Optimisation of Active Rule Agents Using a Genetic Algorithm Approach

Evaggelos Nonas[1] and Alexandra Poulovassilis[2]

[1] Department of Computer Science, King’s College London, Strand,
London WC2R 2LS, U.K.
vagelis@dcs.kcl.ac.uk
[2] Department of Computer Science, King’s College London, Strand,
London WC2R 2LS, U.K.
alex@dcs.kcl.ac.uk

Abstract. *Intelligent agents and active databases have a number of common characteristics, the most important of which is that they both execute actions by firing rules upon events occurring provided certain conditions hold. This paper assumes that the knowledge of an intelligent agent is expressed using a set of active rules and proposes a method for optimising the rule-base of such an agent using a Genetic Algorithm. We illustrate the applicability of this method by using it to optimise the performance of a self-adaptive network. The benefits of our approach are simplified design and reduced development and maintenance times of rule-based agents in the face of dynamically evolving environments.*

1 Introduction

Beliefs-desires-intentions (BDI) agents have been around for some years now and have been extensively studied ([5], [9], [11], [2]). A BDI agent has the following components:

- Beliefs Database: Contains facts about the sate of the world, as well as about the agent's internal state.
- Desires: Contains agent's goals expressed as conditions over some interval of time and are described by applying various temporal operators to state descriptions.
- Plans: Actions the agent has to take in order to fulfill its goals. They have an invocation condition which specifies upon which events the plan should be fired and a context condition which specifies under what condition the plan applies.
- Intentions: Plans that are valid for firing are placed in an intentions structure where they are executed. They can be hierarchically ordered.

Active databases are also based upon an Event-Action architecture [4]. An active database system consists of the “traditional” components of a database system plus a component that is concerned with the firing of ECA (event, condition, action) rules. The meaning of an ECA rule is: “when an event occurs check the condition and if it is true execute the action”. There is an event language for defining events and for specifying composite events from a set of primitive ones. The condition part of an ECA rule formulates in which state the database has to be, in order for the action to be executed. The action part of an ECA rule usually starts a new transaction which when executed may trigger new ECA rules. In this way we can have trees of triggering and triggered transactions.

Similarities and differences between BDI agents and active databases are discussed in [1], [12], where characteristics such as events, actions, consistency, query expressiveness, goal achievement, responsiveness and others are compared. The most important of their common characteristics is the way actions are executed, in that upon a certain event occurring, if a condition holds a rule is fired. There may be cases where more than one rule may be triggered by the same event occurrence. The system will then select the rule with the highest priority to fire, or will arbitrarily select a rule to fire if there are more than one with the same priority.

In this paper we assume that the knowledge of an intelligent agent is expressed using a set of active rules, and that a genetic algorithm determines the “best” rule to fire if more than one rule is triggered. Genetic algorithms and genetic programming have been used before in the design of agent systems [8], [10]. The novelty of our work is that we are using GAs to select the “best” rule to be fired and also to automatically respond to changes in the environment.

The outline of this paper is as follows: In section 2 GAs are described and their use in optimising rule based agents is proposed. In section 3 we apply our methods to the problem of optimising a self-adaptive network. In section 4 some results of our application are presented. Finally, some conclusions and possible directions for further research are presented.

2 GAs and Rule-Based Agents

Genetic algorithms ([7], [3], [6]) are methods of solving problems based upon an abstraction of the process of Natural Selection. They attempt to mimic nature by evolving solutions to problems rather than designing them. Genetic algorithms work by analogy with Natural Selection as follows. First, a population pool of chromosomes is maintained. The chromosomes are strings of symbols or numbers. They might be as simple as strings of bits - the simplest type of string possible. There is good precedence for this since humans are defined in DNA using a four-symbol alphabet. The chromosomes are also called the genotype (the coding of the solution), as opposed to the phenotype (the solution itself). In the Genetic algorithm we maintain a pool of chromosomes, which are strings. These chromosomes must be evaluated for fitness. Poor solutions are purged and small changes are made to existing solutions. We then allow "natural selection" to take its course, evolving the gene pool so that steadily better solutions are discovered.

The basic outline of a Genetic Algorithm is as follows:

```
Initialise pool randomly

for each generation

{

  Select good solutions to breed new population

  Create new solutions from parents

  Evaluate new solutions for fitness

  Replace old population with new ones

}
```

The randomly assigned initial pool is presumably pretty poor. However, successive generations improve, for a number of reasons:

- Selection. On each generation parents are selected to produce new children. The selection of parents is biased by fitness, so that fit parents produce more children; very unfit solutions produce no children. This is known as selection. The genes of good solutions thus begin to proliferate through the population.
- Mutation. Small changes (mutations) are made to at least some of the newly created children. Some of these mutations may be harmful. However, this doesn't matter as bad mutations will soon be purged by selection. Good mutations, on the other hand, will succeed, causing further increases in fitness.
- Crossover. It combines the genetic material from parents in order to produce children, during breeding. Since only the good solutions are picked for breeding, during the selection procedure, the crossover operator mixes the genetic material, in order to produce children with even greater fitness. For example, if we assume single point crossover at position 3 two binary chromosomes with values (000000, 111111) will produce (000111, 111000) as children. Moreover, there can be multiple point crossover.

What we propose here, is an “automatic” way of selecting the best action to execute upon an event occurring. The action is selected by a genetic algorithm. For the moment we do not support conditions in our active rules, although we plan to add a condition part in the future. When an event has occurred the system can take several actions. For each of possible events, the system holds an ordered set of possible actions that can be taken when the event occurs. The first action is always selected, but a genetic algorithm running in parallel may dynamically change the order of the actions. Obviously this approach requires a measure of the performance of the agent, which must be available at run-time, to be given to the genetic algorithm.

Since the genetic algorithm controls the way the agents respond to events, we can say that the reactive behaviour of the agent is controlled by the genetic algorithm. But there can also be another "level" (the "rational" level) to control the agent, especially if our architecture is part of an agent built partially using another method and controlled partially by the constructs this method provides. If for instance the agent is built conforming to the BDI model, it will have facts, goals, plans and intentions (plans that have been selected for execution). Some of the plans will be selected for execution using the traditional approach, but some others using the GA approach. This rational part of the agent can also control several parameters of the GA, restart it when needed, or schedule it to be run when the load is low.

Our architecture can also be embedded in more complex systems. When an event/action language is necessary for the building of an agent type system, our method can be used for a subset of the events and the actions of the system. This simplifies the design and reduces testing and maintenance times when compared to a deterministic rule-set with many conditions and checks.

3 An Example Application: Self-Adaptive Networks

A self adaptive network is a network that can automatically adapt to changes in its environment without human intervention being necessary. While load conditions change and nodes and links may fail, the network continues to operate near the optimum state requiring little or no assistance from its operators. In other words, the network must be autonomous, intelligent and have distributed control. There should be no global knowledge for the network. On the contrary all information must be kept as local as possible.

Our network model is a simple yet powerful one. The network is composed of a set of nodes and a set of connections between them. Each node can exchange messages only with the ones it is connected with. There is no global knowledge of the topology of the network stored in any node. There is a set of services provided by the network and each node can provide some or all of the services. The task for every node is to provide the services requested from it with the minimum cost. The cost can be a function of the number of intermediate nodes and links the service is using, as well as the load and the spare capacity of those nodes and links respectively. Obviously, the bigger the number of intermediate nodes and links a service is using, the bigger the cost for the provision of that service is.

Messages are exchanged between nodes to allow service establishment and service canceling. Messages can be exchanged only between connected nodes. For the time being we use three kinds of messages, but we intend to add more in the future. The first requests a service from a node and has as parameters the requesting node, the service number as well as the hop-count (number of intermediate nodes the request has used). Messages with a hop-count greater than a specific number are canceled automatically, to avoid flooding the network with cyclic or very long requests. The second type of message concerns the answer to a request for a service. If the service can be provided, the cost is returned, otherwise the message just rejects the request. The third kind of message cancels services already provided.

Table 1. A Simple Example

Events	Actions
On provide any service	Local, Remote
On send request for service 1	Send to node 2, Send to node 3, Send to node 4
On send request for service 2	Send to node 2, Send to node 3, Send to node 4
On send request for service 3	Send to node 2, Send to node 3, Send to node 4

In each node of the network there is an agent running which is built using the proposed architecture. For the time being only the reactive part of the agent has been built, although in the future we intend to build a rational part, together with event and action languages. There is a set of events and a set of actions for each event. An event occurs when a service is requested. There are two actions that can be triggered for this event: the service can be provided either remotely or locally. When a request for a service is to be sent to another node a separate event is generated for each service. The actions corresponding to this event are all the nodes that the requesting node is connected with. For instance, if node 1 is connected with nodes 2, 3 and 4 and the network provides 3 services, the events and actions for node 1 are shown in Table 1 (no order is shown, just all the events and all the possible actions for each event).

A simple genetic algorithm is used to try out several permutations of the rule set and finally find the best rule set for each node. Permutations of actions for each event are enumerated and placed in the chromosome one after the other. We define a correspondence between all possible permutations of the actions for an event and an integer in the range [0..n!-1], where n is the number of actions. The binary representation of this number is placed in the chromosome to encode the ordered set of actions for an event. The whole chromosome is composed of K (where K is the number of events) numbers placed in it, in their binary representation, one after the other. In this way one chromosome can encode the rule set with which each agent works.

Each agent has a chromosome pool which is initially randomly instantiated. Then the chromosomes in the chromosome pool are evolved by the genetic algorithm to better solutions. We use a constant population size, selection proportional to fitness

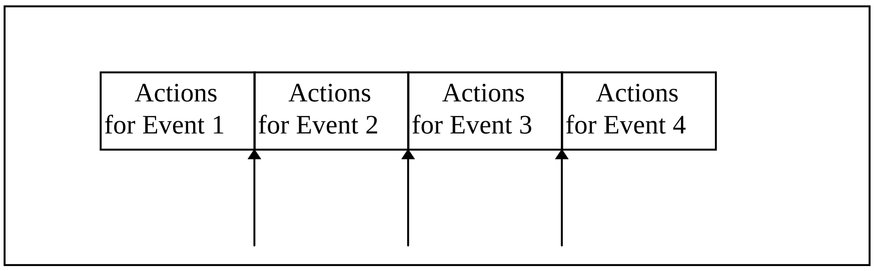

Fig. 1. Chromosome Encoding

and full replacement of parents by their children. Multiple point crossover is used for breeding. Crossover points are set at the end of each event in the chromosome. The chromosome for the example of Table 1 is shown in Figure 1, where arrows show the positions of the crossover points.

The fitness of each rule is calculated as follows: When a node provides a service to another node, it also sends to it the cost of this service. This cost, is a function of the number of intermediate nodes and links the service is using as well as their load and free capacity respectively. Obviously, when the service is provided locally, the cost is minimum. Each rule in the chromosome pool is used for service provision for some time and the costs of the services provided using this rule are averaged. The fitness, then, for this rule is inversely proportional to this average cost. So the bigger the cost, the smaller the fitness of the rule and vice-versa.

The current implementation of our architecture is in Borland C++ Builder and runs under the Win32 platform (Windows 95 and Windows NT). A network simulator as well as the actual agents running on each node of the network have been built. Finally the genetic algorithms used by the agents have also been programmed. There is a graphical user interface that provides for the design of the network, the design of the rules the agents are using and the fine-tuning of the genetic algorithm that each agent runs.

4 Experiments and Results

In this section we present some results for several network configurations. In all the graphs the Y axis shows the average of the mean fitness of the chromosome pools over all the nodes. The X axis shows the number of generations the genetic algorithm has been run. While nodes are being trained service requests have a uniform distribution as far as type of service is concerned, across all nodes. Of course, real traffic data can ultimately be used for more effective training.

Our first experiment is concerned with a network of 100 nodes and 200 links. The topology has been randomly created by our software. In total there are 40 services provided across the network. We examine three different cases with varying service distribution across nodes. In the first case all 40 services are provided by all the nodes. In the second case there is a random distribution of services across nodes. The number of different services provided by each node is drawn randomly from the range [1..40]. In the third case, services and nodes are split into 5 disjoint sets and eight services are provided by each node. For example, nodes 1 to 20 provide services 1 to 8, nodes 21 to 40 provide services 9 to 16, etc. Graphs for all three cases are shown in Figure 2 under the legends Totally Replicated, Random and Partitioned respectively.

As we would expect, the best performance is achieved when services are totally replicated across all nodes. The worst performance is achieved when services and nodes are partitioned into disjoint sets. This is because only a few of the total number of services can be provided locally, or with a small hop-count. Random distribution of services results in a performance between the two “extreme” cases.

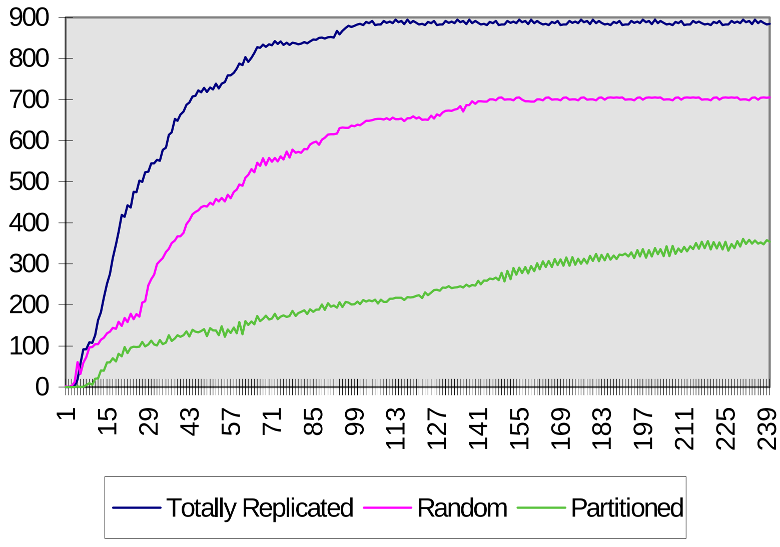

Fig. 2. Varying the Distribution of Services

Our second experiment demonstrates the fault tolerance of the network and its behaviour is illustrated in Figure 3. There is a network of 11 nodes and 10 services. 10 of the nodes are connected in a ring and provide only 3 services each, which vary from node to node. The 11th node provides all 10 services and is connected with all the other nodes. So it is the most important node of the network. The GAs are run and at generation 150, this 11th node of the network goes down. We see initially a

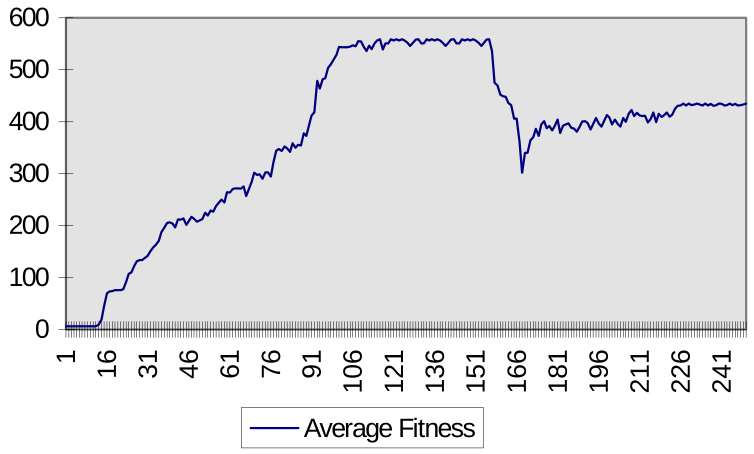

Fig. 3. Fault Tolerance Demonstration

decrease in the fitness of the rule-sets in the remaining 10 nodes, since the services that were provided by node 11 to these nodes cannot now be provided. But the GAs quickly evolve better rule-sets and their fitness increases again. Not surprisingly, it cannot be restored to its original value, since services that were before provided in only one hop-count from node 11 (since node 11 was connected to every other node) now have to be provided with a bigger hop-count from other nodes.

5 Conclusions

In this paper we have described how sets of active rules can be used to express the knowledge of intelligent agents, and how a genetic algorithm can be used to dynamically prioritise rules in the face of dynamically evolving environments. To our knowledge, this is the first time that GAs have been used for this purpose. We have illustrated the applicability of our method by using it to optimise the performance of a self-adaptive network. The advantages of our approach to optimising self-adaptive networks are apparent: distributed solution, load balancing and sharing, and self adaptation to varying load conditions and fault situations.

One could argue that the genetic algorithm can find a local optimum and then stop. This is always a danger with a genetic algorithm, but again it depends on the search space. In a network where service distribution across nodes is done in such way that neighbouring nodes have some services in common there are many good solutions and the genetic algorithm will find one of them. In extreme cases where there is only one good solution the genetic algorithm may fail, but again it can be restarted by the rational part of the agent with many chances to find a better solution. Finally, in such cases the advantages of adaptation, autonomy and distributed operation are more important than the discovery of the best solution, especially in a dynamic and continually changing environment where keeping track of global information would be difficult if not impossible.

One could also argue that this architecture is not powerful enough since it does not work based on an event/action language. However there is nothing to prevent this architecture from being a subset of a rich and powerful event/action language. In such a case it can be used to pick the rule to be fired when there are no other criteria available for rule selection. In other cases it may be better to let the genetic algorithm pick the rule to be fired, instead of having many conditions which will complicate the active rule set and consequently increase design, test and maintenance times.

For further work we plan to construct the rational part of the agent. This too will be based on active rules. It will schedule, restart and fine tune the genetic algorithm. It will also feed it with a good initial population and will provide for knowledge exchange between neighboring nodes. We believe that this combination of intelligence and heuristic search methods like genetic algorithms will lead to a much better performance than use of the latter alone.

We will also explore different types of genetic algorithms, for example ones with overlapping populations such as steady state or incremental GAs. In such cases, we can have a small replacement percentage, so that the GA could be used for driving the nodes at real time (once an initial good state has been reached) and not just training them. The above would be useful mostly for relatively stable environments. We also plan to investigate other methods for finding the optimum rule set (for example, neural networks or other heuristic search methods like simulated annealing) and to formally compare our results with theoretical results obtained by a statistical analysis of the network.

Acknowledgments

E. Nonas is sponsored by B.T. Laboratories, Systems and Software Unit, Martlesham Heath, Ipswich.

References

1. J. Bailey, M. Georgeff, D. B. Kemp, and D. Kinny, "Active databases and agent systems --- A comparison", Lecture Notes in Computer Science, 985, 342-356, (1995).
2. Michael Bratman, Intention, plans, and practical reason, Harvard University press, 1987.
3. Lawrence Davis, Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.
4. K. R. Dittrich, S. Gatziu, and A. Geppert, "The active database management system manifesto: A rulebase of ADBMS features", Lecture Notes in Computer Science, 985, 3-17, (1995).
5. Klaus Fischer, Jorg P. Muller, and Markus Pischel, "A pragmatic BDI architecture", in Proceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages, volume 1037 of LNAI, pp. 203-218, Berlin, (19-20 August 1996). Springer Verlag.
6. D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, Mass., 1989.
7. David Goldberg, Genetic Algorithms, Addison Wesley, Reading, 1989.
8. Thomas Haynes and Sandip Sen, "Evolving behavioral strategies in predators and prey", in IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems, pp. 32-37, (1995).
9. David Kinny, Michael Georgeff, and Anand Rao, "A methodology and modelling technique for systems of BDI agents", in Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, volume 1038 of LNAI, pp. 56-71, Berlin, (22-25 January 1996). Springer Verlag.
10. Mauro Manela and J. A. Campbell, "Designing good pursuit problems as testbeds for distributed AI: A novel application of genetic algorithms, in Proceedings of the 5th European Workshop on Modelling Autonomous Agents in a Multi-Agend World (MAAMAW'93), volume 957 of LNAI, pp. 231-252, Berlin, GER, (August 1995). Springer.
11. Anand S. Rao and Michael P. Georgeff, "BDI agents: from theory to practice", in Proceedings of the First International Conference on Multi—Agent Systems, pp. 312-319, San Francisco, CA, (1995). MIT Press.

12.Johan van den Akker and Arno Siebes, “Enriching active databases with agent technology”, in Proceedings ot the First International Workshop on Cooperative Information Agents, volume 1202 of LNAI, pp. 116--125, Berlin, (February26--28 1997). Springer.