A Genetic Approach to the Automatic Generation of Fuzzy Control Systems from Numerical Controllers

Giuseppe Della Penna[1], Francesca Fallucchi[3], Benedetto Intrigila[2], and Daniele Magazzeni[1]

[1] Department of Computer Science
University of L'Aquila, Italy
[2] Department of Mathematics
University of Roma "Tor Vergata", Italy
[3] DISP, University of Roma "Tor Vergata", Italy

Abstract. Control systems are small components that control the behavior of larger systems. In the last years, sophisticated controllers have been widely used in the hardware/software *embedded systems* contained in a growing number of everyday products and appliances. Therefore, the problem of the automatic synthesis of controllers is extremely important. To this aim, several techniques have been applied, like *cell-to-cell mapping*, *dynamic programming* and, more recently, *model checking*. The controllers generated using these techniques are typically *numerical controllers* that, however, often have a huge size and not enough robustness. In this paper we present an automatic iterative process, based on *genetic algorithms*, that can be used to compress the huge information contained in such numerical controllers into smaller and more robust *fuzzy control systems*.

1 Introduction

Control systems (or, shortly, *controllers*) are small hardware/software components that control the behavior of larger systems, the *plants*. A controller continuously analyzes the plant state (looking at its *state variables*) and possibly adjusts some of its parameters (called *control variables*) to keep the system in a condition called *setpoint*, which usually represents the *normal* or *correct* behavior of the system.

In the last years, the use of sophisticated controllers has become very common in robotics, critical systems and, in general, in the hardware/software *embedded systems* contained in a growing number of everyday products and appliances.

Therefore, the problem of the automatic synthesis of control systems starting from the plant model is extremely important. This problem is particulary difficult for *non-linear systems*, where the mathematical model of the plant is not analytically tractable. To this aim, several techniques have been developed, based on a more or less systematic exploration of the state space. One can mention, among others, *cell-to-cell mapping* techniques [1] and *dynamic programming* [2].

R. Basili and M.T. Pazienza (Eds.): AI*IA 2007, LNAI 4733, pp. 230–241, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Recently, *model checking* techniques have also been applied [3, 4] in the field of automatic controller generation. In particular, this approach can be actually considered as a *planning* technique. Indeed, a model checking-based controller generator does not simply look for good *local* actions towards the setpoint, but searches for *the best possible sequence of actions* to bring the plant to the setpoint. Therefore, with this technique it is possible to find an *optimal* solution to the control problem.

The controllers generated using all these techniques are typically *numerical controllers*, i.e. tables indexed by the plant states, whose entries are commands for the plant. These commands are used to set the control variables in order to reach the setpoint from the corresponding states. Namely, when the controller reads a state from the plant, it looks up the action described in the associated table entry and sends it to the plant. However, this kind of controllers can present two main problems.

The first problem is the *size* of the table, which for complex systems may contain millions of entries, since it should be embedded in the control system hardware that is usually very limited.

The second problem is the controller *robustness*. A controller is robust if it is able to handle all the possible plant states. Due to approximation of continuous variables, plants unavoidably present states that are not known to the controller, although they may be more or less *close* to some states in the table. In numerical controllers this problem is typically handled by interpolation techniques (e.g., see [2]). However, interpolation does not always work well [1]: table based control may also give a bumpy response as the controller jumps from one table value to another. Therefore, other approaches have been proposed (e.g. see [5, 6]).

A natural solution to these problems is to derive, from the huge numerical information contained in the table, a small *fuzzy control system*. This solution is natural since fuzzy rules are very flexible and can be adapted to cope with any kind of system. Moreover, there are a number of well-established techniques to guide the choice of fuzzy rules by statistical considerations, such as in Kosko space clustering method [7], or by abstracting them from a neural network [8]. In particular, the approach that we have adopted in the present paper is inspired by [1]. However, there are several substantial differences.

The crucial point is of course the algorithm to extract the fuzzy rules from the numerical controller table. We used *genetic algorithms* [9]. This choice is based on the following considerations:

- genetic algorithms are suitable to cope with very large state spaces [9]; this is particularly important when the starting point is the huge table generated by model checking techniques;
- the *crossover* mechanism ensures a fair average behavior of the system, avoiding irregularities;
- the *fitness function*, which is the core of any genetic algorithm, can be obtained in a rather direct way from the control table of the numerical controller;
- the genome coding can also be derived from the structure of the sought-for fuzzy system.

Therefore, the process is almost automatic. Only a few parameters, such as the number of fuzzy sets, have to be determined by hand. However, also the choice of such parameters can be automatized or at least supported by an automatic process. Indeed, the correctness of the resulting fuzzy controller can be verified [10], so in case of a poor behavior the parameters are changed and the process restarted. This automatic loop is stopped when the right values for the parameters are detected. Of course [11], it is possible that no such values exist, that is the system is so complex that the controller table turns out to be *incompressible*.

The paper is organized as follows. In Section 2 we present the numerical control systems and we summarize the methods for their synthesis, while in Section 3 we describe the fuzzy control systems. In Section 4 we give an overview of genetic algorithms and in section 5 we describe how we use them to automatically synthesize fuzzy controllers. In Section 6 we present a case study and experimental results. Section 7 concludes the paper.

2 Numerical Control Systems

As mentioned in the Introduction, a *numerical controller* is a table, indexed by the plant states, whose entries are commands for the plant.

The use of such kind of controller is very suitable (and often necessary) to cope with non-linear systems, which have a dynamics too complex to allow an analytical treatment [3, 4]. On the other hand, the table size of a numerical controller could be huge, especially when we are interested in the efficiency of controller and thus we use a high precision. In these cases, we could have tables containing millions of state-action pairs and if we are working with small embedded systems, the table size could be a potential issue.

2.1 Numerical Control System Synthesis

There are a number of well-established techniques for the synthesis of numerical control systems. For short, we mention only three of them: (1) dynamic programming, (2) cell mapping and (3) model checking.

The classical dynamic programming approach for the synthesis of controllers of a plant P (see [2] for details) is based on an optimal cost function J defined as follows:

$$J(x) =_{def} \inf_{\underline{u}} \left[\sum_{t=0}^{\infty} l(f(x, u_t), u_t) \right] \quad (1)$$

where $f(x, u)$ is the continuous dynamics of the plant, $l(x, u)$ is a continuous, positive definite *cost function* and $\underline{u}$ stands for a generic control sequence: $\underline{u} = \{u_0, u_1, u_2, \ldots\}$.

J is well defined (i.e. the infimum always exists in the region of interest) if and only if the plant P is controllable. So, assuming that J is well defined, then it satisfies the so-called *Bellman Equation*:

$$J(x) = \inf_u \; [l(x,u) \; + \; J(f(x,u))] \tag{2}$$

and it can be computed by the following iterative method:

$$\begin{array}{c} J_{T+1}(x) = \inf_u [l(x,u) + J_T(f(x,u))] \\ J_0 =_{def} 0, T \; \in \mathbf{Z}_0^+ \end{array} \tag{3}$$

In the cell mapping method [1], the trajectories (i.e. sequences of state-control pairs) in the continuous space are converted to trajectories in the discrete cell state space. The discrete cells have rectangular shape and each point of the continuous space is represented with the center of the cell containing the point itself. Then, the dynamic $f(x,u)$ of the plant is transformed into a dynamic f_C in the discrete cell space. The image of a cell under f_C, can be determined as follows: for a given cell $z(k)$, first find the coordinates of its center $x(k)$. Under control action u, $x(k+1)$ is determined as the image of $x(k)$ by the plant dynamic, that is $x(k+1) = f(x(k),u)$. If the cell corresponding to the point $x(k+1)$ is $z(k+1)$, then $z(k+1)$ is the image cell of the cell $z(k)$ and the control action u, that is we put $f_C(z(k),u) = z(k+1)$.

Finally, we recall how model checking techniques can be applied for the synthesis of controllers. This kind of methodology allows to *automatically* synthesize *optimal* controllers starting from the plant description [3, 4]. The main idea is that, in order to build a controller for a plant P, a suitable discretization of the state space of P is considered, as well as of the control actions u.

The plant behavior, under the (discretized) control actions, gives rise to a transition graph $\mathcal{G}$, where the nodes are the reachable states and a transition between two nodes models an allowed control action between the corresponding states. In this setting, the problem of designing the optimal controller reduces to finding the minimum path in $\mathcal{G}$ between each state and the nearest *goal state* (a discretization of the setpoint). Clearly, a transition graph for complex, real-world systems could be often huge, due to the well-known *state explosion* problem. However, model checking techniques developed in the last decades have shown to be able to deal with very huge state spaces. In particular, in [3, 4] a model checking based methodology for the automatic synthesis of optimal controllers is presented, and it is also shown that the methodology can cope with very complex systems (e.g. the truck-trailer obstacles avoidance parking problem). Finally, the methodology presented in [3, 4] has been implemented in the CGMurphi tool [12] that, given a model of the plant, automatically generates a controller.

Note that, in the case study of this paper, we consider the numerical controller generated with the CGMurphi tool.

3 Fuzzy Control Systems

Fuzzy logic derives from the fuzzy set theory and is used to deal with approximate reasoning. In the fuzzy set theory, the set membership is expressed by a number usually ranging from 0 to 1, indicating different “membership degrees”.

In other words, an element may have a partial membership in many different (and disjoint) fuzzy sets.

In the same way, in the fuzzy logic there are several degrees of truth and falsehood, and a fuzzy logic statement may be at the same time partially true and partially false. Indeed, there may be several conflicting fuzzy logic statements that are satisfied by the same conditions with a different "degree of truth".

As a typical application, fuzzy logic is used to deal with systems described by continuous variables (e.g., physical systems), where the density of the domain and the consequent approximation problems make it difficult to express the *certainty* required by the classical logic. Therefore, fuzzy logic is very suitable to be applied in the control theory, especially when dealing with *hybrid systems*, where the plant state is characterized by both continuous and discrete variables, and in general systems that are subject to control and actuation errors (e.g., mechanical systems) too complex to allow an analytical treatment [13].

Fuzzy control systems (FCS in the following) are based on qualitative fuzzy rules which have the form "**if** *condition* **then** *control action*", where both *condition* and *control action* are formulated making use of the so called "linguistic variables", which have a qualitative, non mathematical character [14].

In a FCS, the *crisp* input variables read from the plant state are mapped into the "linguistic variables" through the *fuzzification* process. The input variables' domains are divided in (possibly overlapping) subranges, and particular (fuzzy) membership functions are used to determine the degree of membership of each variable to all the subranges of the corresponding domain. Then, the controller makes its decision using the fuzzy rules on the fuzzified input values, and generates a set of fuzzy values for the output (control) variables, that are then converted back into crisp values and sent to the plant.

FCS are very effective in handling "uncertain" or "partially known" situations, and therefore may be used to build very robust controllers for such kind of complex systems. Moreover, FCS are well suited to low-cost implementations based on limited devices, since the fuzzy knowledge representation is usually very compact. Note that, in many cases, FCS can also be used to improve existing controller systems, for example by adding an extra layer of intelligence and robustness to the control algorithm.

The design of a FCS is usually accomplished by "translating" into fuzzy sets and inference rules the knowledge derived from human experts or mathematical models. Unfortunately, this process may be often difficult and error-prone. From this point of view, designing an effective controller at a reasonable cost is the real problem to deal with in the FCS field.

To this aim, different approaches have been studied to automatically generate a FCS by analyzing the plant specifications and/or behavior. As shown in [1], a FCS can be also generated from a numerical controller. These approaches are all based on some kind of automatic statistical analysis, which can be done using various techniques, including neural networks and genetic algorithms [15, 16, 17, 11]. In the present paper, we use genetic algorithms.

4 Genetic Algorithms

Genetic algorithms (GA) are used to find approximate solutions to optimization and search problems, using techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover [9].

Generally speaking, in a GA a population of abstract representations of possible solutions (*individuals*) to a problem evolves to find better solutions. The information carried by each individual is called *genome*. The evolution develops through a sequence of steps (*generations*), usually starting from a population of randomly generated individuals. In each generation, the algorithm estimates the *fitness* of every individual in the population, which represents the quality of the problem solution encoded in its genome. If an individual fitness reaches the given threshold, the evolution ends and the corresponding solution is returned. Otherwise, some individuals are probabilistically selected from the current population, so that the higher is the their fitness, the higher is the probability of being selected. The genomes are recombined and possibly mutated to form the population of the next generation.

Therefore, a typical genetic algorithm requires a minimal startup information: a representation of the solution domain in terms of genomes and a fitness (quality) function of the solution domain

Note that no initialization data is usually required, and this makes GA very suitable for problems having no known approximate solutions.

GAs are often able to quickly locate good solutions, even in difficult search spaces. Moreover, such algorithms are very suitable to search irregular solution spaces, since they do not usually get trapped by *local optima*.

Therefore, GA may be a very effective good tool to automatically synthesize FCS. However, as with all machine learning processes, many parameters of a GA should be tuned to improve the overall efficiency. These parameters include the population size and the mutation/crossover probabilities. Moreover, a good implementation of the fitness function considerably affects the speed and efficiency of the algorithm. Due to the difficulty of tuning such parameters by hand, some systematic process of trial-and-error should be used, as mentioned in the Introduction. We plan to consider this aspect in a future work.

5 GA for Automatic Synthesis of the FCS

Many works in the literature use GA to generate part of a FCS. Usually, GA are applied to generate membership functions when inference rules are known [11]. However, when the system under control has several control variables and/or a complex dynamics, deciding the control setting to obtain a desired result may be difficult, even when working with the probabilistic approximation of fuzzy values. Therefore, an effective FCS generator should create both membership functions and inference rules.

The concrete implementation of the fitness function is determined by the kind of FCS that the GA should generate. Indeed, we may want to

1. create a FCS for the entire plant completely *from scratch*, or
2. create one or more FCS to assist a pre-existing plant controller in certain critical situations, or
3. create one or more FCS to encode the complete knowledge of a pre-existing controller.

The first task is very difficult to accomplish in a general setting, whereas the second and third are usually applied to add robustness to a given controller. The third kind of FCS has a further advantage: besides building a more robust controller, it may act as a *controller compressor*. Indeed, the knowledge encoding given by a FCS can greatly decrease the amount of memory needed to store the controller.

Moreover, in the first two cases above a plant simulator is needed to check the fitness of the generated FCS, whereas in the third case the fitness function can be simply derived from the the original controller.

Therefore, in the following we will discuss how GA can act on a numerical controller to transform it in a compact and robust FCS.

5.1 Implementation

To make the process almost automatic we have made some simplifications. In particular, we have supposed that the membership function is always triangular and it is coded by the position of its vertices. So, each fuzzy set is represented by 3 bytes as shown in Fig. 1.

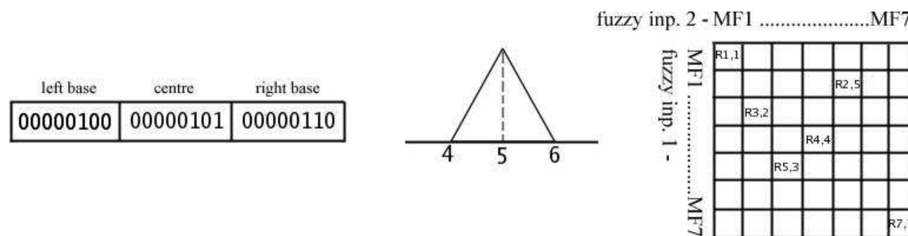

Fig. 1. Fuzzy sets and rules coding

As usual, a matrix codes the structure of the fuzzy rules [11] . An example is shown in Fig. 1.

The fuzzy sets and the fuzzy rules, encoded as described above, form the genome of any individual of the population, as shown in Fig. 2, where MF1... MF7 are the membership functions of a FCS.

Therefore, each individual of the population is a complete FCS. The individuals forming the first generation have a random initialization. Finally, the fitness function is defined as follows.

fuzzy inp. 1	fuzzy inp 2.	fuzzy out	rule base
MF1...MF7	MF1...MF7	MF1...MF7	R1,1 R7,7

Fig. 2. Genome of an individual of the population

First we partition the plant state space into smaller regions to make the learning process easier. In the following, by numerical controller we mean the numerical controller restricted to a given region R. Given a starting point p in R, we require the individuals to completely learn the trajectory of the numerical controller starting from p. To this aim, the fitness function is defined to be the distance between the trajectory generated by the individual and the trajectory stored in the numerical controller.

We repeat the GA for a sample set of points $p_1, ..., p_k \in R$, generating k FCSs, $F_1, ..., F_k$, where each F_i is able to drive the system from p_i to the setpoint. Then we analyze the capability of each F_i to perform well on the other points of R, and select the one who is able to drive all points of R to the setpoint in the most efficient way. If no one is able to cope with all points of the region, we can restart the process with other points. After a given number of negative outcomes, we conclude that the region R is too large to be compressed in the space given by the genome. Then, we may either split the region into smaller subregions or augment the number of fuzzy sets and fuzzy rules (so each individual can store more information), and restart the process.

```
void GAFuzzy(char * finput){
    ...
    GAAlleleSet<double> allele_x(inf_x-1, max_x);
    GAAlleleSet<double> allele_y(inf_y-1, max_y);
    GAAlleleSet<double> allele_o(inf_o-1, max_o);
    GAAlleleSet<double> allele_r(inf_r-1, max_r);
    GAAlleleSetArray<double> alleleArray;
    for (i=0; i< 3*n_x; i++) alleleArray.add(allele_x);
    for (i=0; i< 3*n_y; i++) alleleArray.add(allele_y);
    for (i=0; i< 3*n_o; i++) alleleArray.add(allele_o);
    for (i=0; i< n_r; i++) alleleArray.add(allele_r);
    GA1DArrayAlleleGenome<double> genome(alleleArray,objective);
    ...
}
```

Fig. 3. The genome representation in GAlib

To support the development of our GA technique, we used Galib, an efficient multiplatform open source library containing a set of C++ objects that implement several different kinds of genetic algorithms and genetic operations [18].

The GAlib library defines the main components of the genetic algorithm, but allows to freely choose the genome representation that best fits the problem to solve. In our case, the genome should contain the knowledge base of a fuzzy

controller. In particular, we implemented our genome representation using the built-in Galib `GAAlleleSetArray` class. Each element of this array is a `GAAlleleSet` which represents the (range of) possible different values of a specific gene.

As an example, in the car parking fuzzy controller described in Section 6 we used four different kinds of `GAAlleleSet`: three to encode the membership functions (two input variables and one output variable), and one to encode the fuzzy rules. Each kind of `GAAlleleSet` define a specific range of values (see Fig. 3). The complete genome is a `GAAlleleSetArray` containing the genes required to encode the fuzzy sets of each variable ($3n_x + 3n_y + 3n_o$ genes, where n_x, n_y and n_o are the number of fuzzy sets for variables x, y and o, respectively) and the fuzzy rules (n_r genes). Each group of genes is created from the corresponding `GAAlleleSet`.

Finally, GAlib allows to define a fitness function and a terminator function. The latter is used to specify the terminating condition of the GA, which may be, e.g., a particular value of fitness.

6 A Case Study: The Car Parking Problem

To prove the effectiveness of our approach, in this section we show how it can be applied to the well-known car parking problem. After describing the problem setting, we consider the optimal control system generated with the CGMurphi tool, and finally we present experimental results related to the automatic synthesis of a fuzzy controller that compresses the optimal one.

6.1 Problem Definition

In the car parking problem, the goal is backing a car up to a parking place starting from any initial position in the parking lot [10, 7]. As shown in Fig. 4, we describe the car state with three real values:

- the abscissa and the ordinate of the car $x, y \in [0, 12]$, referred to the center of the rear wheels;
- the angle $\varphi \in [-90°, 270°]$ of the longitudinal axis of the car w.r.t. the horizontal axis of the coordinate system.

The objective is to move the car to a final position satisfying $x = 6, \varphi = 90°$. Note that, as in [7], we have no restrictions on the y coordinate, since we assume the initial position to be sufficiently far away from the parking place: if the final x position is reached, to move the car to the parking place it is sufficient to drive back without steering the wheels.

A controller for the car parking problem takes as input the car position and outputs a suitable steering wheels angle $\theta \in [-30°, 30°]$.

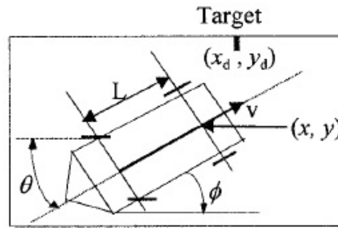

Fig. 4. The simulated car and parking lot

6.2 Experimental Setting and Results

To synthesize the FCS for the car-parking problem, we first partition the state space as follows:

- the abscissa and the ordinate of the car $x, y \in [0, 12]$, are partitioned into 4 regions: $[3i, 3(i + 1)] \times [0, 12]$, $i = 0, \ldots, 3$;
- the angle φ of the longitudinal axis is partitioned into 8 adjacent regions of 45 degrees.

Therefore, the state space is partitioned into 32 regions. In each region R_i, we sample 10 points $p_1, \ldots, p_{10}$ and use the GA to synthesize a FCS for each point, using the following algorithm.

Given a starting point p_i, the task of any individual is to learn the set of control actions, stored in the numerical controller generated by the *CGMurphi* tool, which drives the car from p_i to the parking position. Recall that each individual codes a complete FCS. The fitness of a FCS is obtained by summing up the modula of the distances between the numerical controller actions and the corresponding fuzzy controller actions on the same trajectory. Therefore, the population evolution terminates when the fitness of some individual is zero (i.e., the FCS acts exactly as the numerical controller).

More formally, the evolution process has the following steps:

1. get from the controller table the trajectory from the stating point p_i to the goal; let this trajectory be composed of $k+1$ positions $p_{i1}, \ldots, p_{ik+1}$ and let $r_1, \ldots, r_k$ be the corresponding control actions;
2. WHILE (`fitness` > 0)
   (a) run the genetic algorithm to synthesize the fuzzy controller S;
   (b) determine the action r'_j of S in each position p_{ij};
   (c) calculate the fitness function: `fitness` $= \frac{\sum_{j=1}^{k}(||r_j - r'_j||)}{k}$;

The outcome of the previous process is, for each region R_i, a set of 10 FCS $F_{1,i}, \ldots, F_{10,i}$. Between these, we select the FCS that is able to drive the car to the parking position starting from *any* point in R_i.

To evaluate the performance of the FCS, we have considered all the trajectories starting from each state in the optimal controller table and we have compared the number of steps required to reach the setpoint.

Table 1 shows experimental results of comparison. Note that the FCS is able to cope with a larger number of states, and this proves the improvement in terms of robustness. On the other hand, the FCS performs worse than the optimal one with a degrade of 100%, but however this is an expected result since the CGMurphi-based controller has a tabular representation of *optimal* trajectories.

Moreover, Table 1 shows that with the FCS we obtain a compression of 90% in terms of memory occupation.

Table 1. Experimental comparison between fuzzy and optimal controller performance

Control System	Number of Controlled States	Average Number of Steps to Setpoint	Memory Occupation
FCS	46128	7.568	36608 bytes
CGMurphi	38256	3.614	382560 bytes

7 Conclusions

In this paper we have shown a genetic approach to the automatic generation of fuzzy control systems from preexisting numerical controllers.

Our methodology splits the problem state space in smaller ranges and iteratively uses a GA to synthesize a restricted FCS for each area. The fitness of a FCS is evaluated by comparing its behavior with the one of the numerical controller. The resulting controllers have an average size that is 1/10 of the corresponding numerical controllers, thus achieving a considerable compression ratio. Moreover, the FCS are inherently more robust than the numerical counterparts, so they actually encode a *larger* state space using a *smaller* memory size.

As a natural next step, we are studying how to merge the FCSs generated through our methodology in a single FCS that could be used to drive the system to the setpoint from any position of the state space. We feel that this merge could result in a further compression, since there may be knowledge redundancy between the single FCSs. Moreover, we are interested in finding algorithms for dynamically tuning the GA parameters (e.g., crossover and mutation ratio) in order to speed up the algorithm convergence.

References

[1] Leu, M.C., Kim, T.Q.: Cell mapping based fuzzy control of car parking. In: ICRA, pp. 2494–2499 (1998)

[2] Kreisselmeier, G., Birkholzer, T.: Numerical nonlinear regulator design. IEEE Transactions on Automatic Control 39(1), 33–46 (1994)

[3] Della Penna, G., Intrigila, B., Magazzeni, D., Melatti, I., Tofani, A., Tronci, E.: Automatic generation of optimal controllers through model checking techniques (To be published in Informatics in Control, Automation and Robotics III, Springer-Verlag, Heidelberg) draft available at the `http://www.di.univaq.it/magazzeni/cgmurphi.php`

[4] Della Penna, G., Intrigila, B., Magazzeni, D., Melatti, I., Tofani, A., Tronci, E.: Automatic synthesis of robust numerical controllers (To appear in IEEE Proceedings of the Third International Conference on Autonomic and Autonomous Systems) (ICAS 2007), draft available at the http://www.di.univaq.it/magazzeni/tr.php
[5] Papa, M., Wood, J., Shenoi, S.: Evaluating controller robustness using cell mapping. Fuzzy Sets and Systems 121(1), 3–12 (2001)
[6] Kautz, H., Thomas, W., Vardi, M.Y.: 05241 executive summary – synthesis and planning. In: Kautz, H., Thomas, W., Vardi, M.Y. (eds.) Synthesis and Planning. Number 05241 in Dagstuhl Seminar Proceedings (2006)
[7] Kosko, B.: Neural Networks and Fuzzy Systems. Prentice-Hall, Englewood Cliffs (1992)
[8] Sekine, S., Imasaki, N., Tsunekazu, E.: Application of fuzzy neural network control to automatic train operation and tuning of its control rules. In: Proc. IEEE Int. Conf. on Fuzzy Systems 1993, pp. 1741–1746. IEEE Computer Society Press, Los Alamitos (1995)
[9] Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
[10] Intrigila, B., Magazzeni, D., Melatti, I., Tofani, A., Tronci, E.: A model checking technique for the verification of fuzzy control systems. In: CIMCA05. Proceedings of International Conference on Computational Intelligence for Modelling Control and Automation (2005)
[11] Tan, G.V., Hu, X.: More on design fuzzy controllers using genetic algorithms: Guided constrained optimization. In: Proc. IEEE Int. Conf. on Fuzzy Systems, pp. 497–502. IEEE Computer Society Press, Los Alamitos (1997)
[12] CGMurphi Web Page: http://www.di.univaq.it/magazzeni/cgmurphi.php
[13] Li, H., Gupta, M.: Fuzzy Logic and Intelligent Systems. Kluwer Academic Publishers, Boston, MA (1995)
[14] Jin, J.: Advanced Fuzzy Systems Design and Applications. Physica-Verlag (2003)
[15] Nelles, O., Fischer, M., Muller, B.: Fuzzy rule extraction by a genetic algorithm and constrained nonlinear optimization of membership functions. In: Proceedings of the Fifth IEEE International Conference on Fuzzy Systems, FUZZ–IEEE '96, vol. 1, pp. 213–219. IEEE Computer Society Press, Los Alamitos (1996)
[16] Jian Wang, L.S., Chao, J.F.: An efficient method of fuzzy rules generation. In: Intelligent Processing Systems. ICIPS '97, vol. 1, pp. 295–299 (1997)
[17] Homaifar, A., McCormick, E.: Simultaneous design of membership functions and rule sets for fuzzy controllers using genetic algorithms. In: IEEE Transactions Fuzzy Systems, vol. 3, pp. 129–139. IEEE Computer Society Press, Los Alamitos (1995)
[18] GAlib Web Page: http://lancet.mit.edu/ga