DEE: a Tool for Genetic Tuning of Software Components on a Distributed Network of Workstations

Ernesto Damiani¹, Gianni Degli Antoni¹
and Andrea G. B. Tettamanzi¹

¹Università degli Studi di Milano,
Polo Didattico e di Ricerca di Crema
Via Bramante 65, 26013 Crema (CR), Italy
E-mail: edamiani@crema.unimi.it,
{gda, tettaman}@dsi.unimi.it

Abstract: This paper presents DEE, the Distributed Evolutionary Engine, a complete framework for the off-line tuning of fuzzy-logic based software components using parallel adaptation algorithms. The system was implemented on a high-speed network of workstations by means of a general-purpose task distribution tool. After the description of DEE’s architecture, the tuning of fuzzy software components is discussed as an alternative to maintenance, and some encouraging experimental results are described.

1. Introduction

The idea of using evolutionary algorithms to tune parameters of fuzzy software components is relatively recent. The first attempts in this direction were aimed to the synthesis and optimization of fuzzy controllers (Karr 1991, Thrift 1991).

Besides control, another area of research is data mining, where evolutionary algorithms are used to optimize queries. This optimization task becomes particularly interesting when queries are vague, database indexing is fuzzy and the data themselves are uncertain (Sanchez and Pierre 1994).

Further applications have been envisaged: for example using evolutionary techniques to tune a fuzzy image compression algorithm (Beretta and Tettamanzi 1996).

A broader notion of “fuzzy software components” is currently being adopted by the software engineering community, which includes not only program
modules based on fuzzy “IF-THEN” rules, but also non-rule-based algorithms
and systems that can make use of the fuzzy concepts of linguistic variables and
values, membership functions, fuzzy logical operators and fuzzy numbers and
arithmetics. This comprises a wide class of multipurpose software components
in several application domains.

The main objective of the paper is to describe DEE, a Distributed Evolu-
tionary Engine providing an experimental setting for the tuning of these
components. Our investigation addressed three main aspects:

- inexpensive, large scale synthesis and/or optimization of components;
- reduction of software maintenance efforts;
- reusability, obtained through component tuning rather than classical
  adaptation techniques.

In order to deal with these problems by means of adaptation algorithms,
an intensive use of computational resources is required. Concurrent comput-
ing on networks of heterogeneous machines has gained tremendous attention
and popularity in recent years, after several pioneering works showed its feas-
bility and cost-effectiveness (Sunderam 1992). Moreover, it is well known
that evolutionary algorithms scale well and do not impose high demands on
the communication network. For this reason, they provide an ideal way of
exploiting scattered and loosely interconnected computing resources.

Nowadays even ordinary personal computers, widely available both in busi-
ness and in academic environments, ensure a high level of performance; yet,
these workstations spend most of their time sitting idle. Therefore, we set
out to exploit this unused computational power by designing evolutionary al-
gorithms, which are notably computationally intensive, so as to take advantage
of such cheap resources.

Thus, DEE was designed to run on standard PC networks during idle time;
in particular, a cluster of one hundred Pentium-based computers connected
through an high-speed local ATM network was used for implementing the
present version of the system.

We expect the approach described in the paper to lead to the definition of a
general parallel framework supporting adaptation as a substitute for software
maintenance and an advantage for reuse of general-purpose software compon-
ents.

The paper is organized as follows: in Section 2 we recall the main results
about evolutionary synthesis of software components, with specific reference to
the field of automatic control systems. Then, in Section 3, following Chen and
Rine (1997), we make the case for evolutionary techniques as powerful tools
for software reuse and maintenance reduction, while in Section 4 we briefly
present Active Tools’ Cluster, the task distribution facility we used to implement our system. Section 5 contains a discussion on parallel and distributed evolutionary algorithms and makes the case for a distributed implementation on a network of workstations. The conceptual architecture of DEE is described in Section 6 together with our current experimental setting. Finally, Section 7 presents some experimental results.

2. Evolutionary Synthesis of Software Components

Investigation on using evolutionary algorithms to synthesize software components, let alone fuzzy software components, is relatively recent.

The first significant step in this direction was made by John Koza, of Stanford University, at the beginning of the Nineties with the introduction of his Genetic Programming (Koza 1993). There, for the first time, the idea was suggested that evolutionary techniques could be used to evolve simple pieces of software according to some desired functional specifications.

In parallel with Koza’s work, the first attempts aimed to the synthesis and optimization of fuzzy software components were applied to control, perhaps primed by the good results obtained in the preceding decade by fuzzy controllers hand-crafted by experts.

The task of using a genetic algorithm or, broadly speaking, an evolutionary algorithm for the design of fuzzy controllers was first undertaken at the beginning of this decade by C. L. Karr (Karr 1991) and P. Thrift (Thrift 1991). Similar work has been conducted by Michael Lee and Hideyuki Takagi (Lee and Takagi 1993a, 1993b) at University of California, Berkeley and by Cezary Janikow (Janikow 1994) at University of Missouri, St. Louis.

A common trait among these approaches is that they use very simple shapes for the membership functions (i.e. triangular or trapezoidal), so that these can be compactly encoded by two or three parameters, and they assume that the number of fuzzy domains for each variable is given and fixed. Furthermore, all possible rules, obtained as a Cartesian product of the input variables, are taken into account and their inclusion in the rule set of a controller is usually marked by a bit in the genetic encoding.

These simplifications greatly reduce the degrees of freedom in the search for an optimal controller while guaranteeing a sufficient amount of generality. The design process, however, is reduced to a parameter optimization problem in the framework of a rigid structure. In addition, the structure and implementation of the controllers is tailored for the convenience of the genetic algorithm.

Although the results obtained thus far are promising and this approach might prove to be entirely satisfactory from the practical point of view, this approach is not applicable to the case in which a language for the design of
fuzzy controllers or a hardware architecture for fuzzy inference are given and a fuzzy controller has to be designed upon them.

For this reason other approaches have been tried, which evolve non-parametric fuzzy systems, discovering rules and membership function shapes from scratch (Tettamanzi 1995). These approaches could be described as bottom-up, in the sense that they start from an existing fuzzy implementation from which they tailor a design technique rather than defining a suitable implementation given a design technique.

Besides control, another area of active research is so-called data mining, where evolutionary algorithms can be used to optimize database queries against completeness and relevance of the results they are able to retrieve. This synthesis and optimization task becomes particularly interesting when queries are vague (and therefore expressed through fuzzy propositions), database indexing is fuzzy and the data themselves are uncertain or granulated (Sanchez and Pierre 1994).

Further applications have been envisaged: for example Beretta and Tettamanzi (1996) propose using evolutionary techniques to tune a fuzzy image compression algorithm (not based on rules) by adjusting membership functions.

A representative overview of current state of the art in the field of evolutionary tuning of fuzzy systems is given in Sanchez, Shibata and Zadeh (1996).

3. Maintenance Reduction through Evolutionary Algorithms

Software component reuse is generally considered as an effective approach for decreasing the time and cost of application development (Krueger 1992). However, the effort required by manual modification of components in order to adapt them to new applications has been shown to be a major cause of the failure of software reuse projects (Chen and Rine 1997). Hence, several methods have been proposed to modify components automatically instead of manually, possibly combined with fuzzy classification and retrieval techniques like those presented in Damiani and Fugini (1995).

In the case of fuzzy-logic based components, like the ones described in the previous Section, the reduction of maintenance efforts can be achieved through a tuning process.

In our environment, following the approach presented in Chen and Rine (1997), components training consists of two phases: testing and adapting.

- Component testing consists of generating fault scenarios and using them to locate erroneous fuzzy elements within the component.

- Component adapting means exploiting adaptation algorithms to modify parameters determining the component’s behaviour (e.g. membership
functions and fuzzy rules) in order to improve its response to the environment.

Fault scenarios are sets of states, coded as vectors of environment and input values, that are used to uncover possible faulty operation of the components under processing. In the case of the automotive control components used for our experiments (Chen and Rine 1997), each state is a vector of environmental and vehicular attributes.

As we will see in the following, our distributed experimental setting allows for an extensive adaptation phase, thus avoiding the necessity of a full-fledged scenario generation and test mechanism as the one described in Chen and Rine (1997).

There are two perspectives from which the effectiveness of the combinations under study can be assessed. According to one point of view, effectiveness is determined by the number of erroneous fuzzy elements that are uncovered under new environments. This is, for instance, the perspective of a programmer debugging a software module: the debugging session will be more effective the higher the number of bugs discovered. Therefore, those inputs to the software components (or scenarios) are to be sought for, that make the most bugs in it manifest themselves.

According to another point of view, effectiveness is determined by the reliability of software components found by the search process, understood as the probability that a component behaves correctly in a random scenario, where scenarios might have a given known (a priori) probability distribution. This is the attitude, for instance, of a user choosing among several available software modules to perform the same task: he or she will select the one that, by experience, will have worked best, i.e. correctly most of the times. By the way, this is the essence of evolutionary optimization of software components too.

The former perspective thus leads to the uncovering of faults through the use of a (possibly restricted) set of ad-hoc scenarios, generated so as to maximize their ability to uncover faults when software components are applied to them, while the latter perspective leads to the discovery of the least faulty software components, as in our case, simply by utilizing a much wider statistical sample of all possible scenarios during adaptation. This approach, which is much more computationally expensive, is made viable by the use of parallel hardware. The reader should notice that what is lost on the side of computing performance is gained on the side of generality and reliability of the solutions found. Indeed, there is a risk of overfitting the software components if they are adapted to a small set of ad hoc fault scenarios, in that eventually they might correct their faulty behavior in those cases at the expense of their general performance.
Effectiveness of the adapting phase can later be determined in terms of the well-defined notions of flexibility, adaptability and stability of software components (Chen and Rine 1997). We will not deal with these aspects in this paper.

4. Exploiting a Task Distribution Facility

Networked computers have become common in most working environments. While nowadays even quite ordinary personal computers yield performance that just a few years ago would have been offered only by very expensive machines, it is also true that these workstations spend most of their time either sitting idle, waiting for user input, or powered off during the night. Distributed computing resources are in other words widely available and extremely inexpensive; therefore, designing evolutionary algorithms, which are notably computation intensive, so as to take advantage of such cheap resources, is not only worthwhile, but also compelling.

In particular, the University of Milan’s Soft Computing Lab located in Crema can currently rely on a high-speed ATM network of around 100 Pentium and Pentium Pro workstations running the Microsoft Windows NT operating system. These workstations are normally used by students for their classes. The availability of such a wealth of “undeveloped” computing power prompted us to study ways of exploiting it for our purposes. It is worthwhile remarking that the same situation often exists in industry, especially when many large scale development environments are involved. Many networked systems for concurrent computing on networks of heterogeneous machines have been since long operational and have been used for an impressive range of applications (Sunderam 1992).

A. The Cluster Task Distribution Facility

ActiveTools’ Cluster is an example of a commercial program for managing large computational tasks that executes computationally intensive activity by distributing jobs over a network of computers.

It was designed after the experience of the Nimrod project (Nimrod 1997, Abramson et al. 1997), conceived in 1993 by the Environmental Protection Agency in Melbourne to perform air pollution studies of major Australian cities.

The current version of DEE was realized using the first beta release of Cluster for Windows NT, available on Internet since early 1997, and was instrumental in the evolution of the distribution facility itself.

To assist with a computational task, Cluster generates input parameters, distributes the jobs and collects the results. It provides the following features:
Figure 1: Active Tool’s Clustor’s architecture.

- **Graphical Interface** The graphical interface allows visual generation of input parameters and management of output results.

- **Transparent Task Distribution** Clustor distributes jobs over a network of computers. The distribution of the work and the collection of the results are done transparently to the user.

- **Customization** Custom task interfaces can be prepared through a simple, intuitive graphical interface.

The operation of Clustor can be briefly outlined as follows: the facility executes as a master Root module which is used by the user to initiate and control the computational task, and several slave Clustor Nodes which, in turn, execute the computations on remote computers.

Three programs on the Root are available to the user: the Preparator, the Generator, and the Dispatcher.

The Preparator is used to specify the types of input parameters, file handling and programs to execute on remote computers (in our case, as we shall see in the following, these will be DEE’s modules). The specification consists of a description of task parameters and a description of the task to be executed. The output of the Preparator is a plan file, that can be edited and modified using a text editor.

The Generator is used to specify parameter values for a particular run and the Dispatcher is used to execute the run. The Dispatcher controls the execution of jobs. It starts remote tasks and communicates with them during
job execution. Required files and jobs are distributed to remote computers and results are copied back to the Dispatcher. It is worthwhile repeating that the Preparator allows for a task on the Root machine to be specified for post-processing of results; as we will see in the next Sections, in our case, this will be DEE’s central module.

The Dispatcher provides a graphical interface for remotely controlling the computations on remote machines. The interface offers monitoring commands to obtain statistics on remote jobs.

5. Distributed Genetic Algorithms

Evolutionary Algorithms (EAs) have enjoyed an increasing popularity as reliable stochastic optimization, search and rule-discovering methods in the last few years. The original formulation of this class of algorithms by Holland and others in the Seventies was a sequential one (Goldberg 1989). That approach made it easier to reason about mathematical properties of the algorithms and was justified at the time by the lack of adequate software and hardware. However, it is clear that EAs offer many natural opportunities for parallel or distributed implementation (Mühlenbein 1989).

There are several possible parallel EA models, the most popular being the fine-grained or grid models (Manderick and Spiessens 1989), and the coarse-grained or island models.

In the grid models, large populations of individuals are imagined to be spatially distributed on a low-dimensional grid and individuals interact locally within a small neighborhood. Massively parallel SIMD machines are especially suited for the grid model.

In the island model the individuals are divided into smaller populations which evolve independently of one another and concurrently according to a standard sequential evolutionary algorithm. Periodic migrations of some selected individuals between islands allow to inject new diversity into converging populations. Microprocessor-based multicomputers and workstation clusters are well suited for the implementation of this kind of distributed EA. Being coarse-grained, the island model is less demanding in terms of communication speed and bandwidth, which makes it a good candidate for a cluster implementation, for instance employing an SPMD (Single Program Multiple Data) model in which different populations exchange groups of individual in a loosely synchronous manner. Distributed EAs of this kind have been proposed for example in Cohoon, Hegde, Martin and Richards (1987), Tanese (1989), Starkweather, Whitley and Mathias (1991).

Many authors have observed that parallel and distributed evolutionary algorithms show a superlinear speed-up. An explanation for this is that separation of individuals in space helps to preserve population diversity and vari-

6. DEE Architecture

The software architecture of DEE is outlined in Fig 2. Each one of the distributed slave modules is a copy of the component adaptation part of the tuning environment by Chen and Rine described in (Chen and Rine 1997). The master module, which in charge of executing the pre- and post-processing part of the algorithm, is a simple C++ program called the Shuffler.

At each iteration of DEE’s basic loop, the Shuffler chooses a strategy to evolve a new population from the populations retuned by the slave nodes; once the new population is obtained, the Shuffler sends it back to the slaves for the next iteration. Of course, there are many possible strategies, some of which will be described below. In the current version of the environment the shuffling technique can be chosen among a pre-set list of possibilities, like the simple round-robin technique described in the sequel.

A. Algorithm Overview

When a sequential evolutionary algorithm and 100 workstation are available, the right thing to do would be to rewrite the algorithm, making it distributed. Unfortunately, this takes time and a little debugging. A more straightforward solution consists in taking a cluster management tool off the shelf and use it to run the sequential evolutionary algorithm on multiple machines, perhaps after slightly modifying it in order to enable it to load a population, run for a given number of steps, save its current population and stop, so that exchange of individuals between two populations becomes possible “off-line”.

Figure 2: DEE’s architecture.
This is, basically, what we did by adopting Active Tools’ Clustor, described in Section A. This tool allows a user to set up a cluster of workstation and run an executable on each machine with different arguments. In addition, pre- and post-processing actions can be defined.

Suppose that \( n \) workstations are in the network and that the population size is set to \( m \) for each sequential evolutionary algorithm execution. The overall schema of the distributed algorithm works as follows (see also Figure 3):

1. \( n \) blocks of \( mk + 1 \) scenarios are generated, where \( k \) is the number of sequential evolutions between two subsequent migrations, and each block is sent to a different workstation;

2. an initial population of \( m \) individuals is generated or loaded from a file;

3. the initial population of \( m \) individuals is broadcast to all the \( n \) workstations in the network;

4. each workstation performs, independently of the others, \( k \) sequential evolutions as follows (see also Figure ref:fig:intcycle):
   
   (a) each workstation executes the sequential evolutionary algorithm \( m \) times, with the same initial population, using the \( k \)th set of \( m \) scenarios in its block; each execution produces a single individual;
   
   (b) the resulting \( m \) individuals are assembled to yield the initial population for the next step;

5. each workstation performs an additional sequential evolution, using the \((mk + 1)\)th scenario in its block and the result of Step 4 as the input population, thus producing a single individual;

6. if \( m > n \), the best \( m \) of the resulting \( n \) individuals are assembled to yield the initial population for the next step, otherwise the needed number of individuals is randomly extracted with replacement among the available ones;

7. if the termination condition is not met, go back to Step 3, otherwise the current population is saved to a file.

The scenarios used by the algorithm are randomly generated according to a given probability distribution, which should mirror the expected distribution of actual operating scenarios for the component in the real world. The generated scenarios are distributed into the \( n \) blocks at random, i.e. they are not grouped according to similarity, to avoid biasing the evolutionary process with respect to environmental conditions.
The termination condition can be either reaching a preset number of iterations or a desired quality level in the component performance or, finally, the improvement between two subsequent iterations falling below a given threshold.

The above algorithm was easy to implement using Cluster, except for the looping part. Active Tools was kind enough to implement a new feature to this aim upon our request.

7. Experimental Results

This section illustrates a sample application of DEE to a control problem. In order to make meaningful comparisons possible, we chose the same problem treated in Chen and Rine (1997). We briefly summarize here both the problem and the component architecture used to solve it for sake of completeness, but we refer the interested reader to the cited article.

A. The Auto Cruise Control Problem

The auto cruise control problem consists in maintaining the speed of a vehicle as close as possible to a desired cruising speed $y_d$, under changing road conditions (mainly slope and friction), by acting on its acceleration through the throttle and the brake.

The road conditions (or environment) are specified by four variables:
Figure 4: An instance of the flowchart of the algorithm’s inner loop for each workstation, with $k = 2$ and $m = 6$. 
• grade of slope \( \nu \);
• desired cruising speed \( y_d \);
• road friction \( \mu \);
• vehicle mass \( m \).

The input to the controller consists in the speed error \( e = y_d - y \) (i.e. the difference between the desired speed \( y_d \) and the actual speed of the vehicle \( y \)) and its variation \( \Delta e \). The output of the controller is the force, \( \Delta u \), that has to be applied to the vehicle through appropriate actions on the throttle and the brake in order to correct its speed.

The environment, defined as above, is used in simulating the system operation in order to evaluate the controller performance.

B. The Component Architecture

The fuzzy software components in our sample application are fuzzy rule-based controllers using two linguistic variables as inputs, namely speed error \( e \) and change of error \( \Delta e \), and one output variable, the control \( \Delta u \), which is supposed to be linearly correlated with acceleration.

A fuzzy controller is made up of two conceptual blocks: the “IF-THEN” rules, of the form

\[
\text{IF } e \text{ is } A \text{ AND } \Delta e \text{ is } B \text{ THEN } \Delta u \text{ is } C,
\]

where \( A, B \) and \( C \) are linguistic values, and membership functions definitions for the linguistic values used in the rules.

C. Experimental setting

We performed several test runs, with the settings reported in Table 1.

<table>
<thead>
<tr>
<th>Population size</th>
<th>( m )</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of populations</td>
<td>( n )</td>
<td>6</td>
</tr>
<tr>
<td>Number of sequential evolutions</td>
<td>( k )</td>
<td>4</td>
</tr>
<tr>
<td>Number of scenarios</td>
<td>( n(\frac{mk}{6}+1) )</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 1: Settings used for the test runs.

Fault scenarios were generated at random according to a probability distribution that was intended to mirror representative critical operating conditions while avoiding overfitting with respect to a single situation. In particular, we
proceeded by perturbing a fault scenario that had previously been selected using the method described in Chen and Rine (1997), i.e. such that it maximized the number of erroneous membership functions in a handcrafted controller used as reference. Perturbation was obtained by adding Gaussian noise to the selected fault scenario to obtain a wide range of “nearly critical” conditions.

The initial population of controllers was generated in a similar way from the reference handcrafted controller.

D. Results

Figure 5 shows a sample fuzzy controller found by our algorithm.

The first thing that can be noticed is that such a controller is dramatically different from the reference. This is no wonder, since our controllers have a long evolutionary history and the evolutionary process works in a way that is fundamentally different from how a human expert would reason. For one thing, evolution, whether it is natural or simulated, tends to take advantage of shortcuts, whenever they are available, even if this leads to less elegant solutions. This might constitute a difficulty with respect to readability, when using evolution jointly with conventional software maintenance.

In particular, evolution does not pay attention to the intended meaning of membership function labels: all it matters is in which rules membership functions are used. Therefore, reversals of their ordering are possible.

Another interesting effect is that some linguistic values can be voided of their meaning (e.g. the associated membership function becomes 1 everywhere or 0 everywhere except for a very small interval), thus leading to a sort of simplification of the rule base.

For instance, Linguistic Value SP of Variable $\epsilon$ in the controller of Figure 5 is practically 1 everywhere in the usual range of operation, causing all rules having the clause “$\epsilon$ is SP” in the antecedent to drop that clause. On the other hand, the membership function of Value SN in the same variable has become a crisp value and therefore all rules having the clause “$\epsilon$ is SN” in their antecedent could be eliminated from the rule base without significant change in the controller behavior. If needed, as a part of the maintenance process, one might restructure the rule base according to these considerations, thereby improving readability. In the above controller, for example, of the total 25 rules present in the reference controller, 13 would be eliminated and further 6 rules would be simplified.

8. Concluding Remarks and Future Work

The paper has outlined our current experimental setting for the use of distributed evolutionary algorithms in software component tuning. Our aim at this
Rules:

<table>
<thead>
<tr>
<th>$\epsilon \backslash \Delta \epsilon$</th>
<th>NE</th>
<th>SN</th>
<th>ZE</th>
<th>SP</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>NE</td>
<td>NE</td>
<td>SN</td>
<td>SN</td>
<td>SN</td>
</tr>
<tr>
<td>SN</td>
<td>NE</td>
<td>NE</td>
<td>SN</td>
<td>SN</td>
<td>SN</td>
</tr>
<tr>
<td>ZE</td>
<td>SN</td>
<td>SN</td>
<td>ZE</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>PO</td>
<td>PO</td>
</tr>
<tr>
<td>PO</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>PO</td>
<td>PO</td>
</tr>
</tbody>
</table>

Variable $\epsilon$:

<table>
<thead>
<tr>
<th></th>
<th>$\mu$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>$-11.737389$</td>
<td>$10.545015$</td>
</tr>
<tr>
<td>SN</td>
<td>$-35.183503$</td>
<td>$0.012387$</td>
</tr>
<tr>
<td>ZE</td>
<td>$30.778809$</td>
<td>$46.770811$</td>
</tr>
<tr>
<td>SP</td>
<td>$17.695736$</td>
<td>$802.688367$</td>
</tr>
<tr>
<td>PO</td>
<td>$56.423743$</td>
<td>$1.485902$</td>
</tr>
</tbody>
</table>

Variable $\Delta \epsilon$:

<table>
<thead>
<tr>
<th></th>
<th>$\mu$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>$-2035.405700$</td>
<td>$2172.803163$</td>
</tr>
<tr>
<td>SN</td>
<td>$-24.154917$</td>
<td>$1.800764$</td>
</tr>
<tr>
<td>ZE</td>
<td>$26.689073$</td>
<td>$5.555413$</td>
</tr>
<tr>
<td>SP</td>
<td>$4.582835$</td>
<td>$15.158660$</td>
</tr>
<tr>
<td>PO</td>
<td>$781.997874$</td>
<td>$262.928505$</td>
</tr>
</tbody>
</table>

Variable $\Delta \mu$:

<table>
<thead>
<tr>
<th></th>
<th>$\mu$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>$-2963.104499$</td>
<td>$1393.629347$</td>
</tr>
<tr>
<td>SN</td>
<td>$-12796.308399$</td>
<td>$2773.169689$</td>
</tr>
<tr>
<td>ZE</td>
<td>$364.366226$</td>
<td>$641.611680$</td>
</tr>
<tr>
<td>SP</td>
<td>$1866.759205$</td>
<td>$744.525114$</td>
</tr>
<tr>
<td>PO</td>
<td>$5616.212453$</td>
<td>$170.433953$</td>
</tr>
</tbody>
</table>

Figure 5: A sample fuzzy controller found by the algorithm: best individual after 8 iterations of the main loop. The names of the linguistic values have the following meaning: NE negative, SN small negative, ZE zero, SP small positive, PO positive. Membership functions are bell-shaped and they are defined in terms of their mean, $\mu$, and standard deviation $\sigma$. 
preliminary stage has been to exploit the computational resources available in modern high-speed networks of workstations in conjunction with evolutionary techniques to attain a greater generality and robustness of software components with respect to variations in operational environment, as well as a high degree of optimization. These results can be obtained at the expense of time required for the tuning process. However, on one side the speed-up ensured by increased parallelism can adequately compensate for the additional overhead. On the other hand, it should be borne in mind that the cost of the computational resources involved is negligible in comparison with the cost of a human maintainer.

Future developments of the work illustrated in this paper include the use of DEE for the tuning of other classes of fuzzy software components besides controllers and the extension of functionalities provided by the distributed environment, such as alternative shuffling strategies and subpopulation topologies.

Acknowledgments

This work was partially supported by M.U.R.S.T. 60% funds.

The authors are indebted to David Rine and Junda Chen for their continuous encouragement and precious suggestions during this work. Rok Sosic and the whole development team at ActiveTools were of great help in solving some problems with Cluster software. Finally, many thanks are due to Andrea Bottari for his effort in installing and debugging the Cluster environment and using it to implement DEE as a part of his Master’s thesis.

References


