Agent 99: Implementing a simple card game using agents

Sidney Fels, Kenji Mase, Tameyuki Etani,
Armin Bruderlin, Silvio Esser

ATR Media Integration & Communications Research Laboratories
Seika-cho, Soraku-gun, Kyoto 619-02 JAPAN
81-774-95-1440(phone), 81-774-95-1408(fax)
e-mail: {fels, mase, etani, armin, esser}@mic.atr.co.jp

Abstract

This paper describes progress in defining a computer agent architecture to provide a conceptual framework for agent development. In our nomenclature, agent refers to an autonomous computer program which performs some task or tasks on behalf of an entity, communicates asynchronously with other entities in the world and responds asynchronously to events internal to the agent or external to it in the world. Entities can be people, other agents, standard computer programs or devices. These agents can interact with other entities cooperatively or antagonistically. Our agent architecture centres around a 5 level hierarchical structure where each layer provides separate functional levels based on role abstraction. The five layers are: 1. motivation and needs, 2. policy/planning, 3. behaviour, 4. action/sensation and 5. action/sensor units. Additionally, each agent has a knowledge monitor which provides a communication link between layers in an asynchronous manner. The knowledge monitor is based on a client/server model and consists of a knowledge base, an activity manager and a resource manager. The novel feature of the knowledge monitor is that the activity manager allows each client to specify *perceptual filters*. A perceptual filter is a boolean expression of concurrent activities in the agent and events which activate a function in the agent whenever the expression is true. These filters trigger asynchronously, freeing the agent to do other tasks at the same time if necessary. The same knowledge monitor structure is used to maintain the world where the agents live; in this case the clients are the agents and filters trigger on concurrent activities and events in the world. We have started using the agent architecture to create agents to play a simple card game called 99. This game requires agents to cooperate in turn-taking behaviour as well as allowing players to leave the game when they lose. Further, the agents compete to win the game. Current progress has been encouraging and we plan to apply the structure to increasingly complex tasks including, tour guides, agents for complex games, world wide web searches, process control, job-shop management and file management.

1 Introduction

The term *agent* implies a wide variety of computer program qualities including: autonomy, goal-oriented, collaborative, flexible, self-starting, temporal continuity, personality, communicability, adaptability and mobility [4]. The implication has allowed the term agent to apply to many different types of systems. For this reason, the term has lost the ability to label a specific behaviour exhibited by computer programs and needs qualification. In this paper, the term *agent* refers to an autonomous computer program which performs some task or tasks on behalf of a person, communicates asynchronously with other agents in the world and responds asynchronously to events in the world. The term autonomous computer program implies that the program can run on any machine without intervention from a person. The asynchronous nature of the agent implies that it responds to events which occur as-they-happen rather than requiring a global synchronization signal. Of course, this does not mean that synchronous behaviour cannot be implemented by asynchronous agents, rather, synchronization must be made explicit where it is needed.

In this paper, we define an asynchronous-hierarchical agent architecture called A-HA (shown in figure 1)‘. We use A-HA to implement agent card players in a simple card game called 99 to demonstrate its feasibility. This game requires *take-turn* cooperation between players as well as some relatively simple reasoning to play well. While simple, the game has enough complexity to illustrate how the asynchronous agents interact as well as how the multi-level hierarchy works.

This paper is divided into three main sections. Section 2 describes the game of 99 to provide a concrete example to discuss A-HA. The rules of the game provide the context of the world for the agents. Other applications may be considered as different “rules” of a “game”. In section 3, A-HA is described as an abstraction. A-HA has two major subdivisions with respect to its implementation. These are the knowledge monitor and the multi-level hierarchy which are described in sections 3.1 and 3.2. The knowledge monitor provides

the foundation for the asynchronous behaviour of the agents. Within the knowledge monitor is an activity manager mechanism for agents to register *perceptual filters* to watch for events (and event expressions) to occur in the world. When these filters are activated the activity manager can notify the agent. Other parts of the knowledge monitor include a knowledge base and a resource manager. In section 3.2, the multi-level hierarchy, consisting of: motivation/needs, policy/planning, behaviour, actions/sensations and action/sensor units is described in the context of the game of 99. The perception of the game world and the responses produced by agents playing the game are used to provide concrete examples of how the hierarchy works. The hierarchy is important as it allows a clean separation between different functional roles within an agent. This separation aids creating powerful, complex agents and implementing learning algorithms tailored for different functions within an agent. Further, the separation should benefit the agentee[1] and agent designers by providing a conceptual framework for interacting and building an agent. The last section provides some conclusions.

Card	Value	Notes
Ace (A)	1	
2	2	
3	3	
4	0	reverses order of play
5	5	
6	6	
7	7	
8	8	
9	0	keeps the current total the same
10	-10	decreases the value of the current total (only to 0)
J	10	
Q	10	
K	99	forces the current total to be exactly 99

Table 1: Value of Cards in 99. In the game of 99, 4's, 9's, 10's and K's are special cards. Typically, players do not play these cards until the total of the deck grows to 99. Suits have no special role in this game.

2 The Game of 99

The game of 99 is played with 2 to 8 players. The object of the game is to be the last player left in the game. A player is out of the game when he or she must play a card which will cause the cumulative total of all discarded cards to exceed 99. Each player starts the game with 3 cards. The player to the dealer's left discards a card onto the discard pile. The value of that card is added to the total which begins at 0. The value of each of the cards is listed in table 1. The player then picks a new card from the deck. For example, if the first player plays an 8 of spades the total will increase by 8. Play continues clockwise with each player playing a card which is added to the total. Eventually, the total will reach 99. If a player cannot play a card from their hand without causing the total to exceed 99 they are out of the game. If you are the last player left in the game during your turn you win. There are four special cards numbers in the deck which do not increase the value of the current total thereby allowing a player to continue in the game.

Table 1 shows the value of each of the cards. There are 4 *special* card numbers; "4's", "9's", "10's" and "Kings" ("K"). Fours have a value of 0 and cause play to reverse (if there are only two players then a four is just worth 0 points and play goes to the other player). Nines have 0 value. Tens subtract 10 from the current total down to 0 (i.e. negative values are the same as 0). "Kings" force the total to be exactly 99 regardless of the current total.

There are five general strategies used in the game of 99. During a player's turn they choose the next card to play according to a subset of the following strategies:

1. play the highest card possible
2. keep "4's", "9's", "10's" and "K's" until the current total reaches 99; called safe play
3. play a "K" as soon as you can (the reasoning behind this strategy is that it allows less time for other players to accumulate any special cards), this is called risky play.
4. play a "K" as soon as you have another special card (or two); this is moderate play
5. "10's" are always the worst choice to play and should be reserved for times when it is the only card you can play (the reasoning here is that subtracting 10 from the current total allows other players to continue in the game even if they do not have any special cards)

When a card is played the player should say the new total; however, each player is responsible to keep track of the current total to make sure the player playing a card is adding correctly (which is often not the case when that player has been drinking)! A player is also responsible for announcing when they are out of the game.

This game provides a good test-bed for demonstrating A-HAs. Players are implemented by A-HAs and/or are actually people. Agents in this environment must be able to respond asynchronously to the play of the game. Agents

[1] An agentee is the person for whom the agent is doing the task.

must also reason about their hands and decide which card to play. Players typically want to win the game but have different levels of risk taking. Depending upon the motivation provided to the agent by the agentee it should alter its play. Finally, agents participating in the game must communicate with the other players.

3 The A-HA System

A block diagram of an asynchronous hierarchical agent (A-HA) is shown in figure 1. There are two fundamental components in the system. First, the multi-level hierarchical structure of the system is comprised of motivation/needs, policy/planning, behaviour, action/sensation and action/sensor units. Each level corresponds to a separate level of functional behaviour inside the agent. Each level communicates (through the knowledge monitor) to its neighbours. Typically, for responding to events which occur in the world information flows from the top of the hierarchy to the bottom; eventually reaching the action units which can act on the world directly. When the agent is trying to understand and perceive the world, information flows from the bottom of the hierarchy to the top. The dotted connections indicate logical data-flow. All real data flow occurs through the knowledge monitor, but it is the responsibility of the designer to adher to the logical flow. Each level and the hierarchy is described in detail in section 3.2. The second component is called the *knowledge monitor* and is made up of three parts; the activity manager, the knowledge base and the resource manager. This component provides the foundation upon which the multi-level hierarchy rests. It is responsible for asynchronous activity management, data storage and coordinated access to shared resources. As seen in the figure, the knowledge monitor structure is also used to create a knowledge monitor for the world. This knowledge monitor is used *stand-alone* to maintain the world state and keep track of events which occur in it.

The next section describes how the knowledge monitor system works. The knowledge monitor is described first since it provides the foundation for the agent hierarchy. The agent hierarchy is described second since examples of implementing various levels of the hierarchy will be given with respect to the implementation using the knowledge monitor.

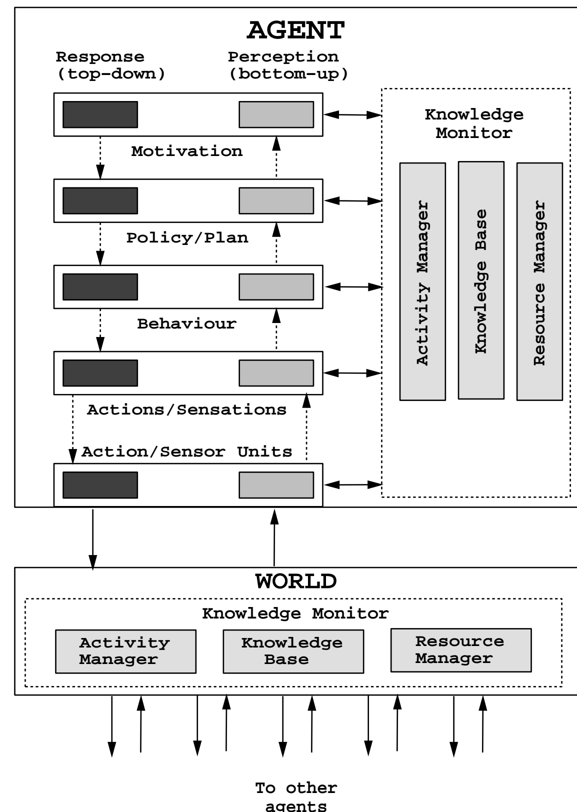

Figure 1: Block diagram of a single agent. Notice that perception is generally a bottom-up process while responses are top-down. All communication between layers is managed by the activity manager and is occurs between adjacent layers only. Further, only the action/sensor units layer communicates with the world. Communication with other agents is through the world (using action units).

3.1 Knowledge Monitor

The knowledge monitor is made up of these three sub-parts:

1. activity manager
2. knowledge base (K.B.)

3. resource manager

3.1.1 The Activity Manager

The activity manager keeps track of three main types of information; first, all the activities which are concurrently happening, second, events which occur and third, a list of all the perceptual filters watching for events and activities to occur[2]. Figure 2 shows a block diagram of the activity manager.

The activity manager is implemented using a client/server model. The activity manager is an [incr tcl] [1] object. It uses one list to keep track of all activities which are occurring and another list containing all the filters watching for events (or event expressions). Additionally, it inherits a base class called server which allows communication to the activity manager via sockets so that multiple clients can access the activity manager asynchronously. The following methods are the main ones for the activity manager:

- registering activities
  1. start
  2. stop
- manipulating filters
  1. addFilter
  2. removeFilter
  3. activateFilter
  4. deActivateFilter

The **start** and **stop** methods allow a client to tell the event manager that some activity is starting or stopping. When the activity is started it is added to the currently running list. If it is stopped, it is removed from the currently running list.

A filter allows a client to ask the activity manager to watch for a specific combination of starting, running and stopping activity conditions. When *all* the conditions specified in the filter are true then the filter triggers a command associated with the filter. Once added, a filter may be activated (default), de-activated, or removed. A filter is specified by a four-tuple where the first three members are the trigger conditions and the fourth is the command to execute. It has the following form:

```
{start activity} {activity running expr.}\
    {stop activity} {command to execute}
```

If either the start, running activity or stop conditions are unspecified they are assumed to be true. For example, a client called **green** may add the following filter (we are assuming there is an activity manager called AM):

```
AM addFilter {} {![NOW GAME_OVER]}       \
   {RED_PLAYING} {green takeTurn}
```

This filter says that when the game is *NOT* over and the activity RED_PLAYING stops that the command *green takeTurn* should be executed. This filter is typically added because **green** plays after **red** when the game is currently running[3]. When a filter is added the activity manager assigns a unique identifier to it and returns the identifier to the client; in this case **green**. The identifier can be used by the client to reference the filter subsequently. The *removeFilter*, *activateFilter*, and the *deActivateFilter* methods allow the user to manipulate the filter using the identifiers. Notice that deactivating a filter is functionally the same as removing it with the main difference coming from speed tradeoffs.

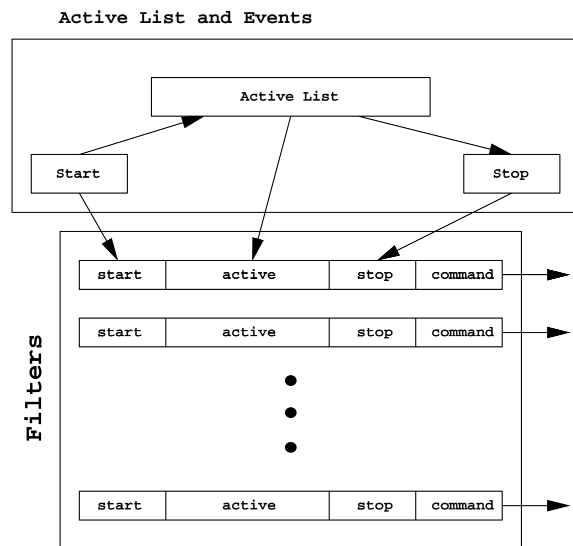

Figure 2: Block diagram of the activity manager: the activity manager keeps track of all activities that are running. All activities are initiated with a start event and terminated with a stop event. Whenever a start or a stop event occurs all the active filters are checked to see if they trigger. If they do, then the associated filter command is executed (usually on the client that added the filter).

There are several points to notice about the activity manager mechanism.

1. There are only two different operations which can specify the status of an activity; it either

[2] A filter also contains a description of what to do when the filter is activated.

[3] Notice, the activity GAME_IN_PROGRESS could be used instead of ![NOW GAME_OVER]. The actual logic of the activity names is determined by the programmer.

starts or stops. Also, starting and stopping are mutually exclusive. That is, no two activities can start and/or stop at exactly the same time.

2. Filters can specify *any* boolean expression involving activities. This feature provides a very powerful mechanism for responding to events and activities in the world.

3. Commands that are specified in the filter can be any **tcl** script. As the activity manager uses a client/server model; if a %S string is found in the command in the filter it is replaced by the clients identity allowing a callback mechanism to the client who originally added the filter. By adding a %S as a fifth argument to the filter the entire command will be sent to the original client.

Typically, the way the activity manager is used is that clients add any filters they will be using when they are created (activating the ones that they want to use and deactivating ones that are not). Then, clients will begin to start and stop events. Each time an activity is started or stopped the filter list is searched to see if any of them will trigger[4]. A list is made of all the filters that will trigger from the event and then each of the commands is executed. Typically, the commands will be executed by one of the clients, so the server just sends the command to the appropriate client. If the command is to run by a client the server normally will not wait for it to finish execution (unless specified otherwise).

When the activity manager is being used to maintain the world events then the clients are usually agents. The filter commands will activate specific perceptual sensor units. If the activity manager is internal to the agent then the clients are the different levels of the agent hierarchy.

To illustrate how this all works, consider a simple example of maintaining the world of the 99 game using the knowledge monitor. In this example, the world is being managed by the activity manager. The agents themselves are implemented using standard non-hierarchical techniques.

In this example, there is a agent called **green** who is created. Player green has player **red** on the left and player **blue** on the right. Here are the filters that **green** adds:

```
1. AM addFilter {} {} {PLAY_CARD} {green \
       checkWhatWasPlayed} %S
2. AM addFilter {LOST} {} {} {green        \
       keepTrackOfLosingPlayer} %S
3. AM addFilter {} {![NOW GAME_OVER]}      \
       {RED} {green takeMyTurn} %S
4. AM addFilter {} {![NOW GAME_OVER]}      \
       {BLUE} {green takeMyTurn} %S"
5. AM addFilter {} {![NOW GAME_OVER]}      \
       {RED} {green IAmOutOfGame} %S
6. AM addFilter {} {![NOW GAME_OVER]}      \
       {BLUE} {green IAmOutOfGame} %S
7. AM addFilter {GAME_OVER} {} {} {green \
       summarizeMyPerformance} %S
```

[4] A filter triggers when all of its conditions are true.

Filter 1 triggers whenever any player plays a card. The method **checkWhatWasPlayed** looks at which card was played by any player (by asking the world's knowledge base) and updates the current total. If a "4" was played the **checkWhatWasPlayed** method must also deactivate and activate the appropriate filters (either 3 4, 5 or 6) so that the agent takes its turn in the correct order.

Filter 2 triggers whenever any player declares they are out of the game and have **LOST** (i.e. by telling the activity manager **start LOST**. The **keepTrackOfLosingPlayer** method also checks to see if this is the second last player who lost; if so, that means that **green** has won the game.

Filters 3 and 4 are filters which trigger whenever the player to the left or the player to the right of **green** respectively finish their turn *and* **green** is still in the game. Only one of these filters is active at any one time. Whenever a "4" is played these two filters switch activation. The **checkWhatWasPlayed** command does the actual filter manipulation. The method **takeMyTurn** starts **green** (i.e. issues a **start GREEN** command), chooses a card to play, plays the card (**PLAY_CARD**), picks a new card, and finally says it has finished playing (**stop GREEN**). Notice, this method also determines when **green** cannot play a card. When **green** loses it must tell all the other players (using **start LOST**), deactivate the **takeMyTurn** filters (3 and 4) and activate the appropriate **IAmOutOfGame** filter (either filter 5 or 6 - see below).

Filters 5 and 6 are similar to filters 3 and 4 in that they trigger whenever the player to the left or the player to the right of **green** respectively finish their turn; however, they are only activated when **green** is no longer in the game. They effectively take the place of the **takeMyTurn** filters so that **green** maintains the order of play without playing any cards. Effectively, the method **IAmOutOfGame** just starts and stops green and does nothing else. Just as with filters 3 and 4, only one of these filters is active at any one time and when a "4" is played these two filters switch activation.

Finally, filter 7 triggers whenever **GAME_OVER** starts. This happens when one of the players determines that it has won. The **summarizeMyPerformance** method prints out winning statistics and summarizes **green's** performance.

With just these 7 simple filters the agents can play the game of 99. Since the entire system runs asynchronously, when the agents are not playing they can be busy doing other tasks.

The activity manager is used inside of an agent to manage all the activities which occur internally. These activities are started and stopped by each of the various levels in the multi-level agent hierarchy. The client/server model employed by the activity manager ensures that each of the levels can act independently. The activity manager is a generic mechanism and does **not** impose any structure on the activities which are started and stopped nor does it impose any restrictions on the filter commands. It is the responsibility of the agent designer to make sure that communication within the multi-level hierarchy does not jump levels by mis-using the activity manager.

In the current description, the activities are relatively simple. In general, activities can carry information with them. The whole structure of an activity includes:

1. the name of the activity
2. the priority of the activity
3. an (optional) message with the activity such as a variable value
4. an (optional) function which returns a message

The optional message allows different layers of the hierarchy to transfer variable values which are kept locally inside them. Of course, the knowledge manager could be used to keep track of all variable values so that messaging is unnecessary, however, the point of the hierarchy is to separate as much of the task space as possible into hierarchical levels. Misuse of the hierarchy is similar to how some C programmers who begin programming in C++ do not use the object oriented features of the language. C++ does not enforce the structure in the language; instead, it provides facilities to enable using object oriented structures. Likewise, the agent hierarchy is a framework that depends on the designer to adhere to. The optional function in an activity is a mechanism which allows activities to have time-varying values which are contained in the message that is returned.

3.1.2 Knowledge Base

The knowledge base is the set of variables which represent the state of the world (or internals of an agent). The content of the knowledge base is available to all the clients connected to the activity manager. If a client wants to hide its data it must keep track of it locally and send it to other clients using a message protocol. In the game of 99 the knowledge base contains the deck of cards being used to play with. Thus, the knowledge base keeps track of the discard pile, how many cards left in the deck and the order of the cards in the deck.

3.1.3 Resource Manager

The resource manager keeps track of all resources which are shared by mutually exclusive clients. The resource manager sets up a priority system of allocating shared resources. Clients using shared resources must give up the resource to a higher priority client request for the resource. Typically, each client sets up a filter which will trigger when there is a higher priority resource request for the resource it is using. The method that is called should do whatever is necessary for the client to give up the resource. This mechanism is similar in spirit to [2].

The activity manager can be used in a variety of settings. In terms of agents, it can be used either to manage the world state or the internals of an agent. When used for maintaining the internal state of an A-HA care must be taken to restrict communication between levels in the hierarchy to properly adhere to the hierarchy model. If not, the logic of the agents will become very difficult as dependencies will cross levels. The next section describes the multi-level hierarchical agent architecture which uses the activity manager as its foundation.

In the simple game of 99 the resource manager is used implicitly. The only resource used is the deck of cards. The resource manager allocates a new card from the deck when requested by an agent.

3.2 Hierarchical Levels in an Agent

Within each agent the task space is divided into a 5 level hierarchy. Within each level are two complimentary functions, one for response and another for perception. These correspond to the top-down and bottom-up processing that an agent performs as shown in figure 1. From a designer's perspective, each new application area can be broken down logically into the separate levels and functions for each level can be systematically created independent of other levels of the hierarchy. The interface between layers must be defined so that only neighbouring layers may communicate. Other researchers have suggested hierarchically layered agent architectures such as [2] [3] [5].

For our agents we decompose function space into 5 levels:

1. Motivation/Needs
2. Policy/Plan
3. Behaviour
4. Action and Sensation
5. Action Units and Sensor Units

The first level is motivation and needs. Motivation and needs are qualities that the agent is trying to fulfill for itself. In the case of the game of 99 the motivations are: to play the game, to win, and maintain its level of conservatism in play.

The amount of conservatism an agent has impacts how much risk it is willing to take when choosing to play a card. Remember, there are two distinctive process which are contained at the motivation level; response and perception. The response process is trying to negotiate with the lower levels to fulfill the motivation and needs of the agent. The perception process is trying to understand the motivation and needs of other agents in the world in a reciprocal manner. In fact, one can imagine that in understanding another agent, an agent can use the same response mechanisms it would use to invert actions that the other agent performs on the world to determine its motivation and needs. This information, which is placed in the agent's knowledge base, can be used to assist all levels of the hierarchy to ultimately fulfill the needs of the agent.

The second level is policy, also called planning. The policy level is responsible for reasoning about the current state of the agent and the world (depending on the information it received from the other layers). Typically, it will use a variety of planning algorithms to determine what should be done. The policy level may have several alternative systems which are available for both responding in the world and for perceiving. It is the role of the motivation layer to activate the appropriate policy at the appropriate time when responding. Similarly, different policy mechanisms may be used to understand actions performed by other agents during perception.

The third level is the behaviour level. Behaviour consists of *goal* oriented actions. Goals oriented behaviour can also be considered as *conscious* behaviour. Thus, when responding in the world, the behaviour level is creating sequences of actions to achieve some goal (usually specified by the policy level). Its perceptual counterpart determines the goal which was trying to be achieved by a sequence of actions performed by another agent.

The fourth level is the action/sensation level. Actions are not goal directed, but are *automatic* sequences (or behaviours) which perform some specified action in the world. The perceptual part of the action/sensation level recognizes actions performed by other agents in the world.

Often, it is the case that actions may be decomposed into small *atomic* actions. These atomic actions are implemented by action/sensor units which make the last level in the hierarchy. The action/sensor units level is the only level which can actually act on the world or sense the world. It has the ability to manipulate filters in both the agent's activity manager and the world's activity manager. Additionally, it may add filters which are conditioned on activities in both places. Likewise, it can start and stop activities in both the world and the agents activity manager. The action/sensor unit level's perceptual process is required to recognize atomic activities performed by other agents. Typically, it will do this by placing appropriate filters on the world.

Notice, that it is possible to implement *learning* algorithms for each level separately. At this point in the project, plan to hard code each level. However, it should be possible to use neighbouring levels to provide training signals for any particular level. If the process to be learned is perceptual then the lower levels provide input for the upper levels and the upper levels can provide evaluation signals for the learning. Similarly, for learning a response process the situation is reversed.

The game of 99 has enough complexity to implement an agent using the multi-level hierarchy to play the game. We plan to incorporate the action/sensor unit layer and the action/sensation layer as all of our actions are atomic. In the next section, the decomposition of the game of 99 from an agent's perspective is considered.

3.3 Agents Hierarchy used to play 99

This section describes a functional decomposition for each of the levels, motivation/needs, policy/plan, behaviour, and action/sensation (action/sensor units are not used for the game of 99) in the 99 playing agents. Currently, we have not implemented the full hierarchy. The hierarchy is relatively simple as the game is relatively simple. We are currently considering the best way to decompose the whole task and to which granularity the task should be decomposed. Further, implementation details have not been finalized.

The first layer is the motivation/needs layer. Each agent has 3 motivations:

1. play the game
2. win
3. maintain level of conservatism

The motivation to play the game drives the player to participate in the rules such as when to take its turn, when to pick up a card, announcing it is out of the game, etc. Motivation to win drives behaviours such as selecting the best card and deciding whether to cheat. Maintaining the level of conservatism drives the selection of which policy to choose for selecting a card to play. The reciprocal role of this layer for perception is not used.

The second layer is the policy/plan layer. There are a number of policies (some of which require planning) which can be applied to this simple game. They are:

1. play "King" when possible
2. evaluate risk with playing "King" and decide to play or not
3. play highest value cards first
4. play "9" or "4" before playing a "10"

The agent's policy layer also perceives when it cannot play a card and when the order of play changes.

Other policies are possible of course; but these ones are sufficient to provide enough flexibility to illustrate the system. Policies generally should not be specific to any particular motivation or need. Thus, policies do not explicitly encode the goals of the agent and act independently of motivation. An example is that if the second policy dictates to play a "King" it may be interpreted as satisfying the need to be conservative or the motivation to win or both. Currently, we do not plan to use the perception of other agents policies.

The third layer is the behaviour layer. An agent has three behaviours which are:

1. take turn: comprising play card and pick up new card
2. indicate loss to other players
3. pick up 3 cards at start of game to begin the game

The agents also perceives the following aspects:

1. the value of a card
2. detect that the game is over

Notice that some of these behaviours could also be considered actions if desired. However, from our perspective it is convenient to group them into the behaviour category. This illustrates that the functional separation between layers is determined by the designer to match the task.

The fourth layer is the actions/sensation (and action/sensor units) layer. There are four actions that an agent performs. They are:

1. Put card on pile
2. pick up card from deck
3. reshuffle deck
4. Say that agent can't play (i.e. that I lose)

Further, there are 3 sensors which the agent has to watch the game progress. They are:

1. Detect when a player has finished their turn
2. Detect which card was played by another entity
3. Detect when a player is out of the game

It is the action/sensation layer that registers activities with the world in response to the play of the game. Currently, the activities it registers include: the name of the agent playing, when an agent plays a card, when an agent loses and when the agent believes the game is over (see section Activity Manager). The action/sensation layer also registers filters with the world's knowledge monitor to watch for activities to occur.

4 Conclusions

Currently, the hierarchy has not been fully implemented yet. The outline above though is the basis upon which we are working. Thus far we have used the knowledge monitor to manage the game of 99 world and implemented agents non-hierarchically. Results thus far have confirmed the feasibility of using the asynchronous behaviour of the knowledge monitor for supporting agents in the game of 99 world.

It is our belief that decomposing the internal structure of agents into hierarchical layers will provide a convenient mechanism for both agentees and designers to easily interact with agents. Further, we expect the structure described to facilitate incorporation of adaptive algorithms at various levels of the agent hierarchy.

This paper has described a five layer hierarchical model for creating and interacting with agents. It is our belief that agents in the real world will have to respond to events which are occurring in *real time* and thus need to be able to respond asynchronously. The knowledge monitor structure behaves as a filtered interrupt vector table which effectively allows agents to manage asynchronous events which occur internally or in the world.

References

[1] AT&T Bell Laboratories. [incr tcl] vrs. 2.1. available through http://www.tcltk.com/itcl.

[2] Blumberg, B. M., and Galyean, T. A. Multi-level direction of autonomous creatures for real-time virtual environments. In *SIGGRAPH 95* (August 1995), pp. 47–54.

[3] Bonasso, R. P., Kortenkamp, D., Miller, D. P., and Slack, M. Experiences with an architecture for intelligent, reactive agents. In *Intelligent Agents II*, M. Woolridge, J. P. Müller, and M. Tambe, Eds. Springer, 1995, pp. 187–202.

[4] Etzioni, W., and Weld, D. S. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert 10*, 4 (1995), 44–49.

[5] Müller, J. P., Pischel, M., and Thiel, M. Modelling reactive behaviour in vertically layered agent architectures. In *Intelligent Agents*, M. Woolridge and N. R. Jennings, Eds. Springer-Verlag, 1994, pp. 261–276.