Halting Problem

The *Halting Problem* is basically an instance of the Entscheidungsproblem and asks for an algorithm that decides whether another algorithm will terminate at some point in time or runs forever if provided with a certain, finite input. Again, Turing [2065, 2066] proved that a general algorithm solving the Halting Problem cannot exist in general. One possible way to show this is to use a simple counter-example: Assume that a correct algorithm *doesHalt* exists (as presumed in Algorithm 4.2) which takes a program *algo* as input and determines whether it will terminate or not. It is now possible to specify a program *trouble* which, in turn, uses *doesHalt* to determine if it will halt at some point in time. If *doesHalt* returns `true`, *trouble* loops forever. Otherwise it halts immediately. In other words, *doesHalt* cannot return the correct result for *trouble* and hence, cannot be applied universally. Thus, it is not possible to solve the Halting Problem algorithmically for Turing-complete programs in a Turing-complete representation. One consequence of this fact is that there are no means to determine when an evolved program will terminate or whether it will do so at all (if its representation allows infinite execution, that is) [2011, 2254]. Langdon and Poli [1243] have shown that in Turing-complete linear Genetic Programming systems, most synthesized programs loop forever and the fraction of halting programs of size *length* is proportional to $\sqrt{length}$, i. e., small.

Algorithm 4.2: *Halting Problem: reductio ad absurdum*

```
1  begin
2  |  doesHalt(algo) ∈ {true, false}
3  |  begin
4  |  |  ...
5  |  end
6  |  Subalgorithm trouble()
7  |  begin
8  |  |  if doesHalt(trouble) then
9  |  |  |  while true do
10 |  |  |  |  ...
11 |  end
12 end
```

Countermeasures

Against the Entscheidungsproblem

For general, Turing-complete program representations, neither exhaustive testing nor algorithmic detection of correctness is possible.

Model Checking Model checking[76] techniques [413, 1483] have made great advance since the 1980s. According to Clarke and Emerson [412], *"Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model."* The result of the checking process is either a confirmation of the correctness of the checked model, a counterexample in which it fails to obey its specification, or failure, i. e., a situation in which no conclusion could be reached.

Hence, in the context of Genetic Programming, a model checker can be utilized as a Boolean function $\varphi : \mathbb{X} \mapsto \mathbb{B}$ which maps the evolved programs to *correct* ($\equiv$ `true`) or

[76] http://en.wikipedia.org/wiki/Model_checking [accessed 2008-10-02]

incorrect (≡ `false`). As objective function, φ therefore is rather infeasible, since it would lead directly to the all-or-nothing problem discussed in Section 4.10.2.[77]

Still, model checkers can be an interesting way to define termination criteria for the evolution or to verify its results. This may require a reduction of the expressiveness of the GP approaches utilized in order to make them compliant with the input languages of the model checkers. Then again, there are very powerful model checkers such as SPIN[78] [955, 176, 256], which processes systems written in the Promela[79] (the Process Meta Language) with which asynchronous distributed algorithms can be specified [175]. If such a system was used, no reduction of the expressiveness of the program representation would be needed at all. Nevertheless, a formal transformation of the GP representation to these languages must be provided in any circumstance. Creating such a transformation is complicated and requires a formal proof of correctness – checking a model without having shown the correctness of the model representation first is, basically, nonsense.[80]

The idea of using model checkers like SPIN is very tempting. One important drawback of this method is the unforeseeable runtime of the checking process which spans from almost instantaneous return up to almost half an hour [2031]. In the same series of experiments ([2031]), the checking process also failed in a fraction of cases ($\approx 18\%$) depending on the problem to be verified. Especially the unpredictable runtime for general problems led us to the decision to not use SPIN in our own works yet, since in the worst case, a few thousand program verifications could be required per generation in the GP system. Still, it is an interesting idea to evolve programs in Promela language and we will reconsider it in our future work and evaluate the utility and applicability of SPIN for the said purposes in detail.

Functional Adequacy In the face of this situation where we cannot automatically determine whether an evolved algorithm is correct, overfitted, or oversimplified, a notation for which solutions are acceptable and which are not is required. One definition which fits perfectly in this context is the idea of *functional adequacy* provided by Camps et al. [327], Gleizes et al. [809]:

Definition 4.7 (Functional Adequacy). When a system has the "right" behavior – judged by an external observer knowing the environment – we say that it is functionally adequate [809].

In the context of Genetic Programming, the *external observer* is represented by the objective functions which evaluate the *behavior* of the programs in the simulation *environments*. According to Gleizes et al. [809], functional adequacy also subsumes non-functional criteria such as memory consumption or response time if they become crucial in a certain environment, i. e., influence the functionality. For optimizing such criteria, different additional approaches are provided in Section 4.10.3.

Against The Halting Problem

In order to circumvent the Halting Problem, the evolved programs can be executed in simulations which allow limiting their runtime [2254, 1027]. Programs which have not finished until the time limit has elapsed are terminated automatically. Especially in linear Genetic Programming approaches, it is easy to do so by simply defining an upper bound for the number of instructions being executed. For tree-based representations, this is slightly more complicated.

Teller [2011] suggests to apply time-limiting approaches too, but also the use of so-called *anytime algorithms*, i. e., algorithms that store their best guess of the result in a certain

[77] One approach to circumvent this problem would be to check for several properties separately.
[78] `http://en.wikipedia.org/wiki/SPIN_model_checker` [accessed 2008-10-02]
[79] `http://en.wikipedia.org/wiki/Promela` [accessed 2008-10-02]
[80] Thanks to Hendrik Skubch for discussing this issue with me.

memory cell and update it during their run. Anytime algorithms can be stopped at any time, since the result is always there, although it would have been refined in the future course of the algorithm.

Another way to deal with this problem is to prohibit the evolution of infinite loops or recursions from the start by restricting the structural elements in the programming language. If there are no loops, there surely cannot be infinite ones either. Imposing such limitations, however, also restricts the programs that can evolve: A representation which does not allow infinite loops cannot be Turing-complete either.

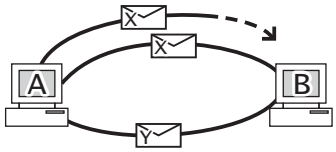

Figure 4.34: A sketch of an infinite message loop.

Often it is not sufficient to restrict just the programming language. An interesting example for this issue is the evolution of *distributed* algorithms. Here, the possible network situations and the reactions to them would also need to be limited. One would need to exclude situations like the one illustrated in Figure 4.34 where

1. node `A` sends message `X` to node `B` which
2. triggers an action there, leading to a response message `Y` from `B` back to node `A` which, in turn,
3. causes an action on `A` that includes sending `X` to `B` again
4. and so on...

Preventing such a situation is even more complicated and will, most likely, also prevent the evolution of useful solutions.

4.10.2 All-Or-Nothing?

The evolution of algorithms often proves as a special instance of the needle-in-a-haystack problem. From a naïve and, at the same time, mathematically precise point of view, an algorithm computing the greatest common divisor of two numbers, for instance, is either correct or wrong. Approaching this problem straightforwardly leads to the application of a single objective function which can take on only two values, provoking the *all-or-nothing* problem in Genetic Programming. In such a fitness landscape, a few steep spikes of equal height represent the correct algorithms and are distributed over a large plane of infeasible solution candidates with equally bad fitness.

The negative influence of all-or-nothing problems have been reported from many areas of Genetic Programming, such as the evolution of distributed protocols [2058] (see Section 23.2.2), quantum algorithms [1932], expression parsers [1027], and mathematical algorithms (such as the GCD).

In Section 21.3.2, we show how to some means to mitigate this problem for the GCD evolution. However, like those mentioned in some of the previously cited works, such methods are normally application dependent and often cannot be transferred to other problems in a simple manner.

Countermeasures

There are two direct countermeasures against the all-or-nothing problem in GP. The first one is to devise objective functions which can take on as many values as possible, i. e., which also reward partial solutions.

The second countermeasure is using as many test cases as possible and applying the objective functions to all of them, setting the final objective values to be the average of the results. Testing with ten training cases will transform a binary objective function to one which (theoretically) can take on eleven values, for instance: 1.0 if all training cases were processed correctly, 0.9 if one training case failed while nine worked out properly, ..., and 0.0 if the evolved algorithm was unable to behave adequately in any of the training cases. Using multiple training cases has, of course, the drawback that the time needed for the objective function evaluation will increase (linearly).

Vaguely related to these two measures is another approach, the utilization of Lamarckian evolution [522, 2215] or the Baldwin effect [123, 929, 930, 2215] (see Section 15.2 and Section 15.3, respectively). As already pointed out in Section 1.4.3, they incorporate a local search into the optimization process which may further help to smoothen out the fitness landscape [864].

In our experiments reported in [2177], an approach similar to Lamarckian evolution was incorporated. Although providing good results, the runtime of the approaches increased to a degree rendering it unfeasible for large-scale.[81]

4.10.3 Non-Functional Features of Algorithms

Besides evaluating an algorithm in terms of its functionality, there always exists a set of non-functional features that should be regarded too. For most non-functional aspects (such as code size, runtime requirements, and memory consumption) and the *parsimony*[82] principle holds: *less is better*. In this section, we will discuss various reasons for applying parsimony pressure in Genetic Programming.

Code Size

In Section 30.1.1 on page 547, we define what algorithms are: compositions of atomic instructions that, if executed, solve some kind of problem or a class of problems. Without specifying any closer what *atomic instructions* are, we can define the following:

Definition 4.8 (Code Size). The code size of an algorithm or program is the number of atomic instructions it is composed of.

The atomic instructions cannot be broken down into smaller pieces. Therefore, the code size is a positive integer number in $\mathbb{N}_0$. Since algorithms are statically finite per definition (see Definition 30.9 on page 550), the code size is always finite.

Code Bloat

Definition 4.9 (Bloat). In Genetic Programming, *bloat* is the uncontrolled growth in size of the individuals during the course of the evolution [1318, 229, 140, 1196, 1241].

The term code bloat is often used in conjunction with code *introns*, which are regions inside programs that do not contribute to the functional objective values (because they can never be reached, for instance; see Definition 3.2 on page 146). Limiting the code size and increasing the code efficiency by reducing the number of introns is an important task in Genetic Programming since disproportionate program growth has many bad side effects like:

1. The evolving programs become unnecessarily big while elegant solutions should always be as small and simple as possible.

[81] These issues were not the subject of the paper and thus, not discussed there.
[82] http://en.wikipedia.org/wiki/Parsimony [accessed 2008-10-14]

2. Mutation and recombination operators always have to select the point in an individual where they will apply their changes. If there are many points that do not contribute to functionality, the probability of selecting such a point for modification is high. The generated offspring will then have exactly the same functionality as its parents and the genetic operation performed was literarily useless.
3. Bloat slows down both, the evaluation [872] and the breeding process of new solution candidates.
4. Furthermore, it leads to increased memory consumption of the Genetic Programming system.

There are many theories about how code bloat emerges [1318], some of them are:

1. Unnecessary code hitchhikes with good individuals. If it is part of a fit solution candidate that creates many offspring, it is likely to be part of many new individuals. According to Tackett [1994], high selection pressure is thus likely to cause code growth. This idea is supported by the research of Langdon and Poli [1241], Smith and Harries [1906], and Gustafson et al. [872].
2. As already stated, unnecessary code makes it harder for genetic operations to alter the functionality of an individual. In most cases, genetic operators yield offspring with worse fitness than its parents. If a solution candidate has good objective values, unnecessary code can be one defense method against recombination and mutation. If the genetic operators are neutralized, the offspring will have the same fitness as its parent. This idea has been suggested in many sources, such as [229, 228, 1544, 1384, 1756, 140, 1244, 1906]. From this point of view, introns are a "bad" form of neutrality[83]. By the way, the reduction of the destructive effect of recombination on the fitness may also have positive effects, as pointed out by Nordin et al. [1546, 1547], since it may lead to a more durable evolution.
3. Luke [1318] defines a theory for tree growth based on the fact that recombination is likely to destroy the functionality of an individual. However, the deeper the crossover point is located in the tree, the smaller is its influence because fewer instructions are removed. If only a few instructions are replaced from a functionally adequate program, they are likely to be exchanged by a larger sub-tree. A new offspring that retains the functionality of its parents therefore tends to be larger.
4. Similar to the last two theories, the idea of removal bias by Soule and Foster [1922] states that removing code from an individual will preserve the individual's functionality if the code removed is non-functional. Since the portion of useless code inside a program is finite, there also exists an upper limit of the amount of code that can be removed without altering the functionality of the program. For the size of new sub-trees that could be inserted instead (due to mutation or crossover), no such limit exists. Therefore, programs tend to grow [1922, 1244].
5. According to the diffusion theory of Langdon et al. [1244], the number of large programs in the problem space that are functionally adequate is higher than the number of small adequate programs. Thus, code bloat could correspond to the movement of the population into the direction of equilibrium [1318].
6. Another theory considers the invalidators that make code unreachable or dysfunctional. In the formula $4+0*(4-x)$ for example, the multiplication with 0 makes the whole part $(4-x)$ *inviable*. Luke [1318] argues that the influence of invalidators would be higher in large trees than in small trees. If programs grow while the fraction of invalidators remains constant and those inherited from the parents stay in place, their chance to occur proportionally closer to the root increases. Then, the amount of unnecessary instructions would increase too and naturally approach 100%.
7. Instead of being real solutions, programs that grow uncontrolled also tend to be some sort of decision tables. This phenomenon is called *overfitting* and has already discussed

[83] You can find the topic of neutrality discussed in Section 1.4.5 on page 64.

in Section 1.4.8 on page 72 and Section 23.1.3 on page 399 in detail. The problem is that overfitted programs tend to have marvelous good fitness for the training cases/sample data, but are normally useless for any other input.

8. Like Tackett [1994], Gustafson et al. [872] link code growth to high selection pressure but also to loss of diversity in general. In populations with less diversity, recombination will frequently be applied to very similar individuals, which often yields slightly larger offspring.

Some approaches for fighting bloat are discussed in Section 4.10.3.

Runtime and Memory Consumption

Another aspect subject to minimization is generally the runtime of the algorithms grown. The amount of steps needed to solve a given task, i. e., the time complexity, is only loosely related to the code size. Although large programs with many instructions tend to run longer than small programs with few instructions, the existence of loops and recursion invalidates a direct relation.

Like the complexity in time, the complexity in memory space of the evolved solutions often is minimized, too. The number of variables and memory cells needed by program in order to perform its work should be as small as possible. Section 30.1.3 on page 550 provides some additional definitions and discussion about the complexity of algorithms.

Errors

An example for an application where the non-functional errors that can occur should be minimized is symbolic regression. Therefore, the property of *closure* specified in Definition 4.1 on page 178 is usually ensured. Then, the division operator `div` is re-defined in order to prevent division-by-zero errors. Therefore, such a division could either be rendered to a `nop` (i. e., does nothing) or yields 1 or the dividend as result. However, the number of such arithmetical errors could also be counted and made the subject to minimization too.

Transmission Count

If evolving distributed algorithms, the number of messages required to solve a problem should be as low as possible since transmissions are especially costly and time-consuming operations.

Optimizing Non-Functional Aspects

Optimizing the non-functional aspects of the individuals evolved is a topic of scientific interest.

1. One of the simplest means of doing so is to define additional objective functions which minimize the program size and to perform a multi-objective optimization. Successful and promising experiments by Bleuler et al. [227], de Jong et al. [510], and Ekárt and Németh [626] showed that this is a viable countermeasure for code bloat, for instance.
2. Another method is limiting the aspect of choice. A very simple measure to limit code bloat, for example, is to prohibit the evolution of trees with a depth surpassing a certain limit [1320].
3. Poli [1660] furthermore suggests that the fitness of a certain portion of the population with above-average code size should simply be set to the worst possible value. These artificial *fitness holes* will repel the individuals from becoming too large and hence, reduce the code bloat.

5

Evolution Strategy

5.1 Introduction

Evolution Strategies[1] (ES) introduced by Rechenberg [1712, 1713, 1714] are a heuristic optimization technique based in the ideas of adaptation and evolution, a special form of evolutionary algorithms [1712, 1713, 1714, 103, 200, 1841, 198, 916]. Evolution Strategies have the following features:

1. They usually use vectors of real numbers as solution candidates, i. e., $\mathbb{G} = \mathbb{X} = \mathbb{R}^n$. In other words, both the search and the problem space are fixed-length strings of floating point numbers, similar to the real-encoded genetic algorithms mentioned in Section 3.3 on page 145.
2. Mutation and selection are the primary operators and recombination is less common.
3. Mutation most often changes the elements $\mathbf{x}_{[i]}$ of the solution candidate vector $\mathbf{x}$ to a number drawn from a normal distribution $N\left(\mathbf{x}_{[i]}, \sigma_i^2\right)$. For reference, you can check Equation 11.1 on page 259 in the text about Random Optimization.
4. Then, the values σ_i are governed by self-adaptation [891, 1400, 1214] such as covariance matrix adaptation [888, 889, 890, 1041].
5. In all other aspects, they perform exactly like basic evolutionary algorithms as defined in Algorithm 2.1 on page 99.

5.2 General Information

5.2.1 Areas Of Application

Some example areas of application of Evolution Strategy are:

Application	References
Data Mining and Data Analysisanalysis	[445]
Scheduling	[971]
Chemistry, Chemical Engineering	[1755, 470, 632]
Ressource Minimization, Environment Surveillance/Protection	[1556]
Combinatorial Optimization	[1536, 193, 197]
Geometry and Physics	[1122, 2173]
Optics and Image Processing	[859, 860, 101, 2218, 2217, 1279]

[1] http://en.wikipedia.org/wiki/Evolution_strategy [accessed 2007-07-03], http://www.scholarpedia.org/article/Evolution_Strategies [accessed 2007-07-03]

5.2.2 Conferences, Workshops, etc.

Some conferences, workshops and such and such on Evolution Strategy are:

EUROGEN: Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems see Section 2.2.2 on page 106

5.2.3 Books

Some books about (or including significant information about) Evolution Strategy are:

Schwefel [1841]: *Evolution and Optimum Seeking: The Sixth Generation*
Rechenberg [1713]: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*
Rechenberg [1714]: *Evolutionsstrategie '94*
Beyer [198]: *The theory of evolution strategies*
Schwefel [1840]: *Numerical Optimization of Computer Models*
Schöneburg, Heinzmann, and Feddersen [1831]: *Genetische Algorithmen und Evolutionsstrategien*
Bäck [99]: *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*

5.3 Populations in Evolution Strategy

Evolution Strategies usually combine truncation selection (as introduced in Section 2.4.2 on page 122) with one of the following population strategies. These strategies listed below have partly been borrowed from German Wikipedia [2219] site for Evolution Strategy[2].

5.3.1 $(1 + 1)$-ES

The population only consists of a single individual which is reproduced. From the elder and the offspring, the better individual will survive and form the next population. This scheme is very close to hill climbing which will be introduced in Chapter 10 on page 253.

5.3.2 $(\mu + 1)$-ES

Here, the population contains μ individuals from which one is drawn randomly. This individual is reproduced from the joint set of its offspring and the current population, the least fit individual is removed.

5.3.3 $(\mu + \lambda)$-ES

Using the reproduction operations, from μ parent individuals $\lambda \geq \mu$ offspring are created. From the joint set of offspring and parents, only the μ fittest ones are kept [936].

[2] http://de.wikipedia.org/wiki/Evolutionsstrategie [accessed 2007-07-03]

5.3.4 (μ, λ)-ES

In (μ, λ) Evolution Strategies, introduced by Schwefel [1840], again $\lambda \geq \mu$ children are created from μ parents. The parents are subsequently deleted and from the λ offspring individuals, only the μ fittest are retained [1840, 196].

5.3.5 $(\mu/\rho, \lambda)$-ES

Evolution Strategies named $(\mu/\rho, \lambda)$ are basically (μ, λ) strategies. The additional parameter ρ is added, denoting the number of parent individuals of one offspring. As already said, normally, we only use mutation ($\rho = 1$). If recombination is also used as in other evolutionary algorithms, $\rho = 2$ holds. A special case of $(\mu/\rho, \lambda)$ algorithms is the $(\mu/\mu, \lambda)$ Evolution Strategy [1369].

5.3.6 $(\mu/\rho + \lambda)$-ES

Analogously to $(\mu/\rho, \lambda)$-Evolution Strategies, the $(\mu/\rho + \lambda)$-Evolution Strategies are (μ, λ) approaches where ρ denotes the number of parents of an offspring individual.

5.3.7 $(\mu', \lambda'(\mu, \lambda)^\gamma)$-ES

Geyer et al. [791, 792, 793] have developed nested Evolution Strategies where λ' offspring are created and isolated for γ generations from a population of the size μ'. In each of the γ generations, λ children are created from which the fittest μ are passed on to the next generation. After the γ generations, the best individuals from each of the γ isolated solution candidates propagated back to the top-level population, i. e., selected. Then, the cycle starts again with λ' new child individuals. This nested Evolution Strategy can be more efficient than the other approaches when applied to complex multimodal fitness environments [1714, 793].

5.4 One-Fifth Rule

The $\frac{1}{5}$ success rule defined by Rechenberg [1713] states that the quotient of the number of successful mutations (i. e., those which lead to fitness improvements) to the total number of mutations should be approximately $\frac{1}{5}$. If the quotient is bigger, the σ-values should be increased and with that, the scatter of the mutation. If it is lower, σ should be decreased and thus, the mutations are narrowed down.

5.5 Differential Evolution

5.5.1 Introduction

Differential Evolution[3] (DE, DES) is a method for mathematical optimization of multidimensional functions that belongs to the group of evolution strategies [1676, 653, 1404, 288, 1234, 1391, 189]. Developed by Storn and Price [1974], the DE technique has been invented in order to solve the Chebyshev polynomial fitting problem. It has proven to be a very reliable optimization strategy for many different tasks where parameters that can be encoded in real vectors.

The essential idea behind Differential Evolution is the way the (ternary) recombination operator "deRecombination" is defined for creating new solution candidates. The difference

[3] http://en.wikipedia.org/wiki/Differential_evolution [accessed 2007-07-03], http://www.icsi.berkeley.edu/~storn/code.html [accessed 2007-07-03]

$\mathbf{x}_1 - \mathbf{x}_2$ of two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ in $\mathbb{X}$ is weighted with a weight $w \in \mathbb{R}$ and added to a third vector $\mathbf{x}_3$ in the population.

$$\mathbf{x} = \text{deRecombination}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \Rightarrow \mathbf{x} = \mathbf{x}_3 + w\,(\mathbf{x}_1 - \mathbf{x}_2) \tag{5.1}$$

Except for determining w, no additional probability distribution has to be used and the Differential Evolution scheme is completely self-organizing. This classical reproduction strategy has been complemented with new ideas like triangle mutation and alternations with weighted directed strategies.

Gao and Wang [770] emphasize the close similarities between the reproduction operators of Differential Evolution and the search step of the downhill simplex. Thus, it is only logical to combine or to compare the two methods (see Section 16.4 on page 286). Further improvements to the basic Differential Evolution scheme have been contributed, for instance, by Kaelo and Ali. Their DERL and DELB algorithms outperformed [1078, 1079, 1077] standard DE on the test benchmark from Ali et al. [38].

5.5.2 General Information

Areas Of Application

Some example areas of application of Differential Evolution are:

Application	References
Engineering, Structural Optimization, and Design	[1233, 1506]
Chemistry, Chemical Engineering	[2148, 1846, 2052, 399]
Scheduling	[1289]
Function Optimization	[1972]
Electrical Engineering and Circuit Design	[1971, 1973]

Journals

Some journals that deal (at least partially) with Differential Evolution are:

Journal of Heuristics (see Section 1.6.3 on page 91)

Books

Some books about (or including significant information about) Differential Evolution are:

Price, Storn, and Lampinen [1676]: *Differential Evolution – A Practical Approach to Global Optimization*
Feoktistov [653]: *Differential Evolution – In Search of Solutions*
Corne, Dorigo, Glover, Dasgupta, Moscato, Poli, and Price [448]: *New Ideas in Optimisation*

6

Evolutionary Programming

6.1 Introduction

Different from the other major types of evolutionary algorithms introduced, there exists no clear specification or algorithmic variant for evolutionary programming[1] (EP) to the knowledge of the author. There is though a semantic difference: while single individuals of a species are the biological metaphor for solution candidates in other evolutionary algorithms, in evolutionary programming, a solution candidate is thought of as a species itself.[2] Hence, mutation and selection are the only operators used in EP and recombination is usually not applied. The selection scheme utilized in evolutionary programming is normally quite similar to the $(\mu + \lambda)$ method in Evolution Strategies.

Evolutionary programming was pioneered by Fogel [705] in his PhD thesis back in 1964. Fogel et al. [708] experimented with the evolution of finite state machines as predictors for data streams [623]. Evolutionary programming is also the research area of his son David Fogel [697, 699, 700] with whom he also published joint work [707, 1671].

Generally, it is hard to distinguish evolutionary programming from Genetic Programming, genetic algorithms, and Evolution Strategy. Although there are semantic differences (as already mentioned), the author thinks that the many aspects of the evolutionary programming approach have merged into these other research areas.

6.2 General Information

6.2.1 Areas Of Application

Some example areas of application of evolutionary programming are:

Application	References
Machine Learning	[697]
Cellular Automata and Finite State Machines	[708]
Evolving Behaviors, e.g., for Agents or Game Players	[699, 700]
Machine Learning	[1671]
Chemistry, Chemical Engineering and Biochemistry	[779, 609, 778]
Electrical Engineering and Circuit Design	[1135, 1518]
Data Mining and Data Analysis	[1802]
Robotics	[1136]

[1] http://en.wikipedia.org/wiki/Evolutionary_programming [accessed 2007-07-03]

[2] In this aspect it is very similar to the much newer Extremal Optimization approach which will be discussed in Chapter 13.

6.2.2 Conferences, Workshops, etc.

Some conferences, workshops and such and such on evolutionary programming are:

EP: International Conference on Evolutionary Programming
now part of *CEC*, see Section 2.2.2 on page 105
History: 1998: San Diego, California, USA, see [1670]
1997: Indianapolis, Indiana, USA, see [68]
1996: San Diego, California, USA, see [709]
1995: San Diego, California, USA, see [1380]
1994: see [1849]
1993: see [702]
1992: see [701]

EUROGEN: Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems
see Section 2.2.2 on page 106

6.2.3 Books

Some books about (or including significant information about) evolutionary programming are:

Fogel, Owens, and Walsh [708]: *Artificial Intelligence through Simulated Evolution*

Fogel [706]: *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*

Fogel [697]: *System Identification through Simulated Evolution: A Machine Learning Approach to Modeling*

Fogel [700]: *Blondie24: playing at the edge of AI*

Bäck [99]: *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*

7

Learning Classifier Systems

7.1 Introduction

In the late 1970s, Holland, the father of genetic algorithms, also invented the concept of classifier systems (CS) [948, 941, 946]. These systems are a special case of production systems [497, 498] and consist of four major parts:

1. a set of interacting production rules, called *classifiers*,
2. a performance algorithm which directs the actions of the system in the environment,
3. a learning algorithm which keeps track on the success of each classifier and distributes rewards, and
4. a genetic algorithm which modifies the set of classifiers so that variants of good classifiers persist and new, potentially better ones are created in an efficient manner [947].

By time, classifier systems have undergone some name changes. In 1986, reinforcement learning was added to the approach and the name changed to Learning Classifier Systems[1] (LCS) [916, 1909]. Learning Classifier Systems are sometimes subsumed under a machine learning paradigm called evolutionary reinforcement learning (ERL) [916] or Evolutionary Algorithms for Reinforcement Learning (EARLs) [1460].

7.2 General Information

7.2.1 Areas Of Application

Some example areas of application of Learning Classifier Systems are:

Application	References
Data Mining and Data Analysisg	[768, 92, 479, 444, 2178]
Grammar Induction	[2073, 2074, 472]
Medicine	[951]
Image Processing	[1287, 1376]
Sequence Prediction	[1736]

7.2.2 Conferences, Workshops, etc.

Some conferences, workshops and such and such on Learning Classifier Systems are:

[1] http://en.wikipedia.org/wiki/Learning_classifier_system [accessed 2007-07-03]

IWLCS: International Workshop on Learning Classifier Systems
Nowadays often co-located with GECCO (see Section 2.2.2 on page 107).
History: 2007: London, England, see [1946]
2006: Seattle, WA, USA, see [1847]
2005: Washington DC, USA, see [2157, 1181]
2004: Seattle, Washington, USA, see [1848, 1181]
2003: Chicago, IL, USA, see [2022, 1181]
2002: Granada, Spain, see [1254]
2001: San Francisco, CA, USA, see [1944]
2000: Paris, France, see [1253]
1999: Orlando, Florida, USA, see [1585]
1992: Houston, Texas, USA, see [1501]

7.2.3 Books

Some books about (or including significant information about) Learning Classifier Systems are:

Bull [301]: *Applications Of Learning Classifier Systems*
Bull and Kovacs [303]: *Foundations of Learning Classifier Systems*
Butz [314]: *Anticipatory Learning Classifier Systems*
Butz [315]: *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*
Lanzi, Stolzmann, and Wilson [1252]: *Learning Classifier Systems, From Foundations to Applications*

7.3 The Basic Idea of Learning Classifier Systems

Figure 7.1 illustrates the structure of a Michigan-style Learning Classifier System. A classifier system is connected via detectors *(b)* and effectors *(c)* to its environment *(a)*. The input in the system (coming from the detectors) is encoded in form of binary messages that are written into a message list *(d)*. On this list, simple `if-then` rules *(e)*, the so-called classifiers, are applied. The result of a classification is again encoded as a message and written to the message list. These new messages may now trigger other rules or are signals for the effectors [507]. The payoff of the performed actions is distributed by the credit apportionment system *(f)* to the rules. Additionally, a rule discovery system *(g)* is responsible for finding new rules and adding them to the classifier population [794].

Classifier systems are special instances of production systems, which were shown to be Turing-complete by Post [1672] and Minsky [1427, 1426]. Thus, Learning Classifier Systems are as powerful as any other Turing-equivalent programming language and can be pictured as something like computer programs where the rules play the role of the instructions and the messages are the memory.

7.3.1 A Small Example

In order to describe how rules and messages are structured in a basic classifier systems, we borrow a simple example from Heitkötter and Beasley [916]. We will orient our explanation at the syntax described by Geyer-Schulz [794]. You should, however, be aware that there are many different forms of classifier system and take this as an example for *how it could be done* rather than as *the way it is to be done*.

So let us imagine that we want to find a classifier system that is able to control the behavior of a frog. Our frog likes to eat nutritious flies. Therefore, it can detect small,

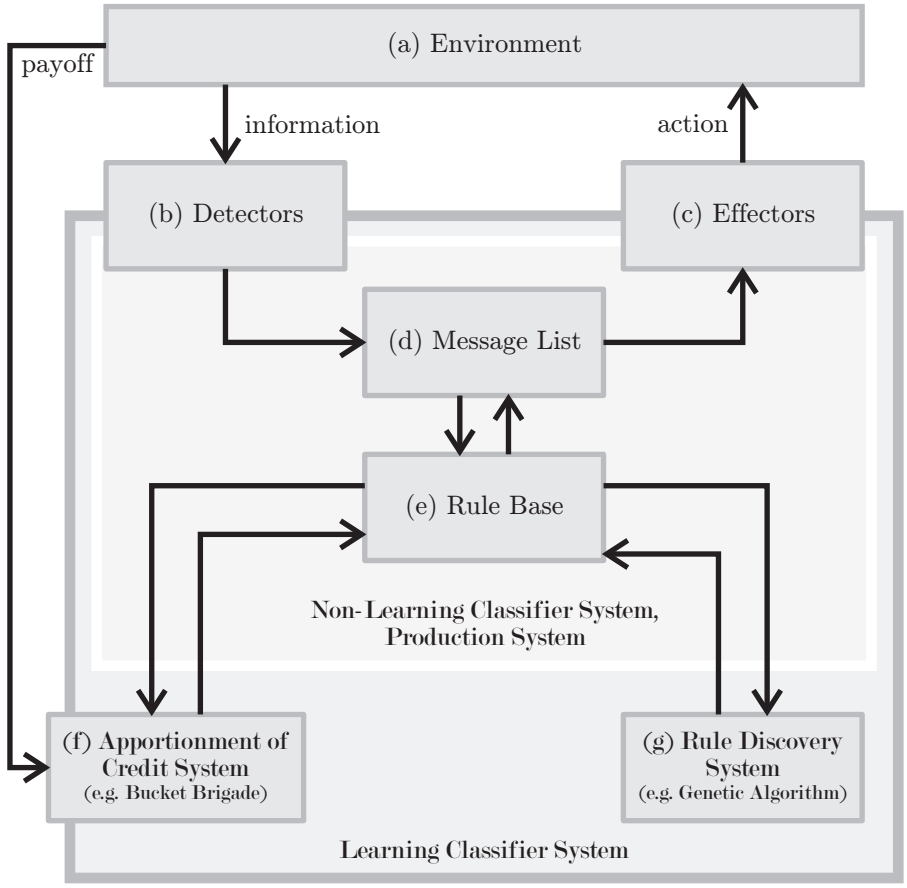

Figure 7.1: The structure of a Michigan style Learning Classifier System according to Geyer-Schulz [794].

flying objects and eat them if they are right in front of it. The frog also has a sense of direction and can distinguish between objects which are in front, to the left, or to the right of it and may also turn into any of these directions. It can furthermore distinguish objects with stripes from those without. Flying objects with stripes are most likely bees or wasps, eating of which would probably result in being stung. The frog can also sense large, looming objects far above: birds, which should be avoided by jumping away quickly. We can compile a corresponding behavior into the form of simple `if-then` rules which are listed in Table 7.1.

No.	premise (`if`-part)	conclusion (`then`-part)
1	small, flying object with no stripes to the left	send a
2	small, flying object with no stripes to the right	send b
3	small, flying object with no stripes to the front	send c
4	large, looming object	send d
5	a and not d	turn left
6	b and not d	turn right
7	c and not d	eat
8	d	move away rapidly

Table 7.1: `if-then` rules for frogs

7.3.2 Messages

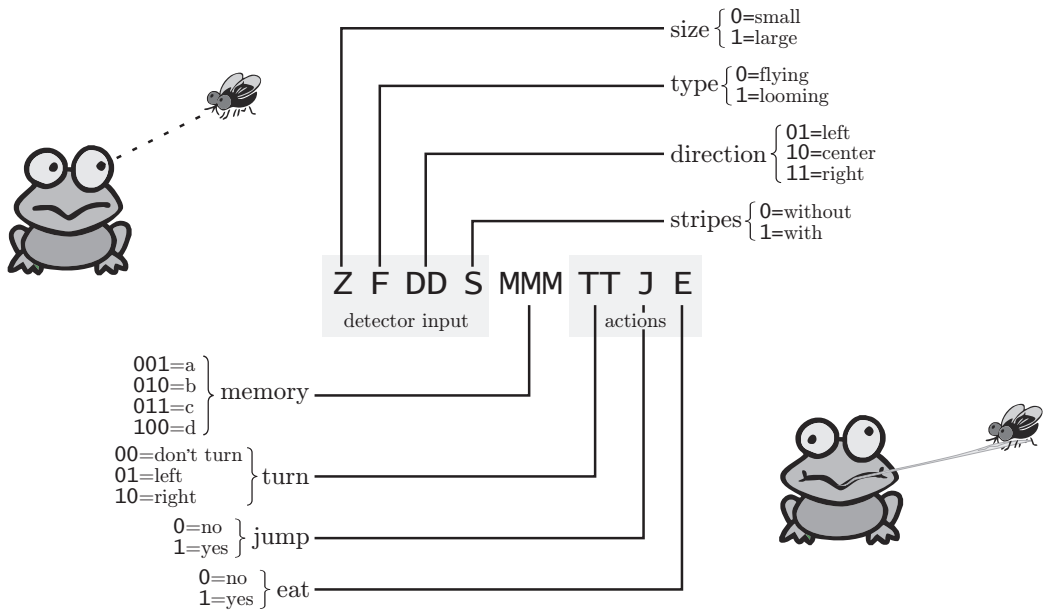

Figure 7.2: One possible encoding of messages for a frog classifier system

In Figure 7.2, we demonstrate how the messages in a classifier system that drives such a frog can be encoded. Here, input information as well as action commands (the conclusions of the rules) are compiled in one message type. Also, three bits are assigned for encoding the internal messages a to d. Two bits would not suffice, since `00` occurs in all "original" input messages. At the beginning of a classification process, the input messages are written to the message list. They contain information only at the positions reserved for detections and have zeros in the bits for memory or actions. The classifiers transform them to internal messages which normally have only the bits marked as "memory" set. These messages are finally transformed to output messages by setting some action bits. In our frog system, a message is in total $k = 12$ bits long, i. e., $\text{len}(m) = 12\ \forall \textit{message}\ m$.

7.3.3 Conditions

Rules in classifier systems consist of a condition part and an action part. The conditions have the same length k as the messages. Instead of being binary encoded strings, a ternary system consisting of the symbols `0`, `1`, and `*` is used. In a condition,

1. `0` means that the corresponding bit in the message must be `0`,
2. `1` means that the corresponding bit in the message must be `1`, and
3. `*` means *don't care*, i. e., the corresponding bit in the message may be `0` as well as `1` for the condition to match.

Definition 7.1 (match**).** A message m matches to a condition c if $\text{match}(m, c)$ evaluates to `true`.

$$\text{match}(m, c) = \ \forall 0 \leq i < |m| \Rightarrow m_{[i]} = c_{[i]} \vee c_{[i]} = * \tag{7.1}$$

The conditional part of a rule may consist of multiple conditions which are implicitly concatenated with logical and ($\wedge$). A classifier is satisfied if all its conditions are satisfied

by at least one message in the current message list. It is allowed that each of the conditions of a classifier may match to different messages.

We can precede each single condition c with an additional ternary digit which defines if it should be negated or not: `*` stands for the negation $\overline{c}$ and `0` as well as `1` denotes c. Here we deviate from the syntax described in Geyer-Schulz [794] because the definition of the "conditionSpecifity" (see Definition 7.2) becomes more beautiful this way. A negated condition evaluates to `true` if no message exists that matches it. By combining `and` and `not`, we get `nand`s with which we can build all other logic operations and, hence, whole computers [2045]. Algorithm 7.1 illustrates how the condition part C is matched against the message list M. If the matching is successful, it returns the list S of messages that satisfied the conditions. Otherwise, the output will be the empty list ().

Algorithm 7.1: $S \longleftarrow \text{matchesConditions}(M, C)$

Input: M: the message list
Input: C: the condition part of a classifier
Input: [implicit] k: the length of the messages $m \in M$ and the single conditions $c \in C$
Input: [implicit] $havePrefix$: `true` if and only if the single conditions have a prefix which determines whether or not they are negated, `false` if no such prefixes are used
Data: i: a counter variable
Data: c: a condition
Data: neg: should the condition be negated?
Data: m: a single message from M
Data: b: a Boolean variable
Output: S: the messages that match the condition part C, or () if none such message exists

```
1  begin
2      S ⟵ ()
3      b ⟵ true
4      i ⟵ 0
5      while (i < len(C)) ∧ b do
6          if havePrefix then
7              neg ⟵ (C[i] = ∗)
8              i ⟵ i + 1
9          else neg ⟵ false
10         c ⟵ subList(C, i, k)
11         i ⟵ i + k
12         if ∃m ∈ M : match(m, c) then
13             b ⟵ ¬neg
14             if b then S ⟵ addListItem(S, m)
15         else
16             b ⟵ neg
17             if b then S ⟵ addListItem(S, createList(k, 0))
18     if b then return S
19     else return ()
20 end
```

Definition 7.2 (Condition Specificity). The condition specificity conditionSpecifity(x) of a classifier x is the number of non-`*` symbols in its condition part $C(x)$.

$$\text{conditionSpecifity}(x) = |\{\forall i : C(x)_{[i]} \neq *\}| \tag{7.2}$$

A classifier (rule) x_1 with a higher condition specificity is more specific than another rule x_2 with a lower condition specificity. On the other hand, a rule x_2 with

conditionSpecifity(x_1) > conditionSpecifity(x_2) is more general than the rule x_1. We can use this information if two rules match to one message, and only one should be allowed to post a message. Preferring the more specific rule in such situations leads to *default hierarchies* [949, 1737, 1739, 1908] which allows general classifications to "delegate" special cases to specialized classifiers. Even more specialized classifiers can then represent exceptions to these refined rules.

7.3.4 Actions

The action part of a rule has normally exactly the same length as a message. It can be represented by a string of either binary or ternary symbols. In the first case, the action part of a rule is simple copied to the message list if the classifier is satisfied. In the latter case, some sort of merging needs to be performed. Here,

1. a `0` in the action part will lead to a `0` in the corresponding message bit,
2. a `1` in the action part will lead to a `1` in the corresponding message bit,
3. and for a `*` in the action part, we copy the corresponding bit from the (first) message that matched the classifier's condition to the newly created message.

Definition 7.3 (*mergeAction*). The function "mergeAction" computes a new message n as product of an action a. If the alphabet the action is based on is ternary and may contain $*$-symbols, mergeAction needs access to the message m which has satisfied the first condition of the classifier to which a belongs. If the classifier contains negation symbols and the first condition was negated, m is assumed to be a string of zeros ($m = \text{createList}(\text{len}(a), 0)$). Notice that we do not explicitly distinguish between binary and ternary encoding in mergeAction, since $*$ cannot occur in actions based on a binary alphabet and Equation 7.3 stays valid.

$$\begin{aligned} n = \text{mergeAction}(a, m) \Leftrightarrow\ & (\text{len}(n) = \text{len}(a)) \wedge \\ & (n_{[i]} = a_{[i]}\ \forall i \in 0..\text{len}(a) - 1 : a_{[i]} \neq *) \wedge \\ & (n_{[i]} = m_{[i]}\ \forall i \in 0..\text{len}(a) - 1 : a_{[i]} = *) \end{aligned} \tag{7.3}$$

7.3.5 Classifiers

So we know that a rule x consists of a condition part $C(x)$ and an action part $a(x)$. C is a list of $r \in \mathbb{N}$ conditions c_i, and we distinguish between representations with ($C = (n_1, c_1, n_2, c_2, \ldots, n_r, c_r)$) and without negation symbol ($C = (c_1, c_2, \ldots, c_r)$). Let us now go back to our frog example. Based on the encoding scheme defined in Figure 7.2, we can translate Table 7.1 into a set of classifiers. We therefore compose the condition parts of two conditions c_1 and c_2 with the negation symbols n_1 and n_2, i. e., $r = 2$. Table 7.2 contains

No.	n_1	c_1	n_2	c_2	a
1	0	0 0 01 0 *** ** * *	0	* * ** * *** ** * *	0 0 00 0 001 00 0 0
2	0	0 0 11 0 *** ** * *	0	* * ** * *** ** * *	0 0 00 0 010 00 0 0
3	0	0 0 10 0 *** ** * *	0	* * ** * *** ** * *	0 0 00 0 011 00 0 0
4	0	1 1 ** * *** ** * *	0	* * ** * *** ** * *	0 0 00 0 100 00 0 0
5	0	* * ** * 001 ** * *	*	* * ** * 100 ** * *	0 0 00 0 000 01 0 0
6	0	* * ** * 010 ** * *	*	* * ** * 100 ** * *	0 0 00 0 000 10 0 0
7	0	* * ** * 011 ** * *	*	* * ** * 100 ** * *	0 0 00 0 000 00 0 1
8	0	* * ** * 100 ** * *	0	* * ** * *** ** * *	0 0 00 0 000 00 1 0

Table 7.2: The encoded form of the `if-then` rules for frogs from Table 7.1.

the result of this encoding. We can apply this classifier to a situation in the life of our frog where it detects

1. a fly to its left,
2. a bee to its right, and
3. a stork left in the air.

How will it react? The input sensors will generate three messages and insert them into the message list $M_1 = (m_1, m_2, m_3)$:

1. $m_1 =$ (000100000000) for the fly,
2. $m_2 =$ (001110000000) for the bee, and
3. $m_3 =$ (110100000000) for the stork.

The first message triggers rule 1 and the third message triggers rule 4 whereas no condition fits to the second message. As a result, the new message list M_2 contains two messages, m_4 and m_5, produced by the corresponding actions.

1. $m_4 =$ (000000010000) from rule 1 and
2. $m_5 =$ (000001000000) from rule 4.

m_4 could trigger rule 5 but is inhibited by the negated second condition c_2 because of message m_5. m_5 matches to classifier 8 which finally produces message $m_6 =$ (000000000010) which forces the frog to jump away. No further classifiers become satisfied with the new message list $M_3 = (m_6)$ and the classification process is terminated.

7.3.6 Non-Learning Classifier Systems

So far, we have described a non-learning classifier system. Algorithm 7.2 defines the behavior of such a system which we also could observe in the example. It still lacks the credit apportionment and the rule discovery systems (see *(f)* and *(g)* in Figure 7.1). A non-learning classifier is able to operate correctly on a fixed set of situations. It is sufficient for all applications where we are able to determine this set beforehand and no further adaptation is required. If this is the case, we can use genetic algorithms to evolve the classifier systems offline, for instance.

Algorithm 7.2 illustrates how a classifier system works. No optimization or approximation of a solution is done; this is a complete control system in action. Therefore we do not need a termination criterion but run an infinite loop.

7.3.7 Learning Classifier Systems

In order to convert this non-learning classifier system to Learning Classifier System as proposed by Holland [943] and sketched in Algorithm 7.3, we have to add the aforementioned missing components. Heitkötter and Beasley [916] suggest two ways for doing so:

1. Currently, the activation of a classifier x results solely from the message-matching process. If a message matches the condition(s) $C(x)$, the classifier may perform its action $a(x)$. We can change this mechanism by making it also dependent on an additional parameter $v(x)$ – a strength value, which can be modified as a result of experience, i. e., by reinforcement from the environment. Therefore, we have to solve the *credit assignment problem* first defined by Minsky [1425, 1428], since chains of multiple classifiers can cause a certain action.
2. Furthermore (or instead), we may also modify the set of classifiers P by adding, removing, or combining condition/action parts of existing classifiers.

A Learning Classifier System hence is a control system which is able to learn while actually running and performing its work. Usually, a training phase will precede any actual deployment. Afterwards, the learning may even be deactivated, which turns the LCS into an ordinary classifier system or the learning rate is decreased.

Algorithm 7.2: nonLearningClassifierSystem(P)

Input: P: the list of rules x_i that determine the behavior of the classifier system
Input: [implicit] readDetectors: a function which creates a new message list containing only the input messages from the detectors
Input: [implicit] sendEffectors: a function which translates all messages concerning effectors to signals for the output interface
Input: [implicit] $t_{max} \in \mathbb{N}$: the maximum number of iterations for the internal loop, avoids endless loops
Data: t: a counter the internal loop
Data: M, N, S: the message lists
Data: x: a single classifier

```
1  begin
2      while true do
3          M ⟵ readDetectors()
4          t ⟵ 0
5          repeat
6              N ⟵ ()
7              foreach x ∈ P do
8                  S ⟵ matchesConditions(M, C(x))
9                  if len(S) > 0 then
10                     N ⟵ addListItem(N, mergeAction(a(x), S[0]))
11             M ⟵ N
12             t ⟵ t + 1
13         until (len(M) = 0) ∨ (t > t_max)
14         if len(M) > 0 then sendEffectors(M)
15 end
```

7.3.8 The Bucket Brigade Algorithm

The Bucket Brigade Algorithm has been developed by Holland [942, 943] as one method of solving the credit assignment problem in Learning Classifier Systems. Research work concerning this approach and its possible extensions has been conducted by Westerdale [2195, 2196, 2197], Antonisse [74], Huang [969], Riolo [1738, 1737], Dorigo [579], Spiessens [1942], Wilson [2234], Holland and Burks [946], and Hewahi and Bharadwaj [922] and has neatly been summarized by Hewahi [920, 921]. In the following, we will outline this approach with the notation of de Boer [507].

The Bucket Brigade Algorithm selects the classifiers from the match set X that are allowed to post a message (i. e., becoming member in the activated set U) by an auction. Therefore, each matching classifier x places a bid $B(x)$ which is the product of a linear function ϑ of the condition specificity of x, a constant $0 < \beta \leq 1$ that determines the fraction of the strength of $x$should be used and its strength $v(x)$ itself. In practical applications, values like $\frac{1}{8}$ or $\frac{1}{16}$ are often chosen for β.

$$B(x) = \vartheta(x) * \beta * v(x) + \text{random}_n\left(0, \sigma^2\right) \tag{7.4}$$

Sometimes, a normal distributed random number is added to each bid in order to make the decisions of the system less deterministic, as done in Equation 7.4.

The condition specificity is included in the bid calculation because it gives a higher value to rules with fewer $*$-symbols in their conditions. These rules match to fewer messages and can be considered more relevant in the cases they do match. For ϑ, the quotient of the number non-$*$-symbols and the condition length plus some constant $0 < \alpha$ determining the importance of the specificity of the classifier is often used [507].

$$\vartheta(x) = \frac{\text{conditionSpecifity}(x)}{\text{len}(C(x))} + \alpha \tag{7.5}$$