A Hybrid Trainable Rule-based System

Riccardo Poli and Mike Brayshaw

School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom
E-mail: {R.Poli,M.C.Brayshaw}@cs.bham.ac.uk

Technical Report: CSRP-95-3
March 1995

Abstract

In this paper we introduce a new formalism for rule specification that extends the behaviour of a traditional rule based system and allows the natural development of hybrid trainable systems.

The formalism in itself allows a simple and concise specification of rules and lends itself to the introduction of symbolic rule induction mechanisms (example-based knowledge acquisition) as well as artificial neural networks.

In the paper we describe such a formalism and four increasingly powerful mechanisms for rule induction. The first one is based on a truth-table representation; the second is based on a form of example based learning; the third on feed-forward artificial neural nets; the fourth on genetic algorithms.

Examples of systems based on these hybrid paradigms are presented and their advantages with respect to traditional approaches are discussed.

1 Introduction

Since the re-emergence of neural networks in the early 1980's there has been a great debate about neural versus symbolic approaches to modelling the mind. From the cognitive perspective, some have tried to tell the story solely in terms of one technique (e.g. connectionist systems that implement symbolic accounts at higher levels of explanation [11], or wholly symbolically [2]). Others have argued for a hybrid approach with appropriate virtual machines at various levels of a hierarchy (e.g. [1]). Such hierarchies typically allow low-level and motor functions as well as long-term associative memories to be implemented by networks whilst high level things like logical reasoning and mental models are implemented symbolically.

From an engineering perspective, in symbolic AI the value of the integration of heterogeneous inference methods is already well understood, e.g.

CAKE [7] and KEATS [4]. Therefore, it seems natural to try and combine symbolic inference techniques with non-symbolic models. One possibility is to implement symbol systems using neural networks (e.g. [9, 13, 5]). This has the advantage of producing massively parallel, fault-tolerant systems that have the functionality typically associated with symbolic AI. However, in these systems, extensions often require retraining the nets or changing their topology, so that the symbol systems implemented may not be readily changeable. Another approach is to use different technologies for different parts of the system. In general a neural network is used for pre-processing tasks, such as pattern recognition/classification, while a traditional inference system performs the higher-level tasks, such as reasoning. Schreinmakers [8] provides a good example of this sort. However, this weak integration of the two technologies does not allow the exploitation of their joint power within the same module.

An alternative approach to hybridisation is to develop techniques which allow a more tight and natural integration between symbolic and neural systems. This would allow the learning, generalisation, fault-tolerance and noise-rejection properties of neural networks to be integrated with the inference power of symbolic systems. In this paper we present a truly hybrid system that has been developed according to this approach.

The paper is organised as follows. We first describe our formalism for rule specification. Then, we present four increasingly powerful techniques for rule induction: one based on a truth-table representation, a second one based on a form of example based learning, a third one based on artificial neural nets and a fourth one based on genetic algorithms. Finally, we report on some experimental results using such techniques and draw some conclusions.

2 Rule Formalism

We are all familiar with rules having the following format (see for example [14])

```
Rule <rule name>:
  IF <condition 1> AND <condition2> .... AND <condition N>
  THEN <list of actions>.
```

According to this syntax the rule is fired only if all the conditions are satisfied.

We propose the following syntax:

```
Rule <rule name>:
  IF <condition 1>, <condition2>, ....  <condition N>
  SATISFY <predicate>
  THEN   <list     of    actions>
```

where `<predicate>` is a predicate with arity N. Every time a rule with this format is considered by the interpreter, the pattern of the truth and falsity values of the conditions (we call it the *condition-pattern*) is used as argument for `<predicate>`. If the predicates is true, the rule is fired.

This syntax allows for the definition of more general rules, thus leading to a reduction of the number and size of the rules in the system and to an easier maintenability of the knowledge base. Let us consider some examples.

First, it is worth noting that if `<predicate>` is the standard `AND` predicate (or any more complex combination of `AND`, `OR` and `NOT`), the proposed system behaves exactly as a classical rule-based system having the same conditions and actions.

However, let us suppose that `<predicate>` is the `MAJORITY` predicate (it is true if the majority of the conditions are true). Now, we can see that it is not so simple to write a set of rules equivalent to the rule:

```
Rule <rule name>:
  IF <condition 1>, <condition2>, ....  <condition N>
  SATISFY MAJORITY
  THEN <list of actions>
```

If N=3 the task may be manageable, as we should write *only* the following four rules:

```
Rule <rule name1>:
  IF <condition 1> AND <condition 2> THEN <list of actions>
Rule <rule name2>:
  IF <condition 1> AND <condition 3> THEN <list of actions>
Rule <rule name4>:
  IF <condition 2> AND <condition 3> THEN <list of actions>
Rule <rule name1>:
  IF <condition 1> AND <condition 2> AND <condition 3>
  THEN <list of actions>
```

but for larger N the task is quite tedious and potentially error prone (for any N about 2^{N-1} classical IF-THEN rules are required). This happens because the decision on whether the actions have to be performed depends on properties (the number of true conditions in this case) of the conditions as a whole, as opposed to the truth or falsity of known logical combinations. Other examples of this kind of rule-control regimes include even and odd parity, all-but-one, and multiplexing.

Other situations in which the proposed rule format can be quite useful may arise when the conditions record the current values of sensory transducers (e.g. in the control system for an industrial plant or a robot) and the actions depend on some non-Boolean combination of such values. In such cases, the calculations needed to make a decision could naturally be performed in `<predicate>`.

3 Rule Induction

Thanks to the presence of the `<predicate>` part, a simple form of constrained rule induction/knowledge acquisition can be performed. Others have obtained rule induction by decision trees (e.g. [6]) or statistical methods (e.g. [12]). We here obtain rule induction as a result of `<predicate>` induction.

3.1 Trainable Truth-table-based Predicates

As any Boolean function can be implemented via a truth table, this can also be done for `<predicate>`. This suggests a simple (though limited) form of rule induction from examples.

Let us imagine that the truth table of `<predicate>` is (maybe partially) unspecified at the beginning of the use of our rule-based system. For example, the `<predicate>` of a rule such as

```
Rule <rule name>:
  IF <condition 1>, <condition2>, <condition 3>
  SATISFY <predicate>
  THEN <list of actions>
```

could be represented by the truth table shown in Table 1, where `FIREABLE` is a value that represents the "output" of the `<predicate>` (i.e. if the rule can or cannot be fired). Then, every time the rule is considered by the interpreter the condition-pattern (e.g. the pattern [False,True,False] obtained by checking the conditions `<condition 1>`, `<condition 2>` and `<condition 3>` of the previous rule) is looked up in the truth table and the related `FIREABLE` value is considered. If such a value is True or False it is used to make `<predicate>` succeed or fail, respectively, if it is Unknown the user (at first probably the expert) is asked to provide a value for `FIREABLE`. Such a value is then stored in the truth table and used thereafter.

`<condition 1>`	`<condition 2>`	`<condition 3>`	FIREABLE
False	False	False	True
False	False	True	False
False	True	False	Unknown
False	True	True	Unknown
True	False	False	True
True	False	True	True
True	True	False	Unknown
True	True	True	Unknown

Table 1: Truth-table-based predicate implementation.

It can be argued that for applications such as real-time control or medical systems, it is much safer to have a system asking for advice in the presence of a new/unexpected situations than a system that simply ignores such situations. This mechanism can thus be thought of as an extra validation trap to catch cases that may have accidentally been missed out. If the rule base designer specifically doesn't wish to prompt the user then they are free to program the rulebase in a conventional manner.

3.2 Trainable Collection-based Predicates

The previous idea can be extended by considering predicates that instead of being implemented with a truth-table are implemented as a collection of pattern/value pairs.

Each pair includes an input pattern representing a possible set of truth and falsity values for the conditions of the rule, i.e. a possible condition-pattern, and an output value in the range [0,1] termed the `FIREABILITY` of the rule. The `FIREABILITY` value represents the level of confidence about the fact that the rule should be fired in the presence of a given condition-pattern.

Every time a rule is considered by the interpreter the actual condition-pattern is searched in the collection of pattern/value pairs. If a pair whose pattern is equal to the condition-pattern is found then the related `FIREABILITY` value is used to evaluate the `FIREABLE` value through the following formula:

$$\text{FIREABLE} = \begin{cases} True & \text{if FIREABILITY} > P, \\ False & \text{otherwise.} \end{cases}$$

where P is a parameter in the range $[0, 1]$, termed the *prudence* of the system. Once known, `FIREABLE` is used to make `<predicate>` succeed or fail as with the truth-table method. Therefore, the prudence parameter P can be used to modify the behaviour of the rules and, therefore, of the system: the greater the *prudence*, the more confidence (`FIREABILITY`) is required for an induced rule to be fired.

As in the previous method, if the condition-pattern is not found among the stored pattern-conditions, the user is asked to provide a value for `FIREABILITY`. An example of reasonable options and the related meanings is shown in Table 2.[1]

`FIREABILITY`	*Meaning*
1	"I'm sure the rule should be fired"
0.75	"Probably the rule should be fired"
0.5	"I'm not sure the rule should be fired"
0.25	"Probably the rule should NOT be fired"
0	"I'm sure the rule should NOT be fired"

Table 2: A possible set of meanings for the `FIREABILITY` value.

The condition-pattern plus the `FIREABILITY`-value provided by the user can then be stored in the collection as a new pattern/value pair and used as described previously.

3.3 Artificial-Neural-Network-based Predicates

The pattern/value pairs collected to implement a collection-based predicate can be easily used to implement artificial-neural-network-based predicates, so as to obtain hybrid symbolic/sub-symbolic systems.

[1] An alternative view is that of considering the `FIREABILITY` of a rule as a value quantifying the percentage of cases in which such rule is true, i.e. the *reliability* of the rule (e.g. 1 could mean "always valid", 0.75 "often", 0.5 "sometimes", 0.25 "seldom", 0 "never"). In such a case if the *prudence* P is high the system uses only rules that are always valid, if P is low the system also uses rules of thumb and short-cuts that are often valid and lead to quicker (though less certain) conclusions (default reasoning).

The idea is as follows. If the pattern parts of the collected pattern/value pairs are transformed into binary vectors (False→0, True→1), then such pairs can be used as examples for a neural network. The weights and bias of the network can be adapted via a learning mechanism (e.g. the back-propagation rule) so that the network learns to behave according to the examples. Afterwards, in the presence of a given condition-pattern the output of the network can be taken to be the `FIREABILITY` of the rule and used accordingly.

It should be noted that in cases where one can rely on the properties of generalisation, noise rejection, input rectification, etc. of a neural network, the network-based predicate would behave sensibly also in the presence of new or partially inconsistent condition-patterns, without requiring the intervention of the user.

This kind of approach lends itself to several extensions. Firstly, if an example that exactly matches the condition-pattern is present, the related values could be used instead of using the neural network, so as to exactly behave like the user/expert in known cases. Secondly, the user could still be asked to provide an answer whenever the `FIREABILITY` value provided by the network is in an intermediate range such as [0.4,0.6], so as to make the system maximally reliable. Third, floating-point certainty values in the range [0,1] could be associated with the conditions, instead of True and False values only, for instance reflecting uncertainty of beliefs, or values of sensory devices. A neural network could learn the (possibly complex) relations between the resulting real-valued condition-pattern and the corresponding `FIREABILITY` values. Rules including this kind of mechanism could (learn to) treat uncertainty in a simple and natural way.

3.4 Vector Predicates and Extended Syntax

In the previous sections we have seen how our predicate-based rule-syntax allows for a considerable reduction of the number of rules when several (classical) rules share the same list of actions.

However, it is quite common when writing a set of rules, that several of them share the same conditions (although combined via different sets of AND, OR and NOT operators, in the classical formalism, or different predicates in our formalism) but have different actions. It would be desirable to be able to represent such rules with a single rule. In addition quite often the knowledge engineer wants that only one rule of the set fires (e.g. because the actions are mutually exclusive): this can be handled only with procedural hand coding or clever conflict-resolution mechanisms. A more natural distributed conflict resolution mechanism would be desirable.

Even if it is quite simple to solve the problem of mutually exclusive rules with the previously described syntax (e.g. by using mutually related predicates), representing more concisely rules with the same conditions requires an extension to the formalism. Fortunately, this is quite a natural extension.

The extended syntax is the following:

```
Rule <rule name>:
  IF <condition 1>, <condition2>, ....  <condition N>
```

```
SATISFY <vector predicate>
THEN <list of actions 1>, ...,<list of actions M>
```

where `<vector predicate>` is a vector extension of the concept of predicate, i.e. it is a function that transforms the domain $\{\text{True}, \text{False}\}^N$ into the domain $\{\text{True}, \text{False}\}^M$. The previous rule has to be interpreted as follows. Given a condition-pattern, `<vector predicate>` is invoked. It returns a list, called *action-pattern*, containing M True or False values. The lists of actions for which a True value has been returned are added to the list of fireable actions to be considered by the conflict resolution mechanism; the other lists of actions are ignored. It should be noted that with this formalism it is quite easy to represent a large number of rules with a single one, and to implement any form of distributed conflict resolution.

All the methods of predicate induction described in the previous sections can be extended for vector predicates. For truth-table based predicates it is sufficient to include as many `FIREABLE` columns as the number of lists of actions M. Likewise we can extend collection-based and artificial-neural-network-based predicates. In all cases, during the training (or example-collection) phases, the user will be asked to provide a vector of M `FIREABLE` or `FIREABILITY` values.

3.5 Predicate Induction without User Interaction

The previous mechanisms for inducing the `<predicate>` of a rule from examples, require the interaction with the user or the expert(s). Sometimes, it might be desirable not to. For example, there may be situations in which it is known that a certain set of conditions are relevant to undertake an action or to assert a conclusion but there is no easy way of deciding if, for a given sub-set of conditions that are true, the rule should be fired. This can happen when such a decision requires performing boring or complex calculations, visual comparisons, or expensive consultations with experts.

In such cases, an alternative, although computationally expensive, predicate-induction mechanism not requiring rule-by-rule examples is possible. It is based on well-known optimisation techniques known as Genetic Algorithms (GAs) [3].

Let us consider for the sake of simplicity only the problem of finding the truth tables needed for a set of K rules each one including N_i, $i = 1, \cdots, K$, conditions and M_i actions. Each truth table includes $2^{N_i} \times M_i$ `FIREABLE` entries, that can be represented as a bit string provided that we assign the value 1 to True and 0 to False (or vice versa). By chaining the bit strings that represent all the K truth tables we obtain a longer bit string including $\sum_i 2^{N_i} \times M_i$ bits. Each instance of such a bit string represents a different set of rules and therefore a different behaviour of the rule-based system.

If we define an objective (or fitness) function that scores each possible instance of the strings representing different rule-based systems, we can then easily apply a GA for finding the best rule base according to the scoring criterion. One possible fitness function could be, for example,

$$f(R) = \sum_j [A(D_j) + \lambda \times C(D_j)]$$

R being a given rule-base, D_j a database of facts, $C(D_j)$ the computation required to reach a conclusion for database D_j, $A(D_j)$ the accuracy of the conclusions obtained with database D_j and λ a constant factor (computation and accuracy need an operative definition which is problem dependent).

Extensions of this method to the other predicate-induction paradigms are straightforward. For example it is easy to encode the weights and the biases of a neural net into the bit strings that undergo genetic optimisation so as to induce neural-network-based predicates.

4 Experimental Results

We have implemented and used the previously described formalism and the related rule induction mechanisms in several experiments, some of which are described in the following subsections. The first set of experiments (Section 4.1) has been performed merely to provide the reader with simple examples that show some of the features of our approach; the others (Sections 4.2 and 4.3) are more complicated examples that show the advantages of our approach in practical applications.

4.1 Animal Classification

In this subsection we report on some basic experimental results obtained with the well-known rule base for feature-based recognition of animals described in [14, pages 121–124]. The rule-base is summarised in Table 3. From the set of facts shown in Table 4(a) which describes an animal, the rules are capable of inferring that the animal is a giraffe (via rules 1, 8 and 11).

Writing the previous set of rules requires that a clear, complete knowledge on how to classify an animal is available. In more complex situations/domains it might happen that only partial knowledge is available. For example, even if it is known that certain conditions are relevant to draw a given conclusion, it might be difficult to list all the combinations of conditions for which such a conclusion can be drawn (e.g. because it is usually drawn on the grounds of the experience of the expert). This would make writing rules with the standard formalism quite difficult.

To show how similar situations can be handled by our system, let us consider the animal classification problem and suppose that when the rule base was compiled it was not completely clear how to classify an animal as a bird, a carnivore or an ungulate even if all the relevant information that might be important for doing so were known. In such a case we could write the rule base shown in Table 5 where Predicate-A, Predicate-B and Predicate-C are three trainable predicates.

4.1.1 Truth-table-based Predicate Induction

Let us first consider a run of the system with the set of facts listed in Table 4(a) when a truth-table-based implementation of the trainable pre-

Rule #	*Conditions*	*Conclusion*
1	Has hair	Is a mammal
2	Gives milk	Is a mammal
3	Has feathers	Is a bird
4	Flies AND Lays eggs	Is a bird
5	Is a mammal AND Eats meat	Is a carnivore
6	Is a mammal AND Has pointed teeth AND Has claws AND Has forward pointing eyes	Is a carnivore
7	Is a mammal AND Has hoofs	Is an ungulate
8	Is a mammal AND Chews cud	Is an ungulate
9	Is a carnivore AND Has tawny colour AND Has dark spots	Is a cheetah
10	Is a carnivore AND Has tawny colour AND Has black stripes	Is a tiger
11	Is an ungulate AND Has long legs AND Has long neck AND Has tawny colour AND Has dark spots	Is a giraffe
12	Is an ungulate AND Has white colour AND Has black stripes	Is a zebra
13	Is a bird AND Does not fly AND Has long legs AND Has long neck AND Is black and white	Is an ostrich
14	Is a bird AND Does not fly AND Swims AND Is black and white	Is a penguin
15	Is a bird AND Is a good flyer	Is an albatross

Table 3: Winston's rule base for animal classification.

dicates is used. The predicates have not been trained at all in advance. The following events take place:[2]

1. Rule 1 is fired and the fact that the animal "is a mammal" is added to working memory.

2. Rule 2 is tested. Its three conditions are all False, so that Predicate-A is invoked with the condition-pattern [False,False,False]. As the related `FIREABLE` value is unknown, the user is asked to provide such a value. As there is no evidence that the animal is a bird, the user replies False, Predicate-A fails and the rule is not fired.

3. Rule 3 is tested. The condition-pattern is [True,False,False,False,False]. However, as the fact that an animal is a mammal does not necessarily mean that it is a carnivore, the user provides again a False for the related `FIREABLE` value, Predicate-B fails and the rule is not fired.

4. Rule 4 is tested. The condition-pattern is [True,False,True]. The user in this case provides a True `FIREABLE` value, Predicate-C succeeds, the rule is fired and the fact that the animal "is ungulate" is added to the working memory.

5. Rules 5 and 6 are considered but not be fired.

[2]For the sake of simplicity no conflict resolution mechanism is considered in this example: the rules are considered in their order, and all the fireable rules are fired.

Facts
Has hair
Chews cud
Has long legs
Has a long neck
Has tawny colour
Has dark spots

(a)

Facts
Does not fly
Has tawny colour
Has black strips
Eats meat
Has hair

(b)

Facts
Has hair
Has tawny colour
Has dark spots
Has pointed teeth
Has claws

(c)

Table 4: Some possible initial working memories.

Rule #	*Conditions*	*Predicate*	*Conclusion*
1	Has hair, Gives milk	OR	Is a mammal
2	Has feathers, Flies, Lays eggs	Predicate-A	Is a bird
3	Is a mammal, Eats meat, Has pointed teeth, Has claws, Has forward pointing eyes	Predicate-B	Is a carnivore
4	Is a mammal, Has hoofs, Chews cud	Predicate-C	Is an ungulate
5	Is a carnivore, Has tawny colour, Has dark spots	AND	Is a cheetah
6	Is a carnivore, Has tawny colour, Has black strips	AND	Is a tiger
7	Is an ungulate, Has long legs , Has long neck, Has tawny colour, Has dark spots	AND	Is a giraffe
8	Is an ungulate, Has white colour, Has black stripes	AND	Is a zebra
9	Is a bird, Does not fly, Has long legs, Has long neck, Is black and white	AND	Is an ostrich
10	Is a bird, Does not fly, Swims, Is black and white	AND	Is a penguin
11	Is a bird, Is a good flyer	AND	Is an albatross

Table 5: A rulebase for animal classification.

6. Rule 7 is fired and the animal is classified as a giraffe.

After this first run the system has acquired some experience on when rules 2, 3 and 4 should be fired, so that running the system again with the same working memory would result in a completely automated answer (i.e. without user interaction). In order to complete the elicitation of the knowledge needed to fire rules 2, 3 and 4 other examples are needed. However, the more the cases presented to the system, the lower the number of subsequent interactions with the user.

For example, if after the previous run, the system is run again with the working memory shown in Table 4(b), then the following set of events take place: rule 1 fired, rule 2 checked (no user interaction), rule 3 checked (user interaction) and fired, rule 4 checked (user interaction), rule 5 checked, rule 6 fired. In this case the animal is classified (correctly) as a tiger with only two user interactions.

4.1.2 Collection-based Predicate Induction

In order to show some of the properties of collection-based predicates, let us run the previous rule base, with the initial set of facts shown in Table 4(c) that forms a partial description of a cheetah (the fact "Has forward pointing eyes" is lacking). However, let us hypothesise that now Predicate-A, Predicate-B and Predicate-C are implemented via a collection of condition-pattern/`FIREABILITY`-value pairs. In addition, let us set the prudence $P = 0.99$, so that only if `FIREABILITY` is near 1 can trainable-predicate-based rules be fired.

The following events take place:

1. Rule 1 is fired and the fact that the animal "is a mammal" is added to the working memory.

2. Rule 2 is tested. The condition-pattern is [False,False,False]. As no example is present in the collection, the user is asked to provide a `FIREABILITY` value. As there is no evidence that the animal is a bird, the user provides a 0, Predicate-A fails and the rule is not fired.

3. Rule 3 is tested. The condition-pattern is [True,False,True,True,False]. In this case the user is quite confident that the rule should be fired as the animal has most of the features of a carnivore. However, as a condition "Has forward pointing eyes" is lacking, the user provides a 0.8 for the `FIREABILITY`. As $P = 0.99$, Predicate-B fails and the rule is not fired.

4. Rule 4 is tested. The condition-pattern is [True,False,True]. The user in this case is not sure: the fact that an animal is a mammal does not necessarily mean that it is an ungulate. So the user provides a 0.5, Predicate-C fails and the rule is not fired.

5. None of the remaining rules can be fired and therefore no conclusions about the species of the animal can be drawn.

The failure to reach a conclusion in this example is due to the incomplete information available about the animal.

Let us now suppose to reduce the prudence of the system to the value $P = 0.7$, and run the system again. The sequence of events is the same apart from the following facts: a) no user interaction is now required, b) rule 4 is now fired so that the fact "Is a carnivore" is added to the working memory, c) thanks to this additional information rule 5 can fire and the (correct) conclusion "Is a cheetah" is reached.

Reducing further the prudence of the system in general can lead to conclusions that are alternative to each other (multiple solutions, multiple answers). For example, by setting $P = 0.4$ the conclusions that the animal "Is an ungulate" and "Is a carnivore" are both drawn. This is not a mistake: in the lack of additional information they should be interpreted as two reasonable hypotheses about the animal. In this example this behaviour does not produce any final misclassification. In other cases it might lead to multiple classifications.

4.2 Robot Motor Control

In order to show the advantages of using predicates based on artificial neural networks we now consider a more complex example. The problem consists of designing a set of rules that allow a robot to react properly in the presence of obstacles.

The robot has engines and wheels that allow it to move in (at least) four main directions: forward, backward, left and right. It has eight proximity sensors that reveal the presence of obstacles at 0°, 45°, 90°, ... 315° (clockwise with respect to its current heading). As 2 or more sensors can detect the presence of obstacles for any given position (e.g. because the robot is in a corner or in a tight corridor), writing a set of rules for the four possible motor actions could be relatively difficult with a non-trainable rule-based system.

Instead, with our formalism, we need only the following rule

```
Rule Obstacle-Avoidance:
IF Obstacle at 0 degrees, Obstacle at 45 degrees,
   Obstacle at 90 degrees, Obstacle at 135 degrees,
   Obstacle at 180 degrees, Obstacle at 225 degrees,
   Obstacle at 270 degrees, Obstacle at 315 degrees
SATISFY Obstacle-Predicate
THEN Go forward, Go backward, Go left, Go right
```

where Obstacle-Predicate is a vector predicate implemented via a neural network plus a winner-takes-all output filter that prevents more than one True value being present in the action-pattern (the output of the vector predicate).

Before learning, each time the vector predicate is called an example is collected including the present condition-pattern and four `FIREABILITY` values, one for each action. After enough examples have been collected the network is trained and thereafter used. In this experiment we have used as the training set 8 condition-patterns including only one True value (only one sensor has detected a collision) and 17 condition-patterns including two True values (a total of 25 examples out of the 256 possible condition-patterns).

After learning, as soon as the predicate is invoked the network is used to evaluate the four `FIREABILITY` values needed to compute the action-pattern. If the maximum `FIREABILITY` is greater then the prudence P of the system the predicate succeeds and an action-pattern with a single True item is provided; otherwise the predicate fails.

Figure 1 reports on some experimental results with this scenario. The lower part of the figure shows six different situations, labeled (a)–(f), in which the robot (represented as an octagon) can collide with obstacles. The table in the upper part of the figure reports the sensory conditions for each situation and the correspondingly selected action. The local conflict resolution implemented via the winner-takes-all filter prevents more than one action being fireable. The generalisation properties of the neural net provide the correct behaviour even in the presence of new condition-patterns containing two or more True values (situations (c)–(f)). A variant of this rule (with a different syntax which is functionally equivalent to the one presented in this paper) has been used as a behaviour sub-system for the subsumptive architecture of a simulated robot, as described in [10].

	Conditions								*Actions*			
Exp.	0°	45°	90°	135°	180°	225°	270°	315°	Forward	Backward	Left	Right
(a)	×									×		
(b)	×	×								×		
(c)	×	×	×								×	
(d)	×	×	×	×							×	
(e)	×	×	×	×	×						×	
(f)	×	×		×	×						×	

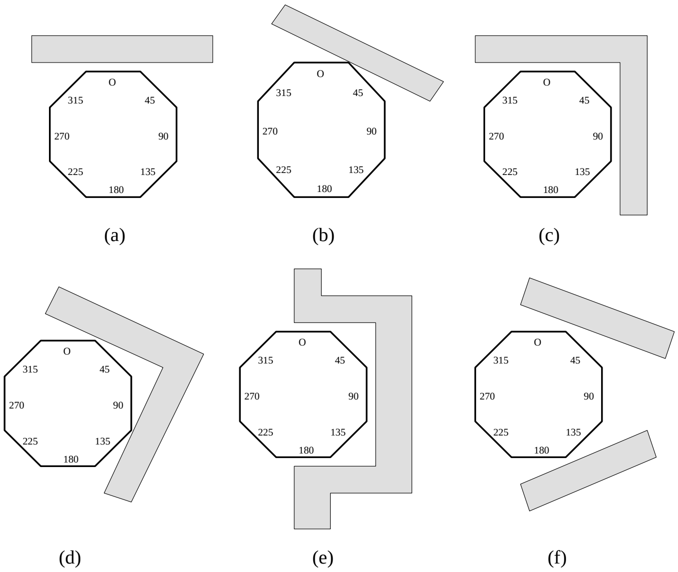

Figure 1: Experimental results for robot obstacle avoidance.

4.3 Inter-Agent Communication

The GA-based rule induction mechanism described in Section 3.5 has been (and it is currently) used, in more complicated experiments, for the development of communication between agents. In the following we describe the simplest of these experiments (the experiment has been implemented using the toolkit for building and running agents described in [10]).

The experiment involves two agents: a blind agent and a lazy one. The blind agent can move in the world (the 2-D Cartesian plane) and can receive messages from the other agent, but is not capable of "visually" perceiving it. The lazy agent can perceive the (roughly quantised) relative position of the blind agent and can send messages, but cannot move. Both agents are implemented via a small set of rules the most important of which are:

```
Rule Lazy-Message-Transmission:
  IF Blind is north, Blind is south, Blind is east, Blind is west
  SATISFY Lazy-GA-predicate
  THEN Send message 1, Send message 2, Send message 3, Send message 4
```

```
Rule Blind-Message-Interpretation:
  IF Received message 1, Received message 2,
     Received message 3, Received message 4
  SATISFY Blind-GA-predicate
  THEN Go east, Go west, Go north, Go south
```

where `Lazy-GA-predicate` and `Blind-GA-predicate` are two vector predicates implemented via truth tables. The objective of the experiment is to induce such predicates (by means of a genetic algorithm) so as to obtain a cooperation (via communication) between the two agents that results in the blind agent moving towards and finally reaching the lazy one, for any possible initial mutual positions.

In the simulation message sending and position perception are obtained by a scheduler that repeatedly runs the rulebase of each agent and changes the database of each one so as to include new sensory data or new messages (clean-up rules remove the old data from each database).

In order to represent the truth tables of the aforementioned rules bit strings including 128 bits are needed. Each of such bit strings represents a possible rule-base R. We associate to each R a fitness value $f(R)$ which is the negative of the sum of the distances between the blind and the lazy agents measured at the end of four different runs of the simulation. In each run the lazy and the blind agents start in different relative locations (the furthest corners of a square).

The truth tables induced by running a GA with a population of 20 rule-sets (bit strings) for 50 iterations are shown in Tables 6 and 7.[3] The corresponding rules provide the two agents with the required behaviour, i.e. they meet each other within a minimum number of steps. An example of such a behaviour is shown in Figure 2. The figure is a graphical representation of a run (not included in the training set) in which the lazy agent (the circle marked with a dot) was at $(0.2, 0.3)$ and the blind one started from $(0.9, 0.7)$. Note how the blind agent finds the shortest route to the lazy agent.

5 Conclusions

In this paper we have presented a formalism for rule representation that lends itself to rule induction and the development of hybrid systems in which neural networks and symbolic techniques cooperate tightly to produce robust, flexible, trainable and efficient architectures.

On the grounds of the experimental results, we believe that the proposed model is not "yet-another-rule-based system" but offers new powerful solutions for problems in which knowledge is uncertain, inconsistent, incomplete or variable.

[3] Not all the entries of these truth tables are meaningful as not all the combinations of conditions can actually occur (e.g. the blind agent cannot be north and south of the lazy one at the same time).

Conditions				*Actions*			
Blind is north	Blind is south	Blind is east	Blind is west	Send message 1	Send message 2	Send message 3	Send message 4
						×	×
			×	×			×
		×		×			×
		×	×		×	×	×
	×			×		×	×
	×		×		×		
	×	×					×
	×	×	×			×	×
×				×		×	
×			×		×	×	×
×		×				×	
×		×	×	×			
×	×				×	×	
×	×		×		×	×	×
×	×	×		×	×		×
×	×	×	×	×			×

Table 6: Truth table representing `Lazy-GA-predicate`.

Acknowledgements

The authors wish to thank Professor Aaron Sloman and the Cognition and Affect Group of the School of Computer Science, The University of Birmingham, for useful suggestions and help in developing the system described in this paper.

References

[1] A. Clark. *Microcognition.* MIT Press, 1989.

[2] J. A. Fodor and Z. W. Pylyshyn. Connectionism and cognitive architecture: a critical analysis. In S. Pinker and J. Mehler, editors, *Connections and Symbolcs*, pages 3–71, Cambridge:MA, 1988. The MIT Press.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[4] E. Motta, M. Eisenstadt, M. West, K. Pitman, and R. Evertsz. Keats: The knowledge engineers assistant. *Expert Systems: The International Journal of Knowledge Engineering*, 1988.

[5] R. Poli, S. Cagnoni, R. Livi, G. Coppini, and G. Valli. A neural network expert system for diagnosing and treating hypertension. *IEEE Computer*, 24(3):64–71, 1991.

Conditions				*Actions*			
Received message 1	Received message 2	Received message 3	Received message 4	Go east	Go west	Go north	Go south
						×	×
			×		×	×	
		×			×		×
		×	×	×		×	
	×			×		×	
	×		×				×
	×	×		×	×	×	×
	×	×	×	×			×
×				×	×	×	
×			×		×	×	×
×		×		×	×	×	
×		×	×	×		×	
×	×			×	×	×	
×	×		×				×
×	×	×		×			×
×	×	×	×	×			×

Table 7: Truth table representing `Blind-GA-predicate`.

[6] J. R. Quinlan. Discovering rules by induction from large collections of examples. In D. Michie, editor, *Expert systems in micro-electronics age*. Edinburgh University Press, Edinburgh, Scotland, 1979.

[7] C. Rich. Cake: An implemented hybrid knowledge representation and limited reasoning system. *SIGART Bulletin*, 2(3):120–127, 1991.

[8] J. F. Schreinmakers. *Pattern Recognition and Symbolic Approaches to Diagnosis*. Eburon:Delft, The Netherlands, 1991.

[9] L. Shastri. A connectionist approach to knowledge representation and limited inference. *Cognitive Science*, pages 331–392, 1988.

[10] A. Sloman and R. Poli. SIM_AGENT: A toolkit for building agents. Technical report, School of Computer Science, The University of Birmingham, 1995.

[11] P. Smolensky. Connectionist AI, symbolic AI, and the brain. *AI Review*, 1990.

[12] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Transaction on Knowledge and Data Engineering*, 4(4):301–316, 1992.

[13] D. S. Touretzky and G. E. Hinton. A distributed connectionist production system. *Cognitive Science*, 1988.

[14] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition, 1992.

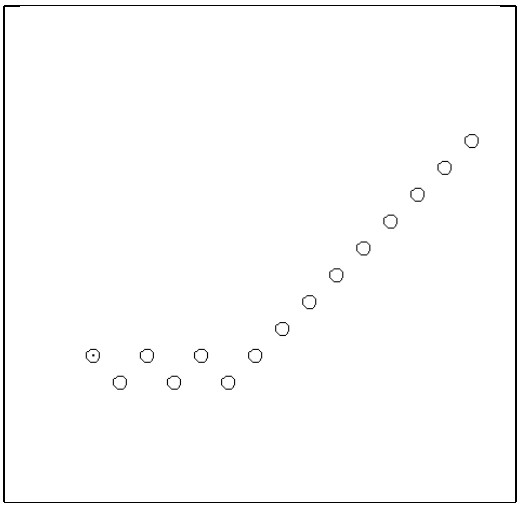

Figure 2: A run of the communication induction experiment.