents operating under stringent performance constraints.

Finite State Automata

The Finite State Automata (FSA) family represents a class of computing machines that can be used to express a wide range of algorithms in a simple and mathematically concise way. Whilst they have been most widely explored in natural language parsing and generation, they do have a long history in agent control, from Walter's turtle **[Walter 1950]** which is an analogue implementation of a simple FSA, to the layered subsumption architectures of Brooks **[Brooks 1991]**.

If we imagine a network of nodes (states) each connected to one or more others by edges (transitions) the network describes a directed graph. Whilst there can be several transitions between states, each transition has exactly one percept associated with it and if that percept is triggered then the agent will cross that arc and change its state accordingly. In the new state a different set of percepts will control changes to different states.

For example in **Figure 1** if we begin in state *A*, then the state *D* can only be reached through percept *2* or through percept *1* followed by *3* going via state *B* (and not the other way around).

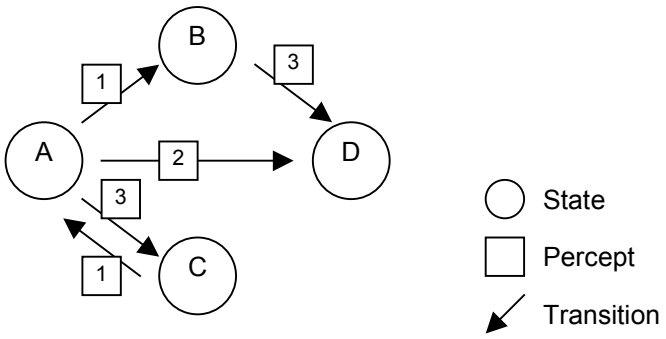

Figure 1: Simple Finite State Transition network

The FSA family extends to finite state transition networks, finite state transducers, recursive transition networks and augmented transition networks.

Finite State Transition Networks

There are two broad groups of finite state transition networks (FSTN), deterministic and non-deterministic. In a deterministic network we are never in any doubt over which arc to take, in a non-deterministic network we can receive two percepts directing us across

different arcs. In this implementation we avoid issues of non-determinism by associating an ordering and strength to each percept, we can then always take the arc dictated by the strongest percept, or where there is a tie, the first in order.

A FSTN can be said to have a limited memory because it knows its own state. For example if the percept *Hungry* is registered it may trigger a transition to *HungryState*; in this new state the agent might have a different set of transitions, all relating to getting food. It could be said to have remembered that it was hungry.

Finite State Transducers

The finite state transducer (FST) is a subclass of FSTN. Each transition of the FSTN is driven by a percept, however it is also possible to use those transitions to drive the agent's actions by attaching an action alongside the percept. The idea of this being a transducer comes from the application of this algorithm in linguistics where a word might be read in and a fact written out by the same transition.

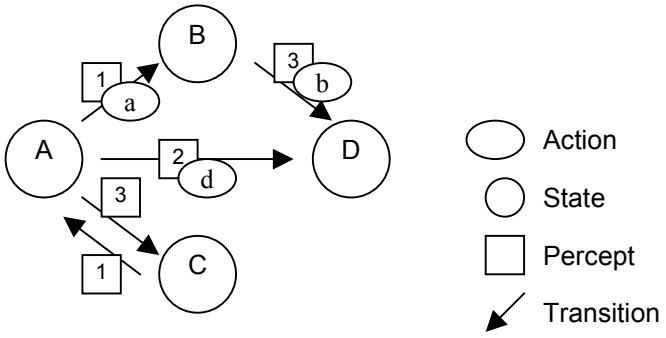

Figure 2: Finite State Transducer

In our implementation we are able to attach an action to a transition alongside a percept, crossing that arc will instigate the action. **Figure 2** shows the simple FSTN in **Figure 1** converted to a FST: Reaching state *D* via percept *1,3* executes actions *a, b* and via percept *2* executes action *d*.

Recursive Transition Networks

The recursive transition network (RTN) is an extension of the FST that is able to push its current state onto a stack and jump to a new state machine. If it crosses a specific exit transition from a state machine then it pops the previous state from the stack and returns to it. As the new state may be a fresh copy of the current state machine, this allows recursive state changes. In linguistics, the recursive machine is able to describe (or parse) a larger range of languages.

The memory of the RTN is extended from the state memory of the FSTN to a higher order, remembering the stack of all successive states.

Augmented Transition Networks

By supplying the state machine with a set of registers we are able to extend the RTN to the more descriptive augmented transition network (ATN). A set of registers is maintained which are accessible from any state, and rather than limited to recording previous states only as in the RTN, these registers may hold arbitrary data.

In our implementation, the registers generally hold pointers to objects within the agent's environment, which can be queried by the percepts and actions. For example an action might select another agent from the environment and put it into the Target register. Consequently a percept measuring the distance to the agent in the Target register would then change.

The ATN extends further the range of languages that can be parsed or generated by the network as was demonstrated to great effect in the ELIZA program **[Weizenbaum 1965]**.

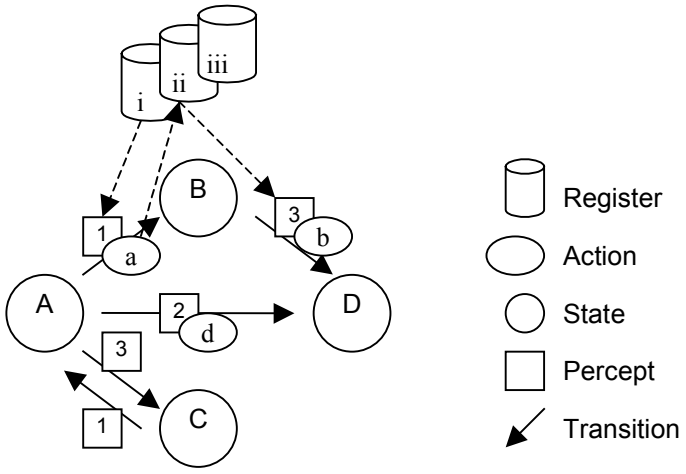

Figure 3: Augmented Transition Network

Extending the FST of **Figure 2** the ATN represented in **Figure 3** shows how the action *a* modifies the register *ii* in the transition from *A* to *B*, which is in turn read by percept *3* in the transition from state *B* to *D*.

Implementation

Agents are controlled by an ATN that is driven by percepts and drives actions. Actions are presented in a hierarchical format allowing compound actions to be built up out of a base set of axioms. Percepts measure the environment and may also be compounded together using functions such as min and max.

The State Machine

We have implemented an ATN architecture for control of individual agents within the world. Each agent is a container for the *GPlanner* class containing the current state, and drives state transitions reading percepts from *GSensor*, and actions written to *GPlan*. **Figure 4** shows the *GPlanner* class contained within the *GAgentObject* class that describes all intelligent agents in the game world.

The state machine is described within the *GSettingState* class, contained in the static class *GSettingsCollection*, a generic container into which data files are read. Each state has a set of three arrays of integers, each with a number of members corresponding to the number of arcs leaving that state. The three integer arrays correspond to the ID of the state to which the transition leads, the percept that drives the transition and the action that is initiated by that transition.

There is a separate state network for each type of agent within the world, and each is held in a static class rather than within the planner. This is because all agents of a given type are able to use the same instance of the appropriate network, as it contains no agent specific information.

The planner is a container for the *GPlan* class, which holds each of the registers and the hierarchy of actions. There are Approximately twenty registers for each agent and as these are specific to the agent they must be contained within the planner. Examples of the registers are included in **Figure 4** e.g. *pTarget* is a pointer to the agent's current target.

In any given state, there will be a set of transitions to other states, each with a percept attached to it. Percepts are contained within the static *GSensor* class, to which calls are made through the *TestPercept()* function. By passing a reference to the agent's *GPlan* to

the function (and hence all of its registers) we are able to use a single set of percepts rather than one for each agent.

Actions may also be associated with a state transition and because these take the form of an n-ary tree, the plan needs only to point to the highest member. If a percept causes a state transition that dictates an action, then that action will replace the current action.

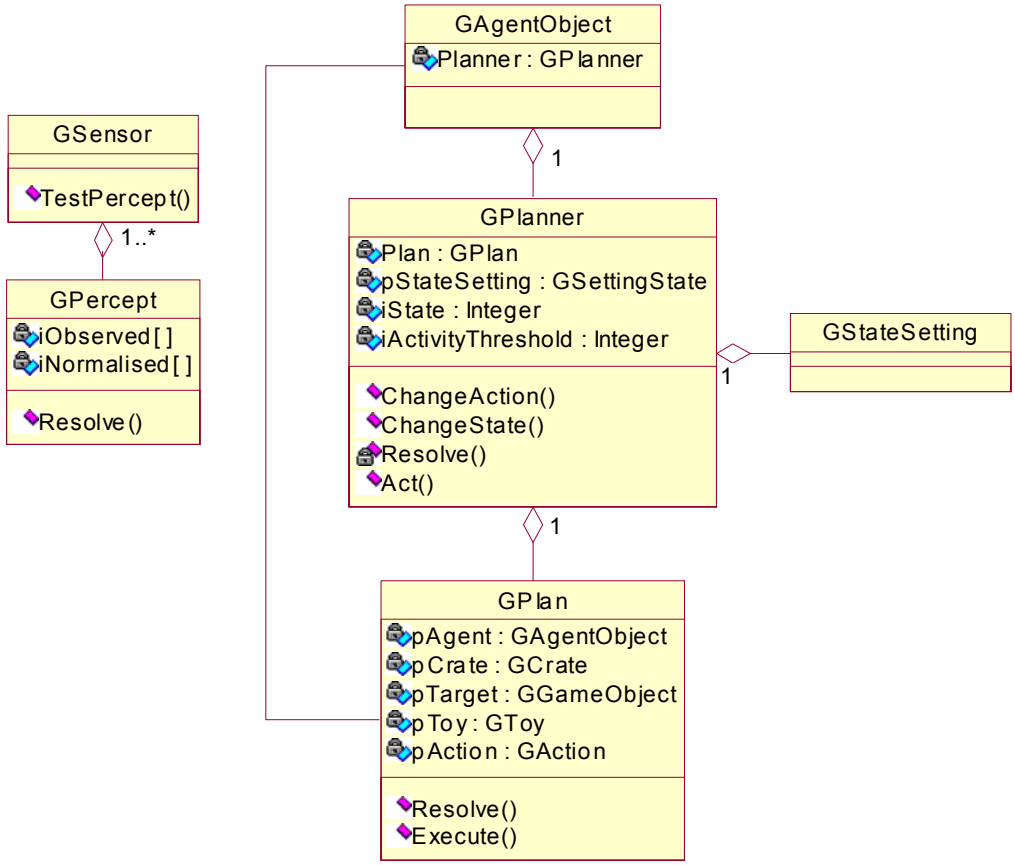

Figure 4: Inheritance of the Planner Class

Percepts

There are three types of percept all contained within the static *GSensor* class:

Simple percepts	normalise an observed value and return an integer between zero and 100
Compound percepts	operate a function across a set of component percepts (simple, compound or team)
Team percepts	operate a function on a single percept (simple or compound) measured for each team member

Figure 5 shows the logical structure of the *GSensor* class containing the percept hierarchy. The GCompoundPercept class can contain one (though generally two) or more percepts, the GTeamPercept class contain just one percept. Note that the containment described for the compound and team percepts is by an integer reference to the ID number of the percept rather than a memory pointer.

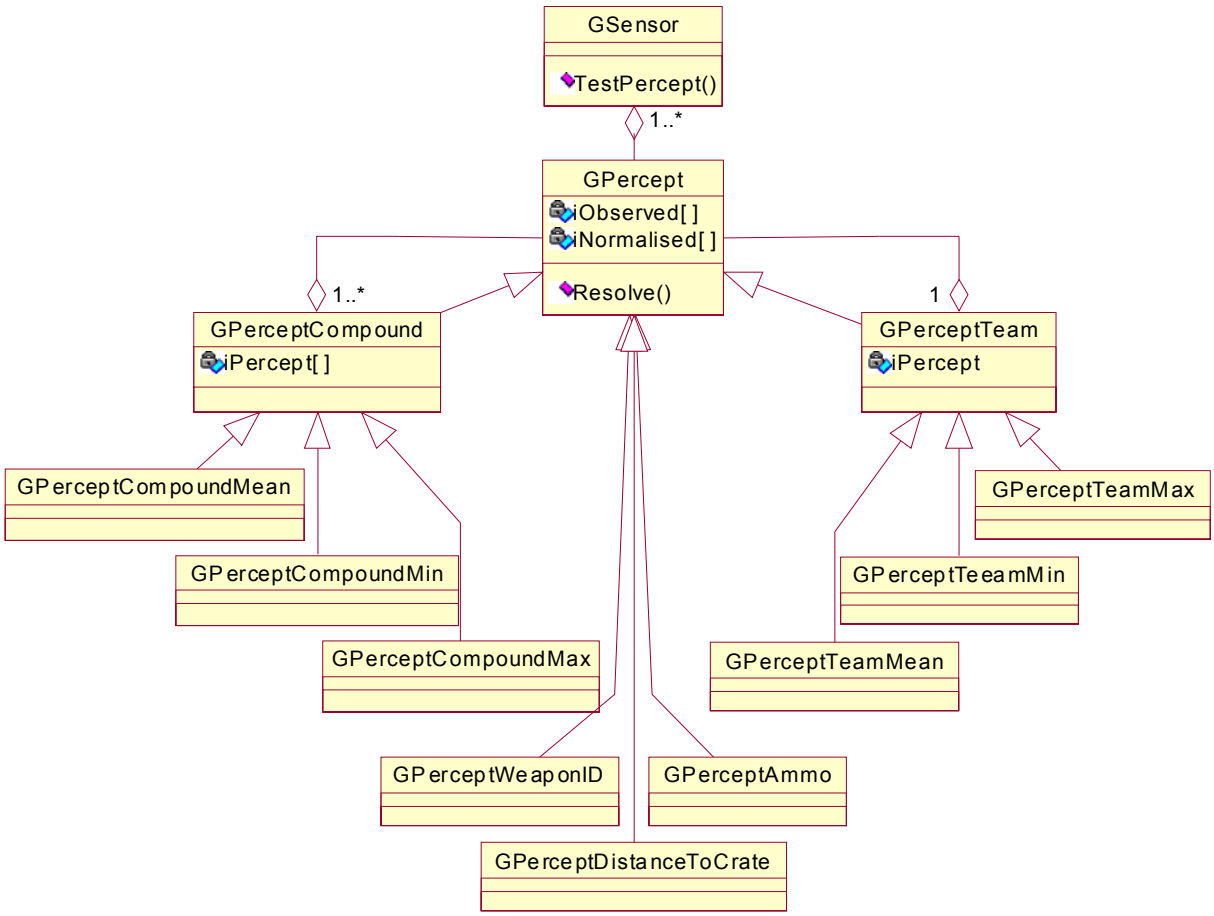

Figure 5: GSensor containment of the GPercept class

Simple Percepts

The basic *GPercept* class reads an observable value, either from the environment external to the agent or from its internal registers, and normalises that value between 0 and 100 according to an associated graph. For example **Figure 6** shows how the percept *LowAmmo* reads the amount of ammunition carried by the agent and normalises to maximise the return value when the agent has very low ammo giving a sigmoid response, with anything higher than 100 returning zero.

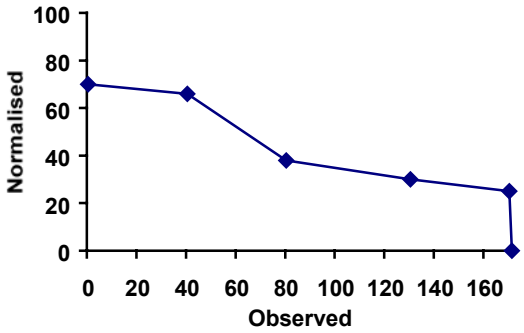

Figure 6: The LowAmmo percept normalisation curve

Compound Percepts

The *GPercept* class has been extended to allow compound operators to act on sets of percepts. There are two types; the *GCompoundPercept* class operates a function across the normalised results of a set of percepts (compound, simple or team). The *GTeamPercept* class operates a function across the normalised result of a percept (simple or compound) for a set of agents.

There are currently four *GCompoundPercept* functions operating on the normalised scores of the component percepts. The first three operate on two or more components, the GreaterThan operator acts on two:

GCompoundMin	The lowest of the normalised component percept scores
GCompoundMax	The highest of the normalised component percept scores
GCompoundMean	Average of the normalised component percept scores
GCompoundGreaterThan	Difference between the first and second component percept score if greater than zero, otherwise zero

The compound percepts allow us to create complex and intelligent responses to stimuli. For example, in an explore state, we wish to limit the agent's inquisitiveness by its distance from the base, however that distance can be higher if the agent has a good weapon. The *Adventurous* percept could measure a *GCompoundMean* operator over the percepts *NearToBase, HighArmed* and *HighAmmo*. The agent would become more adventurous with a good weapon and lots of ammunition.

Team Percepts

There are three *GTeamPercept* operators corresponding to the compound operators (the greater than operation is binary and would fail on teams of more than two agents):

GTeamMin	The lowest of the normalised team percept scores
GTeamMax	The highest of the normalised team percept scores
GTeamMean	Average of the normalised team percept scores

The *GTeamPercept* class allows us to make state transitions dependent on percepts measured across the team. Because agents are able to influence the states of other team members, we are able to drive team strategies from the state machine. For example the percept *TeamLookingForTrouble* could use the *GTeamMin* operator on the compound percept *Adventurous*, to decide whether the to initiate an *Assault* strategy.

Actions

Actions are divided into two types, compound and axiomatic. The axiomatic actions are described in code and they can be referenced by their ID number. The compound actions are data driven, and may reference other compound actions but ultimately resolve down to an ordered list of axioms. The base class for an action is *GTreeNode* shown in **Figure 7**. Note that the reference is bi-directional, the parent can refer to the child, the previous to the next and vice-versa. In practice, multiple actions are linked together to form a tree with one action at the top (for which *pNext* and *pPrevious* are both NULL) and an increasing number of actions at each child level.

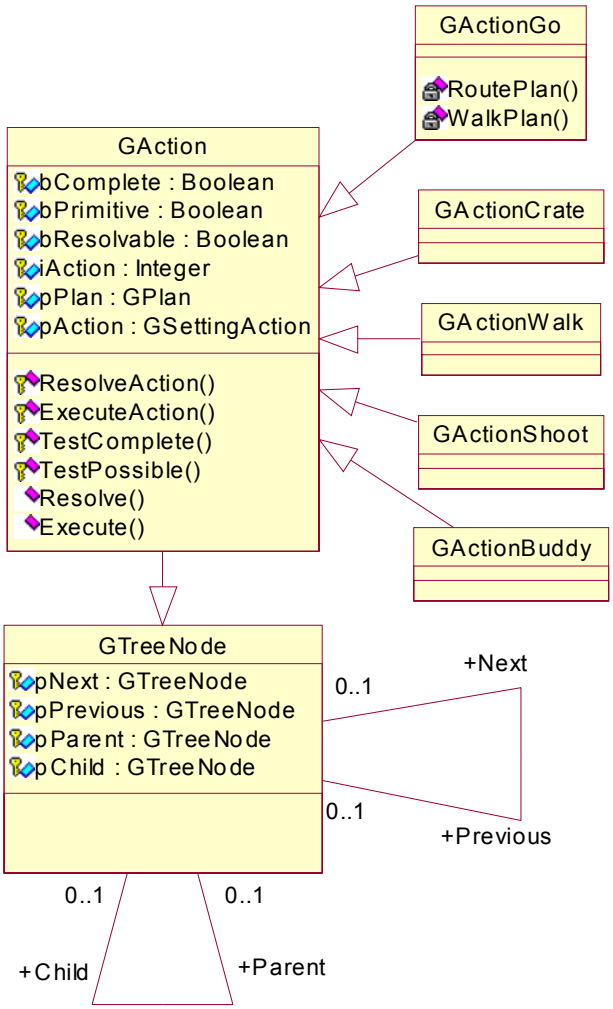

Figure 7: The GAction class structure

There is currently a base set of approximately 160 axiomatic actions, and 1000 compound actions, which are built from compound and axiomatic actions. This database of actions provides us with a set of partial plans, which opens up the possibility of using a partial planner to generate complete plans **[Sacerdoti 1974]**.

The *GAction* class resolves in to a collection of descendants that are classified loosely by function. **Figure 7** gives a few examples of the hierarchy.

Human-controlled agents are controlled via an interface device such as a joystick or joypad (the effector), autonomous agents are controlled by a virtual equivalent, and *effective* actions generate virtual key-presses on that device. Actions are called through one of two functions dependent on whether the *bResolvable* flag is set. Both the *Resolve*() and *Execute()* functions will test that the preconditions are met with *TestPossible()* and that the post-conditions are not yet satisfied with *TestComplete()*.

Provided that the conditions are met, *Resolve()* will then either call the virtual function *ResolveAction()* for the few primitive resolvable actions (such as route planning) or the *ResolveCompound()* function to resolve an action into its component actions as described by the partial plan database. Similarly the *Execute()* function will call the virtual *ExecuteAction()* function for the axiomatic action class if it can be resolved no further.

Actions are sequential and are decomposed to components from top to bottom, left to right, depth first to the first axiom, which is then executed. When its post-conditions are met *Execute()* returns true, it is marked complete and the next axiom is searched for.

There are four kinds of axiomatic action:

Effectors	move the agent within its domain
Register operators	manipulate values within the *GPlan* registers
Communicators	pass messages to other agents
State changers	change the agent's state on execution

Effectors

This group of actions translates an axiomatic action into effective key-presses via the virtual controller. These include those actions relating to movement, attacking, and manipulating crates.

There is a small group of actions relating to route planning which are resolved within the code, ultimately down to the effective directional key-presses, rather than as described by the action database. The route-planning algorithm uses a partial planner to navigate around static obstacles and terrain features within the game, and demonstrates the possibility of adopting a partial planner in all resolution steps.

In order to accommodate route planning within the action architecture we have a set of specialised functions called from *ResolveAction()* within the *GActionGo* class:

1. The action *GoTo* directs the agent to a position on the map and is set as resolvable.
2. Calls to *GActionGoTo::ResolveAction()* are passed to the *RoutePlan()* function.
3. *RoutePlan()* draws a vector between the agents position and the goal location and test for collision with any objects.
4. If the vector collides with an object then the agent perceives the leftmost and rightmost vertices of the obstacle as its limits.
5. One of the two limit vertices is selected as a new intermediate goal and the *GActionGoTo* action resolves into two *GActionGoTo* sub-actions that are again resolved.
6. If no object intercedes between agent and goal then the GActionGoTo action resolves into a *GActionWalk* action.
7. *GActionWalk* actions resolve to relevant key-presses.

Register Operators

Registers are accessed by the percepts to alter the agent's behaviour. The register operators are able to modify the register contents. For example the axiom *SelectNearestCrate* searches the immediate environment for a crate which is written to the crate register.

Communicators

Whilst the *GTeamPercept* class of percepts allows a degree of indirect communication, these actions offer explicit communication between team members. In order to achieve team co-operation agents are able to communicate with each other by issuing orders to team members. Orders pass from the sender to a communication layer and from there after the addition of an amount of noise proportional to the range, to the target.

The order comprises an action identifier, a priority and a reference to the sender. The receiving agent can decide whether to obey the order based on the action, its priority, its sender and the agent's current activity threshold. Actions can be channelled to a specific team member or to the entire team.

State Changers

When a state change action is executed, the agent will change its current state to the state dictated by the action regardless of whether a state transition exists. These actions are not generally included in any compound actions but are communicated from one agent to another as orders.

The communication and state change axioms allow us to build strategies involving several agents in a team. For example one agent can issue orders to all the other team members to go to a point and enter an ambush state where they will do nothing until a target agent comes into range. The agent issuing orders can then instigate a feint attack and hope to draw enemy agents into the ambush.

Application

We include some examples of the pattern implementations from game.

Actions

Most selection actions, associated with the retrieval of desired information such as agent ID, are axiomatic and feature heavily throughout the state machine. They are used to draw the agent's attention toward a specific target.

For example, the compound action *GoAttackTarget* concatenates the actions *MoveWithinRangeOfTarget*, *FaceTarget* and *ShootTarget* and is appended to the action *SelectNearestEnemy* to create *GoAttackNearestEnemy* as seen in **Figure 8**.

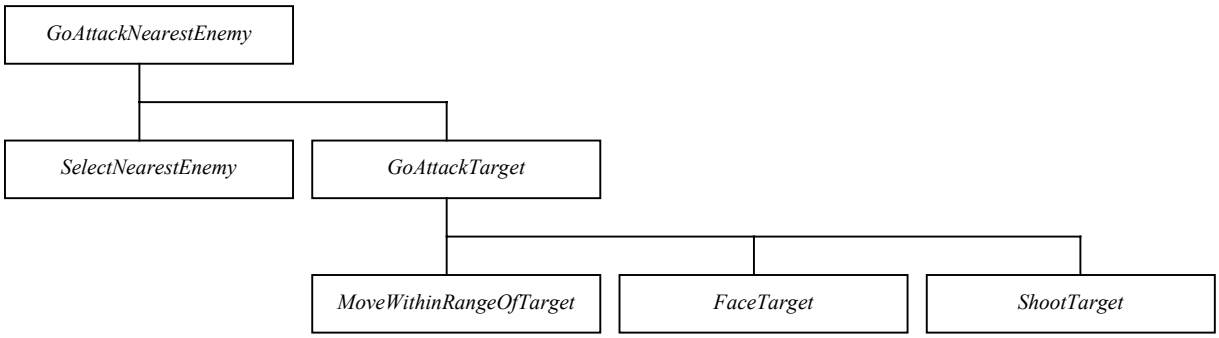

Figure 8: Decomposition of the action *GoAttackNearestEnemy*

The axiomatic and (primitive resolvable) component actions are:

SelectNearestEnemy	Select a target and store the agent ID
MoveWithinRangeOfTarget	Move within weapon range of the target
FaceTarget	Face the target
ShootTarget	Attack the target

Strategy actions pass orders to all team members to act in a coherent manner. For example the *GPileIn* strategy action selects an enemy target and tells all agents to copy this target into their register and then go to the *PileIn* state, from which the agents emerge only when that agent is destroyed:

SelectNearestEnemy	Select a target and store the agent ID
TellTeamSelectMyTarget	Instruct all agents to copy the target from my target register to theirs
TellAllChangeStatePileIn	Tell agents to jump to the *PileIn* state

Percepts

A frequently used axiomatic percept is *EnemyNear*, based on the distance between the agent calling the function and the nearest enemy agent. *EnemyNear* obviously returns a high value when the enemy is near and a low or zero value beyond an arbitrary distance.

A simple yet powerful compound percept uses the *CompoundMin* operator to combine *EnemyNear* and the percept *NoTarget* (which returns a maximum when there is no target in the register). Effectively *NoTarget* acts as a switch on *EnemyNear*, if we have a target, then we're not interested in the enemy's proximity.

Team percepts perform the same operations as compound percepts, using the values of all the agents in a team, and become useful when considering the team as a unit. The building and resource management cycles are affected by the team's current attributes. Ideally we want all team members to have a similarly powerful weapon, or at least that every agent has some kind of weapon. Once sufficient crates have been stacked to build a weapon, the building agent considers whether to open the crates and collect the weapon or add more crates and upgrade it.

A decision to upgrade a current weapon to a stronger weapon is determined by the weakest weapon possessed by all team members. The percept used is *TeamMin(WellArmed)*. Essentially, If the weapon in the currently completed crate configuration is 'better' than that of the weakest member in the team, then the upgrade request is disallowed, the crate is opened and the pad made available, giving weakest member access to request an upgrade.

States

In order to produce behavioural quirks in the attempt to make interesting characters, certain basic states (classed as instinctive) occasionally test for rare circumstances that lead to particular states where 'normal' behaviour doesn't apply. Agents will remain in this state (or state cycle involving several states) until exit conditions are met.

In the case of the state *Hero,* the agent is required to attack all he can see. Using the percept *CompoundMin(NoTargetEnemyNear)* mentioned above, if he has a target then *NoTarget* returns a minimum and consequently *EnemyNear* is disabled (i.e. we already have a target – the enemy's distance to us is unimportant). While it has a target it will continuously attack, however, if the agent destroys the target, its target register is cleared (*NoTarget* returns a maximum) so now any enemies nearby can be detected and selected as new targets if the *EnemyNear* percept is strong enough. The exit condition in this state occurs when there are no further enemies nearby, and the agent returns to the default state.

Editor

The axiomatic actions and percepts are used to construct a state table using a purpose built editor. The editor provides a graphical representation of the state machine, the percept normalisation graph and the partial plan action hierarchy. The data is exported in the simple text format required by the game allowing AI editors to simply generate and test new architectures.

Performance

Whilst we are able to make objective measures of performance on the PlayStation these only acquire meaning when compared to the performance of other similar algorithms. It has not been possible for us to make this comparison due to the inaccessibility of commercial AI implementations within game code, and so we are reduced in the main to subjective commentary.

We are however, able to compare the performance of the algorithm against the requirements of the game, and show that it has satisfied them entirely. The comparison breaks down into three areas, processing speed, apparent intelligence and access to the game designers.

For speed, the principal concern is the impact of the AI processing time on the screen drawing time (frame rate). We have measured the mean processing time using the Playstation performance analyser and presented the results in Table 1.

The timings are measured in raster lines, there are 256 raster lines per frame with the game running at 50 frames per second at maximum speed. The measurements are taken for seven agents, split between times for graphics code – the time to write the images to screen and game code - the time to manipulate the game objects without drawing them to screen. The AI code results show the time taken by one agent to act

within the environment, with 256 raster lines taking 0.02 seconds. This means that each agent takes an average of 3.125 milliseconds or 1.38% of the processing time.

The peak values for agent processing time represent complex route planning actions and are infrequent. Whilst it is clear that the game could not sustain seven agents running at this peak performance, it is known that these peaks are very short and unlikely to coincide with significant frequency.

Table 1: Game timings in raster lines for seven agents

	Graphics code per frame	Game code per frame	AI code per frame	Total code per frame	AI code per agent
Minimum	136	41	7	184	1
Maximum	261	204	98	563	74
Mean	171	91	28	290	4

The agent's intelligence is best described in their ability to win the game, and this is dependent to a large part on the foresight of the game designer editing the transition network. At the current development stage agents are able to play against other autonomous agents and human opponents in an apparently intelligent manner and consistently win.

For the game designers, the ability to describe an agent's intelligence directly through the manipulation of a database is unprecedented. Modifications can be implemented and applied to agent behaviour within five minutes and this along with extensive state machine debugging tools makes for a very rapid development cycle.

Conclusions

Implementation of the model onto the PlayStation architecture has generated few problems, which have been easily overcome. Once the axiomatic actions and percepts are provided, the nature of the ATN architecture means that agent behaviours can be rapidly prototyped, tested and implemented onto the Playstation platform.

It should be stressed that the agents act from an embodied position within the game world, at no time do the percepts measure anything that the player could not measure. Also the agents' actions are effected through a virtual controller, identical to the players and are similarly limited. Whilst this is the norm for mobile agents, it is novel for agents within a game environment.

The variety and appropriateness of the agents' behaviour coupled with the speed of reaction contributes to player's perception of playing against an intelligent and unpredictable opponent. Added to this, the co-operative nature of the strategic actions lends a sense of underlying goal-directed behaviour. These aspects are critical in lending an immersive quality to the game.

Whilst the performance of the implementation can only be measured subjectively, it has consistently surpassed our expectations and we believe it competes strongly with others in the field.

Acknowledgements

This work was made possible through the efforts of the Buddies Team who are:

Tom Beaumont
Peter Hodges
Dave Sullivan
Tim Dann
James Baker
Miguel Melo
Alex Whittaker
Neil Rowlands
Kevin Pim
Nicola Cavalla
Tiziano Riolfo
Analou Marsh
Richard Holdsworth
Dominic Clark
Noel Flores-Watson
Chris Petts
Dan Leslie
Antonio Miscellaneo
Ainsley Fernando
Graem Monk
Steve Oldacre
Kym Seligman

References

Brooks, R.A. (1991) Intelligence without representation. *Artificial Intelligence* **47** 139-159

Fikes, R.E. and Nilsson, N.J. (1971) STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* **2** 189-208

Sacerdoti, E.D. (1974) Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* **5(2)** 115-135

Walter, G. (1950) An imitation of life. *Scientific American* **182** 42-45

Weizenbaum, J. (1965) ELIZA - a computer program for the study of natural language communication between man and machine. *CACM* **9(1)** 36-45