Optimising Self Adaptive Networks by Evolving Rule-Based Agents

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. The need for networks that adapt autonomously to dynamic environments is apparent. In this paper we describe how self adaptive networks can be optimised by means of agents residing on the nodes of the network. The knowledge of these agents is a set of active rules. A genetic algorithm dynamically prioritises these rules in the face of dynamically evolving conditions. To our knowledge, this is the first time that GAs have been used for this purpose. We demonstrate the applicability of our method by presenting several experiments and results.

1 Introduction

As telecommunication networks have become bigger and more complex, the need for managing them effectively, optimising their capacity and reducing their operation costs has become apparent. This need is becoming more urgent as the telecommunications market is continuously changing and new services have to be constructed and provided quickly and cheaply. Moreover, since users are becoming mobile, networks have to adapt quickly to varying load conditions and traffic patterns.

There are many methods for optimising a network. Some of them solve this problem using an analytical approach, others using an evolutionary approach. Both categories give good results from a long-term point of view. They use statistical data to calculate average costs, which do not change over a long period of time: weeks or even months.

In contrast, what we propose is a method for optimising the network on the protocol level, by using costs that can be predefined or collected at run-time. For the optimisation procedure we combine the analytical with the evolutionary approach. Parts of the problem are solved using a deterministic algorithm, and other more

complicated parts are solved using a genetic algorithm. Such a network will have the ability to adapt to dynamic environments with little or no human intervention, so it is termed a Self Adaptive Network.

We use software agents that reside on each node of the network and optimise it in real time. The knowledge of each agent is expressed in the form of active rules consisting of events and actions. The reactive part of the agent (the part that responds to external events) is dynamically optimised by a genetic algorithm. The rational part of the agent also uses active rules, but these are statically defined.

The outline of this paper is as follows: Section 2 describes the main features of self adaptive networks. Section 3 describes and compares two software architectures that work using active rules: beliefs, desires, intentions (BDI) agents and active databases. It then describes the active rules that our system uses. Section 4 describes the genetic algorithm that optimises the rule-based agents. Section 5 gives some experiments and results from our system. In section 6 we present conclusions and future research directions.

2 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 need for global knowledge in 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 its immediate neighbours. 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 of the load and spare capacity of those nodes and links respectively. Obviously, the larger the number of intermediate nodes and links a service is using, the larger the cost for the provision of that service.

Messages are exchanged between nodes to allow service establishment and service cancelling. 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 kind of message 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 kind of message concerns the answer to a request for a

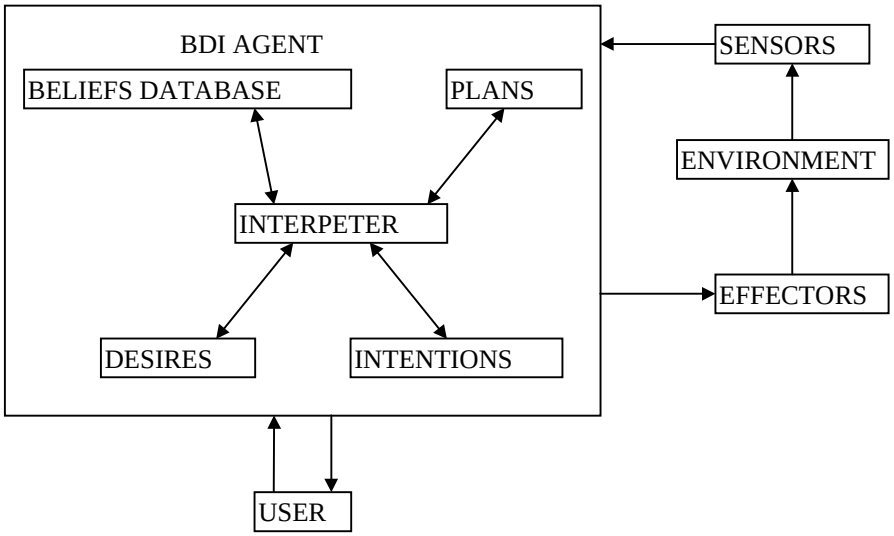

Fig. 1. Typical BDI System

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.

When a service is provided its cost is calculated as follows:

$$\sum_i N_i + \sum_k L_k \quad (1)$$

where i is the number of nodes and k is the number of links the service is using, and N_i and L_k is the load imposed by this service on each node and link, respectively. When the service can not be provided (because a node or link has reached its maximum capacity, or because the hop count has exceeded the maximum allowed limit), the same formula is used for the cost of the service, but i and k are now set to the total number of nodes and links in the network respectively. Clearly, more sophisticated cost functions can be used in dynamic environments.

3 Active Databases and BDI Agents

Beliefs-desires-intentions (BDI) agents have been extensively studied for some years [5], [9], [11], [2]. A BDI agent has the following components (see Figure 1):

- 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.

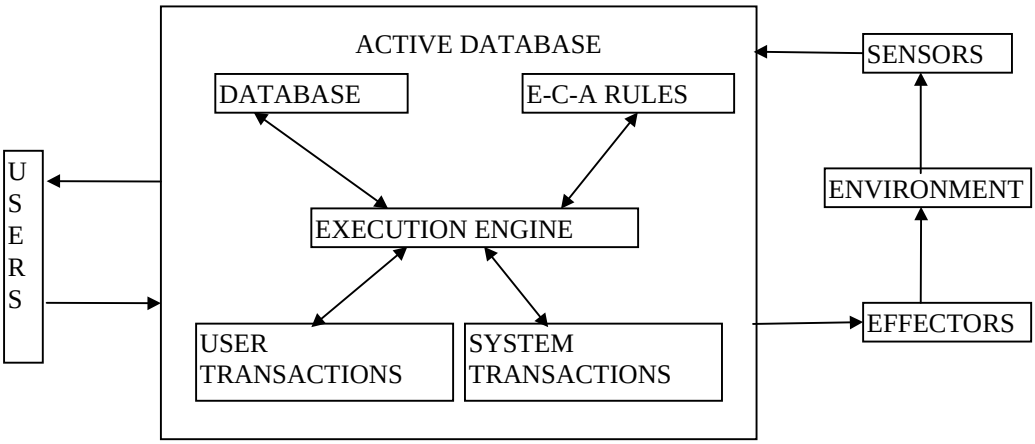

Fig. 2. Typical Active Database System

- 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] (see Figure 2). An active database system consists of the "traditional" components of a database system plus a component that is concerned with the firing of event-condition-action (ECA) 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 may start a new transaction which when executed may trigger new ECA rules. In this way we can have trees of triggering and triggered transactions.

By comparing Figures 1 and 2 we can very easily see the correspondence between the components of the BDI agent and active database architectures. The beliefs database of the BDI agent corresponds to the main database store of an active database. The desires of a BDI agent are expressed in an active database as transactions submitted by users. The plans of a BDI agent correspond to the ECA rules of the active database. Finally the intention structure of the BDI agent is expressed in the active database as transactions generated by the system, i.e. transactions generated from the activation of ECA rules.

Similarities and differences between BDI agents and active databases are discussed in more detail in [1], [12], where characteristics such as events, actions, consistency, query expressiveness, goal achievement and responsiveness are compared. The most important of their common characteristics is the way that actions are executed, in that

Table 1. A Simple Example Rule Set

Events	Actions
Provide service A	Local Remote
Provide service B	Local Remote
Provide service C	Local Remote
Send request for service A	Send to node 2 Send to node 3 Send to node 4
Send request for service B	Send to node 2 Send to node 3 Send to node 4
Send request for service C	Send to node 2 Send to node 3 Send to node 4

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 there is an agent running on each node of the network. The knowledge of each such agent is expressed using a set of rules. An event occurs at a node when it is asked to provide a service. There are two possible actions that can be triggered for this event: the service can be provided remotely, or the service can be provided locally. When a service is to be provided remotely, a new `send request' event is generated. The possible 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 services A, B and C, 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).

4 Using a GA to optimise the rule based agents

Our method provides an automatic way of selecting the "best" rule to fire upon an event occurring, using a genetic algorithm to determine which 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 dynamically optimise a set of rules in response to changes in the environment.

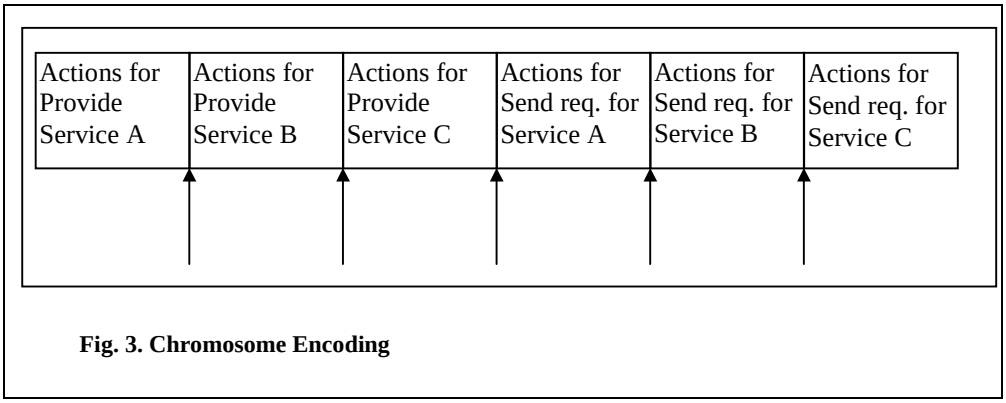

Fig. 3. Chromosome Encoding

For the moment we do not support conditions in our active rules, although we plan to cater for conditions in the future.

At each node, the system holds a list of possible actions that can be taken for each event that may occur. The first action is always selected, but a simple genetic algorithm running in parallel dynamically changes 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.

The GA is used to try out several permutations of the rule set and finally find the best ordering. Permutations of the possible actions for each event are enumerated and placed in the chromosome one after the other. We assign each permutation 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 that permutation of actions for the event. The whole chromosome is composed of a sequence of K such numbers, in their binary representation, where K is the number of possible events. Thus one chromosome can encode all the rules with which each agent works[1].

Each agent has a chromosome pool which is initially randomly instantiated. These chromosomes are evolved by the genetic algorithm to better solutions. We use a constant population size, selection proportional to fitness, 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 3, where the arrows show the positions of the crossover points.

The fitness of each chromosome 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

---

[1] This particular encoding of the GA was chosen initially for ease of programming, but it was adopted since it performed well. Our current library that implements the Genetic Algorithm, does not support other than the binary encoding. An alternative method for describing permutations would be as ordered lists. We are in the process of adding this feature to our library. Once we've done this, we are planning to test the performance of PMX or other permutation crossovers ([3, pp. 72], [6], [13]).

their load and free capacity respectively. Obviously, when the service is provided locally, the cost is minimum. Each chromosome in the chromosome pool is used for service provision for some time and the costs of the services provided using it are averaged. The fitness, then, for this chromosome is inversely proportional to this average cost. So the larger the cost, the smaller the fitness of the chromosome and vice-versa.

The fitness of each rule set is given by:

$$F = 1000 \times \frac{(M - A)^2}{M^2} \quad (2)$$

where M is the maximum cost for service provision and A is the average cost for all the services provided using this rule set. Fitness is normalized between 0 and 1000. The squared term helps the GA to converge more quickly to a solution.

The current implementation of our architecture is in Borland C++ Builder and runs under Windows 95 or Windows NT. A network simulator as well as the actual agents running on each node of the network have been built. 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.

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 part, the "rational" part, that controls the agent, for example 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. Some of the plans will be selected for execution using the traditional approach, but some others using the GA approach. The 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.

5 Experiments and Results

In this section we present some results for several network configurations. In all the graphs, the Y axis shows the mean fitness of the nodes' chromosome pools, averaged 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 uses a network of 100 nodes and 200 links. The topology has been randomly created by our software. 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

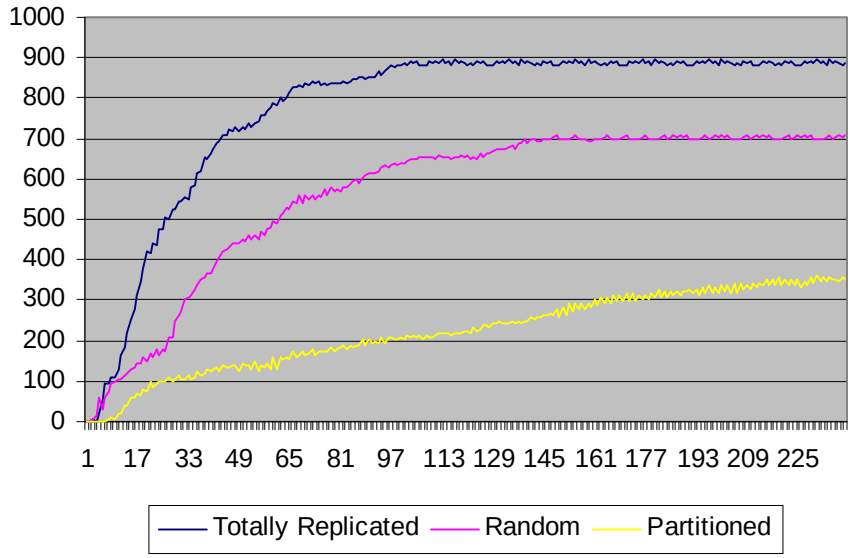

Fig. 4. Varying the Distribution of Services

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 4 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.

Our second experiment demonstrates the fault tolerance of the network and its behaviour is illustrated in Figure 5. There is a network, Network A, consisting 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 black curve in Figure 5 shows the performance of the network when node 11 is down from the beginning of the run until it finishes: we call this network Network B. Network B is optimised to an average fitness of approximately 380. The grey curve initially shows the behavior of Network A, which is optimised to a state higher than Network B. After 500 generations node 11 goes down and the performance of the network decreases initially but after approximately 500 more generations it reaches the expected performance for Network B. At that point node 11 comes up again and the performance of the network is restored to its original value. After 1400 generations from the beginning of the experiment node 11 goes down again but this

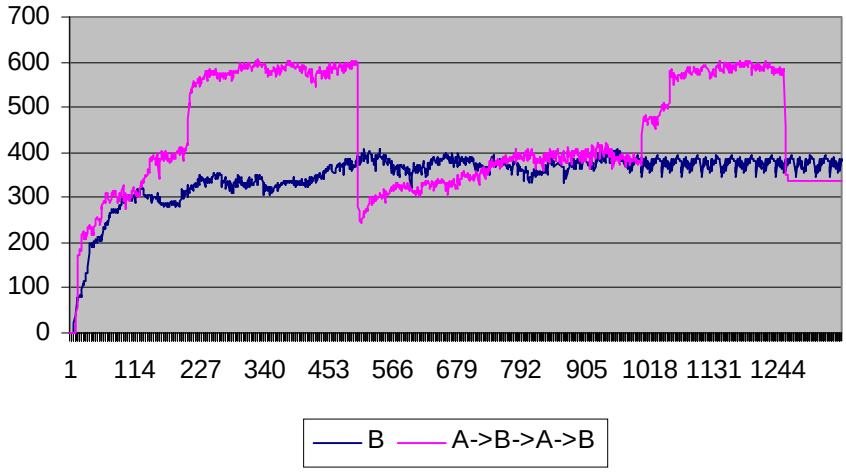

Fig. 5. Fault Tolerance Demonstration

time no GA is used for optimising the network. Instead the agents remove from their rule sets any dependencies they have on node 11. To do this they degrade the priority of actions involving node 11 to the last position in the rule set. The performance of the network after this point is shown by a flat line, since there is no evolution of the rule sets. The performance is about 350, which is close to the 380 mark that the GA can achieve after evolution and certainly near (but a bit lower) the optimum performance for this configuration. This last observation shows that our approach can be used for very quick network restoration and perhaps also for congestion control.

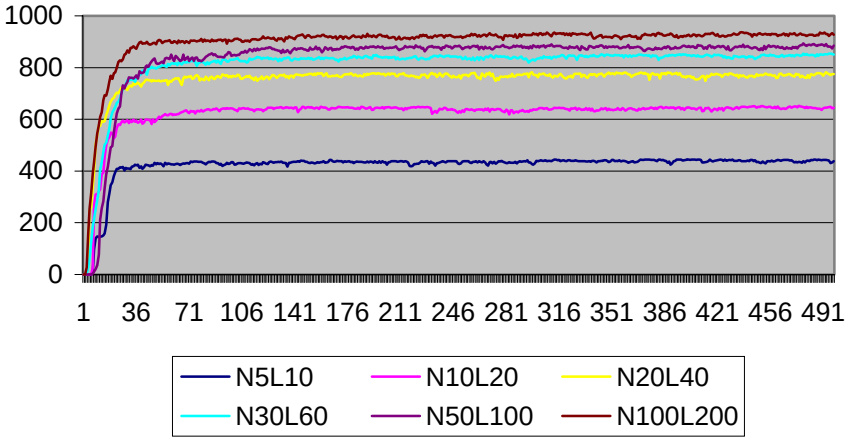

Fig. 6. Speed of Optimisation

It is for this reason that we keep a permutation in the chromosome, instead of a single “best” action for each event. As we have demonstrated in our experiments the second action for an event can be used for network restoration in case of failures. It could be argued though that the actions at the bottom of the event action table will be used rarely and thus that maintaining them is an unnecessary overheard. Thus, further investigation into the usefulness and fitness of the lower-order actions is necessary.

Finally we present a third experiment, which shows the speed the GA optimises a network according to its size. In figure 6 we present six different cases named NXLY, where X and Y are respectively the number of nodes and links the network has. All six networks have a links-to-nodes ratio of two to one. All the networks provide 5 services. We see that the time taken by the GA to optimise the network is not dependent on the size of the network. This is a very important fact that demonstrates the distributed solution and load balancing our method supports. One can also observe that bigger networks have better performance. This is to be expected since bigger networks have more alternatives for providing “cheap” (i.e. lower cost, so higher fitness) services.

6 Conclusions

In this paper we have described how self adaptive networks can be optimised by means of agents residing on the nodes. The knowledge of these agents is a set of active rules. A genetic algorithm dynamically prioritises these 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 showed that our approach is good for network failures and network restoration. We expect it to be well suited to more general conditions of varying load, and more experimentation is necessary into this. 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.

Our network model is connectionless and best effort. In other words it very much matches the TCP/IP routers used to handle traffic on the Internet. It will try to transmit a packet (provide a service in our model) using the best possible way. It will always take the first choice of the active rule set, but if this is unavailable, then it will take the second, and so forth. Another application domain for our approach is global query optimisation in distributed heterogeneous databases. Such systems consist of multiple autonomous databases, and there is little or no global information about local cost models and database contents. We envisage that agents residing on each node could use dynamically evolving active rules to determine the best way to process each type of query (i.e. service) requested at that node.

One could argue that in our system the genetic algorithm can find a local optimum and then stop. This is always a possibility with genetic algorithms, but 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 it can be restarted by the rational part of the agent with many chances of finding a better solution. Overall, the advantages of adaptation, autonomy and distributed operation are more important in self adaptive networks 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.

For further work we plan to construct the rational part of the agents. 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 neighbouring nodes. Scheduling and restarting can be done depending on changing load conditions, on changing network topologies and on the spare computational capacity of the nodes, since they also have to provide services to the network. Depending on those conditions, the rational part can either use the GA to re-optimise the network or based on the knowledge it already has can adjust the rule base for better performance. We believe that this combination of intelligence and heuristic search methods will lead to a much better performance than use of the latter alone.

Finally, we plan to apply our approach to the problem of query optimisation in distributed, heterogeneous databases, where there may be many possible ways for a query to be answered. In this context, the rational parts of the agents will facitate information sharing between the data sources while the reactive parts will optimise distributed access to the data.

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.
13. G. Syswerda, “Schedule Optimisation using Genetic Algorithms”, In L. Davis, editor, “Handbook of Genetic Algorithms”, chapter 21, pp. 332-349, Van Nostrand Reinhold, 1991.