Technical Report

Pruning population size in XCS for complex problems

Barbara Rakitsch, Andreas Bernauer, Oliver Bringmann, and Wolfgang Rosenstiel

February 2010

Computer Engineering Department
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen
Sand 13
D-72076 Tübingen, Germany

http://www.ti.uni-tuebingen.de/

©WSI 2010

Pruning population size in XCS for complex problems

Barbara Rakitsch, Andreas Bernauer, Oliver Bringmann, and Wolfgang Rosenstiel

Abstract

In this report, we show how to prune the population size of the Learning Classifier System XCS for complex problems. We say a problem is complex, when the number of specified bits of the optimal start classifiers (the *problem dimension*) is not constant. First, we derive how to estimate an equivalent problem dimension for complex problems based on the optimal start classifiers. With the equivalent problem dimension, we calculate the optimal maximum population size just like for regular problems, which has already been done. We empirically validate our results.

Furthermore, we introduce a subsumption method to reduce the number of classifiers. In contrast to existing methods, we subsume the classifiers after the learning process, so subsuming does not hinder the evolution of optimal classifiers, which has been reported previously. After subsumption, the number of classifiers drops to about the order of magnitude of the optimal classifiers while the correctness rate nearly stays constant.

1 Introduction

System-on-Chips (SoC) provide a high level of system integration and reduced design costs compared to other chip designs. The International Technology Roadmap for Semiconductors (ITRS) [1] expects that the number of SoC designs to increase strongly. However, keeping the effort for SoC designs low becomes more and more difficult. As integration density increases and feature size decreases, the complexity of chip designs increases. Additionally, increasing transistor variability [2, 3], process variation [4] and degradation effects [5] make it even more difficult to manufacture reliable chips [6]. The ITRS estimates a requirement of a design reuse rate of 70% until 2015 [1] to keep SoC designs worthwhile.

Adding Organic Computing properties to a SoC helps to reduce the design effort. With the right set of monitors, evaluators, and actuators, the chip can self-adapt to its actual variability and manifested process variation, self-heal from degradation effects, and self-optimize to its current environment, ensuring an efficient but reliable chip. In [7], the authors have shown that using the Learning

Classifier System XCS [8] as the evaluator allows a chip to self-optimize its operating point to its current environment. In [9], the authors have shown that the XCS also enables the SoC to self-heal from unforeseen core failures.

The Learning Classifier System XCS learns a population of classifiers (or rules) that instruct the SoC what to do in a given situation. During the design process, the necessary number of classifiers to solve a particular problem is needed to estimate the final costs of the SoC, as chip area is one of the main costs in chip manufacturing. Furthermore, for the same reason, it is desirable to keep the number of classifiers on the chip to a minimum. The XCS has a parameter N that defines the maximum allowable size of the classifier population and is thus a natural choice to limit the number of classifiers. Choosing N optimally is crucial: if N is too small, the XCS cannot evolve enough classifiers to solve the problem; if N is too large, the selection pressure on non-optimal classifiers is low, they don't get deleted and the resulting classifier population is thus larger than necessary. Previous work [10, 11] has shown how to select N optimally and how N depends on the *problem dimension* k of the optimal classifiers. The *problem dimension* is the number of bits that have to be set in the beginning classifier population so that the XCS can evolve optimal classifiers. However, the current estimation only applies to regular problems where the problem dimension is constant for all optimal classifiers. Hence, pruning the population size by selecting N optimally is only known for regular problems.

In this report, we show how to prune the population size of XCS for complex problems where the problem dimension of the optimal classifiers is not constant. For this, we make the following major contributions:

1. We present a method to select the maximum population size N for complex problems optimally. We show how to derive the equivalent problem dimension k based on the variable problem dimension k_{cl} of each optimal classifier.

2. We introduce a subsuming method that is applied after learning has finished to further reduce the number of classifiers.

3. We validate our theoretical findings on choosing N optimally with the core-allocation problem, a complex problem first presented in [9].

4. We show that our subsuming method avoids the disruptive effect of subsumption reported in [10] yet retains the property of self-adaptation to unforeseen events.

This work is structured as follows. Section 2 shows related work. Section 3 briefly introduces the XCS and the previous work on estimating N optimally for regular problems. Section 4 presents our method to estimate an equivalent problem dimension k and how to select N optimally for complex problems. Section 5 presents our subsumption method after learning. Section 6 describes the experimental setup

and shows how to apply our methods to the core allocation problem. Section 7 presents the results of our validation and Section 8 concludes.

2 Related work

The content of this report is closely linked to [11], and [10] in regard of the theoretical aspects of the XCS and to [9] in our use of the XCS as a generic self-adaptation system. The LCS has been used in various Organic Computing projects. In [12], the LCS is part of a three-layered organic controller for traffic lights and running in embedded hardware. In [13], the authors strive for a minimal set of classifiers that can evolve a cooperating communication structure in autonomous agents with limited resources. In [14], Richter analyzes the LCS as part of the generic observer/controller architecture [15] for organic computing systems and how the emergent patterns can be controlled. In [16], the authors propose a hardware implementation of an LCS that uses little resources but retains the adaptation capabilities of the LCS.

3 XCS in a nutshell

In this section, we briefly explain the XCS. A more detailed description can be found in [17] and [18].

The XCS is a learning classifier system; it consists of a population $[P]$ of classifiers, and each classifier consists of a condition $C \in \{0, 1, \#\}^L$ of length L, an action $a \in \{a_i, \dots, a_n\}$, a reward prediction p, a fitness value F, a numerosity num, an action set size estimation as and some other house keeping values.

The classifiers adapt to the environment through reinforcement learning and are altered by a genetic algorithm (GA). In contrast to most other learning classifier systems, the accuracy, and not the predicted reward, of the classifier determines its fitness.

One learning step of the XCS looks as follows: The XCS receives the input state $s \in \{0, 1\}^L$ from the environment and places the classifiers whose conditions match s into the match set $[M]$. If one action a is not represented in $[M]$, *covering* occurs; that means a classifier whose condition matches s with action a is generated. A bit in the condition is set to the don't-care symbol $\#$ with probability $P_\#$, otherwise it is set matching the corresponding bit in s. Then, the XCS calculates a system prediction $P(a_i)$ for each action a_i. In explore mode it chooses an action randomly; in exploit mode it chooses the action with the best system prediction. All classifiers which represent this action are put into the action set $[A]$. The action is applied, and the environment returns a reward to the XCS. With the help of this feedback, the XCS adjusts some of the parameters of the classifiers in $[A]$.

The GA working on $[A]$ is applied when the average time since the last GA in $[A]$ has been applied exceeds a certain threshold θ_{GA}. The two parent individu-

als are randomly selected by their fitness and then duplicated. The resulting new classifiers are altered by cross-over and mutation and then inserted in $[P]$.

If the maximal population size N is exceeded, classifiers are deleted throughout the whole population. The deletion probability of a classifier is proportional to its action set size as. If the classifier's fitness is significantly lower than the average, but its experience is large enough, its deletion probability is further increased.

The reproduction of new classifiers in $[A]$ by the genetic algorithm and the deletion of classifiers in $[P]$ results in accurate, maximally general classifiers as proposed in Wilson's Generality Hypothesis [17]. In the following, we call these accurate, maximally general classifiers *optimal classifiers*. The optimal classifiers can only evolve when the *Covering Challenge* [11], the *Schema Challenge* [11], and the *Reproductive Opportunity* [10] are met:

3.1 Covering Challenge

The XCS needs a sufficient number of classifiers in order to cover all input states. When an input state is not covered and N classifiers already exist, the XCS must delete an existing classifier to make room for a new covering classifier. If this occurs too often, the XCS is not able to evolve optimal classifiers. To avoid too many deletions, each input state must be matched by at least one classifier after the population is filled up. Hence, the probability $P(cover)$ of at least one classifier covering a certain input state must be high enough [11]:

$$P(cover) = 1 - \left(1 - \left(\frac{1+P_{\#}}{2}\right)^{L}\right)^{N} \tag{1}$$

Clearly, setting $P_{\#}$ and N high enough guarantees that $P(cover)$ is high enough.

3.2 Schema Challenge

The GA needs sufficiently accurate start classifiers to evolve optimal classifiers. Therefore, the conditions of the classifiers must contain enough specified bits (i.e., bits that are not the don't-care symbol). As mentioned in the introduction, we call the number of specified bits the *problem dimension*. For now, we assume that at least k bits must be specified. In the next section, we explain how to estimate k for complex problems.

If the probability $P(representative)$ that each start classifier hast at least k bits specified is high enough, then the challenge is met [11]:

$$P(representative) = 1 - \left(1 - \frac{1}{n}\left(\frac{1-P_{\#}}{2}\right)^{k}\right)^{N} \tag{2}$$

By setting $P_{\#}$ low and N high enough, $P(representative)$ becomes sufficiently high. Note that the Covering and the Schema Challenge ask for different settings for $P_{\#}$.

3.3 Reproductive Opportunity

While the Schema Challenge ensures that the initial classifiers are sufficiently accurate, the Reproductive Opportunity ensures that the sufficiently accurate classifiers have a chance to reproduce. Therefore, the probability for a classifier cl taking part in $[A]$ must be greater than its probability for being deleted [10]:

$$\frac{1}{n}\left(\frac{1}{2}\right)^{L \cdot s_{cl}} > \frac{2}{N} \tag{3}$$

The specificity s_{cl} can be calculated as follows:

$$s_{cl} = \frac{k + (L-k)s([P])}{L}.$$

The average specificity in the population $s([P])$ is estimated by the mutation rate μ. The relation between $(s([P])$ and μ is derived in [10].

The Reproductive Opportunity is taken by setting N high enough.

4 Estimating the problem dimension k

For solving the Schema Challenge and the Reproductive Opportunity, one needs to know the problem dimension k. In earlier papers like [10], it was simple to specify k since all start classifiers needed the same number of specified bits and the classifiers were not overlapping.

In this section, we show how to estimate the equivalent problem dimension k for complex problems including overlapping classifiers and different k's for different classifiers.

As the population size N and the number of required learning steps grow with the size of k [19], we want k to be as small as possible but still large enough to ensure that all optimal classifiers can evolve. If we neglected the requirement that k should be as small as possible, we could simply set k to the maximum of all occurring k_{cl}s.

$$k_{max} = \max_{cl \in [P]} (k_{cl}) \tag{4}$$

As the classifiers with the most specified bits are most difficult to evolve, k_{max} ensures that all optimal classifiers can evolve.

When we want to work with a smaller k, we have to risk that not all optimal classifiers can evolve. Hence, we have to decide which classifiers are important to solve the problem. Therefore, we give each optimal start classifier a weight w_{cl} according to its importance and calculate k as the weighted average.

Our weight function is based on following ideas:

1. A classifier matching only infrequent system inputs is not as important as a classifier matching frequent system inputs, as it seldom contributes to the overall performance.

2. Only the classifiers that gain the best reward are applied in the exploit mode. Proportional to the probability $P(exploit)$ that the XCS runs in the exploit mode, classifiers that gain a high reward become more important.

3. When an action is propagated by many classifiers, it is not important that all of the classifiers exist.

Putting all things together, we can estimate k, under the assumption that $[P]$ only consists of optimal classifiers, as

$$k_{avg} = \sum_{cl \in [P]} \frac{w_{cl}}{w} k_{cl} \tag{5}$$

with the weights

$$\begin{aligned} w_{cl} &= P(\text{explore}) \frac{1}{|[A]|} P(cl \text{ in } [A] | \text{explore}) k_{cl} \\ &+ P(\text{exploit}) \frac{1}{|[A]|} P(cl \text{ in } [A] | \text{exploit}) k_{cl} \end{aligned}$$

and the total weight

$$w = \sum_{cl \in [P]} w_{cl}.$$

As the XCS works either in explore or in exploit mode, $P(\text{exploit}) = 1 - P(\text{explore})$ holds. The probability $P(cl \text{ in } [A] | \text{explore})$ is $\frac{1}{n}$ because an action is chosen randomly from all actions in the explore mode; whereas $P(cl \text{ in } [A] | \text{exploit})$ is one divided by the number of optimal classifiers which earn the highest reward.

With the equivalent problem dimension k_{avg} and the formulae (1), (2), and (3), we can now approximate N.

5 Subsuming the classifiers after learning

After learning, we expect all the important, accurate, and maximally general classifiers to exist. Additionally, there are inaccurate and too specific classifiers since the GA still evolves new classifiers.

In the following, we introduce a method which selects the optimal classifiers from all the others. With this, we can reduce the population size N considerably.

The algorithm consists of two steps:

1. Subsuming:
   Let cl_1 and cl_2 be two classifiers with the same action. The classifier cl_1 subsumes cl_2 if the following conditions are fulfilled:

   (a) The condition of cl_1 covers the condition of cl_2.

(b) cl_1 is at least as accurate as cl_2.

(c) cl_1 is experienced enough.

2. Deletion of superfluous classifiers:
   A classifier is not required if

   (a) other clas sifiers also cover its condition and

   (b) it is not accurate or not experienced enough

   and is thus deleted.

The algorithm works similarly to the methods *GA Subsumption* and *Action Set Subsumption*, which are described in [18]. Since the latter two are applied during learning, they allow over-general, short-time accurate classifiers to subsume accurate classifiers [10] and thus, hinder the evolution of optimal classifiers and cause an extended learning time or even poor performance. As our method is applied after learning, these disruptive effects do not occur.

6 Experimental setup

As test system for our estimation of k_{avg} and N as well as for our subsuming method, we use the core-allocation problem described in [9].

The core-allocation problem (L, i) is defined as follows: We want to allocate i cores out of L cores while some of the L cores are already occupied. The system input s is a binary string of length L with each bit representing one core. If a bit is set to zero, the core is free; if it is set to one, it is occupied. There is one action for each possible allocation of i cores out of L cores and one action for the case when no allocation is possible. Hence, there is a total of $\binom{L}{i} + 1$ actions. The action "no valid allocation is possible" is encoded as 0, all other actions are encoded with the combinadic function [20].

We choose the core-allocation problem as test system, because, depending on L and i, the number and structure of the optimal classifiers vary considerably which gives us the opportunity to test k_{avg} as the equivalent problem dimension for various cases.

In the following, we consider a correctness rate of over 90% as good enough for our purposes.

6.1 Optimal Classifiers

Let us first have a look at a classifier with an action $\neq 0$. The classifier receives a reward of 1000, if the allocation encoded by the action is free. Therefore, all cores from the allocation must be free (see Table 1). If at least one of the specified i cores is occupied, the classifier receives no reward. Hence, for each action $\neq 0$ $i + 1$ optimal classifiers exist: one classifier whose condition has all bits, specified

by the allocation, set to 0 and i classifiers whose conditions each have one of the specified bits set to one, the other are set to $\#$.

Let us now have a closer look at action zero. A classifier which represents action zero gains a reward of 1000 if no allocation is possible. Therefore, at least $L-i+1$ cores must be occupied. If at least i cores are free, the classifier gains no reward. Thus, $\binom{L}{L-i+1}$ classifiers exist whose conditions each have $L-i+1$ different bits set to 1 and $\binom{L}{i}$ classifiers whose conditions each have i different bits set to 0.

6.2 Special case: Action Zero

A classifier with action zero receives a reward of 1000 if actually no valid action is possible. Therefore, the probability is:

$$P(\text{Action 0 gets reward 1000}) = \sum_{j=L-i+1}^{L} \binom{L}{j} 2^{-L} \tag{6}$$

Table 2 displays the probabilities for the individual problems for $1 \leq i \leq L \leq 10$. As expected, action zero is the best action when many cores have to be allocated (large i). The bold numbers indicate that action zero gains a reward of 1000 for at least 90% of all cases. In these problems, the optimal classifiers are not required since the XCS can simply always choose action zero. Therefore, it needs only one classifier per action whose condition consists only of don't-care terms. Since our estimation of k builds on the assumption that optimal classifiers are required, we will estimate a problem dimension too large for these problems. We see in the next subsection, that the corresponding k values are small, so our estimation is not far away from the optimum.

6.3 Estimating the problem dimension k

Table 3: Probability distributions for the problem $(4, 3)$, with $p_{explore} = \frac{1}{|[A]|} P(cl \text{ in } [A]|\text{explore})$ and $p_{exploit} = \frac{1}{|[A]|} P(cl \text{ in } [A]|\text{exploit})$

s	cl in $[M]$ $cl.C$	$cl.a$	$cl.p$	$p_{explore}$	$p_{exploit}$
0000	0 0 0#	0	0.0	0.05	0.00
	0 0#0	0	0.0	0.05	0.00
	0#0 0	0	0.0	0.05	0.00
	#0 0 0	0	0.0	0.05	0.00
	0 0 0#	1	1000.0	0.20	0.25
	0 0#0	2	1000.0	0.20	0.25
	0#0 0	3	1000.0	0.20	0.25

s	cl in $[M]$			$p_{explore}$	$p_{exploit}$
	$cl.C$	$cl.a$	$cl.p$		
	#0 0 0	4	1000.0	0.20	0.25
0001	0 0 0#	0	0.0	0.20	0.00
	0 0 0#	1	1000.0	0.20	1.00
	###1	2	0.0	0.20	0.00
	###1	3	0.0	0.20	0.00
	###1	4	0.0	0.20	0.00
⋮					
0011	##1 1	0	1000.0	0.20	1.00
	##1#	1	0.0	0.20	0.00
	###1	2	0.0	0.20	0.00
	##1#	3	0.0	0.10	0.00
	###1	3	0.0	0.10	0.00
	##1#	4	0.0	0.10	0.00
	###1	4	0.0	0.10	0.00
⋮					
0111	#1 1#	0	1000.0	0.07	0.33
	#1#1	0	1000.0	0.07	0.33
	##1 1	0	1000.0	0.07	0.33
	#1##	1	0.0	0.10	0.00
	##1#	1	0.0	0.10	0.00
	#1##	2	0.0	0.10	0.00
	###1	2	0.0	0.10	0.00
	##1#	3	0.0	0.10	0.00
	###1	3	0.0	0.10	0.00
	#1##	4	0.0	0.07	0.00
	##1#	4	0.0	0.07	0.00
	###1	4	0.0	0.07	0.00
⋮					
1111	1 1##	0	1000.0	0.03	0.17
	1#1#	0	1000.0	0.03	0.17
	1##1	0	1000.0	0.03	0.17
	#1 1#	0	1000.0	0.03	0.17
	#1#1	0	1000.0	0.03	0.17
	##1 1	0	1000.0	0.03	0.17
	1###	1	0.0	0.07	0.00
	#1##	1	0.0	0.07	0.00
	##1#	1	0.0	0.07	0.00
	1###	2	0.0	0.07	0.00
	#1##	2	0.0	0.07	0.00
	###1	2	0.0	0.07	0.00

s	cl in $[M]$			$p_{explore}$	$p_{exploit}$
	$cl.C$	$cl.a$	$cl.p$		
	1###	3	0.0	0.07	0.00
	##1#	3	0.0	0.07	0.00
	###1	3	0.0	0.07	0.00
	#1##	4	0.0	0.07	0.00
	##1#	4	0.0	0.07	0.00
	###1	4	0.0	0.07	0.00

We use k_{avg} as k. To identify $|[A]|$ and $P(cl \text{ in } [A]|\text{exploit})$, we first search all matching optimal classifiers for each system input s.

Then, we can calculate $|[A]|$ and $P(cl \text{ in } [A]|\text{exploit})$ for each system input s (see Table 3). As all system inputs are equally probable, we get the overall values by simply adding up the single values. $P(\text{explore})$ is set to 0.2.

Table 4 displays the estimated k values. The estimated ks for the problem (L, i) and $(L, L - i + 1)$ are approximately the same, as a valid allocation for (L, i) is only possible if and only if no allocation is found for $(L, L - i + 1)$.

Comparing the estimations k_{avg} and k_{max}, k_{avg} is on average 4.4 smaller than k_{max}, resulting in considerably smaller population sizes.

6.4 Estimating the population size N

We calculate N with k from Table 4.

N shall be as small as possible, but large enough to fulfill the formulae (1), (2) and (3). We say that the challenges are fulfilled when the probabilities $P(cover)$ and $P(representative)$ are at least 0.9. Therefore, the don't-care probability $P_{\#}$ is chosen from $\{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, so that it fits best.

Table 5 shows the calculated population sizes.

7 Results

We run all experiments with $1 \leq i \leq L \leq 10$.

First, we want to show that our k estimations are accurate. Therefore, since k is proportional to N, we let the experiment set run with the newly calculated population size N, with $\frac{3}{4}N$ and with $\frac{1}{2}N$. When the XCS with the population size N reaches a correctness rates of over 90%, then the k values are large enough. When the performance with smaller population sizes declines, our k values must not be smaller.

Next, we want to show that our subsumption method works. In order to do so, we take the classifiers from the previous experiments with population size N and apply the subsumption method to them. Then, we analyze the performance of the subsumed classifiers.

Table 1: Optimal classifiers for the problem $(4, 3)$

condition C	action a	reward p
000#	1	1000
1###	1	0
#1##	1	0
##1#	1	0
00#0	2	1000
1###	2	0
#1##	2	0
###1	2	0
0#00	3	1000
1###	3	0
##1#	3	0
###1	3	0
#000	4	1000
#1##	4	0
##1#	4	0
###1	4	0
11##	0	1000
1#1#	0	1000
1##1	0	1000
#11#	0	1000
#1#1	0	1000
##11	0	1000
000#	0	0
00#0	0	0
0#00	0	0
#000	0	0

Table 2: P(Action 0 gets reward 1000) for the problem (L, i)

$i \backslash L$	1	2	3	4	5	6	7	8	9	10
1	.5	.25	.12	.06	.03	.02	.01	.0	.0	.0
2		.75	.5	.31	.19	.11	.06	.04	.02	.01
3			.88	.69	.5	.34	.23	.14	.09	.05
4				**.94**	.81	.66	.5	.36	.25	.17
5					**.97**	.89	.77	.64	.5	.38
6						**.98**	**.94**	.86	.75	.62
7							**.99**	**.96**	**.91**	.83
8								**1.0**	**.98**	**.95**
9									**1.0**	**.99**
10										**1.0**

Table 4: Estimated k for the core-allocation problem (L, i)

$i \backslash L$	1	2	3	4	5	6	7	8	9	10
1	1.0	1.4	1.5	1.5	1.5	1.4	1.3	1.2	1.1	1.1
2		1.4	1.9	2.3	2.5	2.6	2.5	2.4	2.3	2.2
3			1.6	2.3	2.9	3.3	3.5	3.6	3.6	3.5
4				1.6	2.5	3.3	3.8	4.2	4.5	4.6
5					1.6	2.6	3.5	4.2	4.8	5.2
6						1.4	2.6	3.6	4.5	5.2
7							1.3	2.4	3.6	4.6
8								1.2	2.3	3.5
9									1.1	2.2
10										1.1

Table 5: Estimated N for the core-allocation problem (L, i)

$_i\backslash^L$	1	2	3	4	5	6	7	8	9	10
1	11	23	37	48	68	91	121	159	208	274
2		16	54	132	250	384	616	945	1391	2003
3			19	97	354	962	2030	3493	5942	9927
4				20	141	741	2800	7969	17888	33037
5					24	174	1258	6426	24176	70198
6						27	228	1800	12039	58653
7							31	297	2600	19036
8								36	378	3782
9									43	480
10										51

Finally, we test if the subsumed classifiers can deal with unforeseen events that require self-adaptation. Therefore, we let cores fail without the environment noticing it.

The performance of the XCS is measured with the averaged correctness rate over the last 50 exploitation steps. Repeated simulations show the same results as presented in the following subsections and lead to the same conclusions.

We use the C implementation from [21], version 1.2. The parameters are set to the default values, except: $\alpha = 1.0$, $\epsilon_0 = 0.01$, $\theta_{GA} = 250$, $\mu_{GA} = 0.1$, $\chi_{GA} = 0.5$; *Action Set* and *GA* subsumption are turned off. N and $P_\#$ are calculated individually as described before.

7.1 Core allocation problem during learning

Figure 1 shows the correctness rates depending on N. The y-axis depicts the correctness rate, the x-axis the three different experiment sets with the population sizes N, $0.75N$ and $0.5N$. We can see that with the calculated population size N, all experiments have a correctness rate of at least 90%. When we reduce the population size to $0.75N$, the results are slightly worse; about 30% of the problems have a correctness rate between 80% and 90%. When we decrease the population size further to $0.5N$, only 24% of the experiments have a correctness rate of over 90%. In 20% of the experiments, the XCS has severe difficulties in choosing the right action; the correctness level drops below 70%.

In 6.2, we calculated $P(\text{Action } 0 \text{ gets reward } 1000)$. 13 from 55 problems have a probability greater than 0.9 (see Table 2) meaning that they have a correctness rate of over 90% when the XCS simply chooses action zero all the time. Since

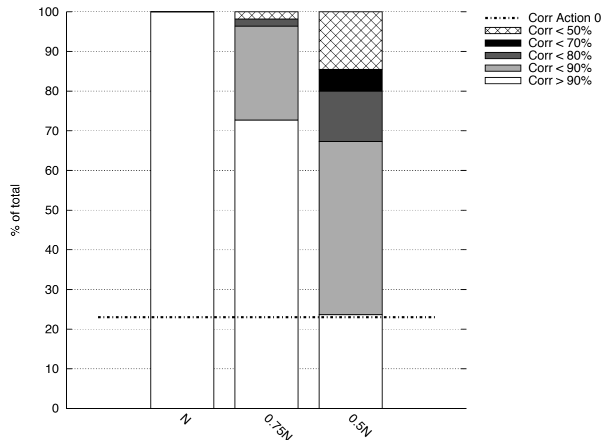

Figure 1: Correctness of the core-allocation problem depending on N

these problems can be solved without the optimal classifiers, we assume that they can also be solved with a much smaller population size than calculated and indeed, these problems reach a correction rate of over 90% in all three experiment sets. We indicate those problems with the black line at the correctness level of 24%. Hence, if we exclude those problems from our observations, no problem from the experiment set with population size $0.5N$ reaches a correctness level of over 90%.

We conclude that the XCS is able to learn all of the problems with our N estimations, but is not able anymore to learn all problems when we reduce each N to one half. That means that our N and thus also our k estimations are large enough to ensure reliability, but not too large since the performance declines when we reduce N.

7.2 Subsuming of classifiers

We apply our subsuming method on the experiment set with normal population size N after learning.

Figure 2 displays the number of classifiers for the problems with ten cores. The x-axis refers to the problems and the y-axis to the number of classifiers on a logarithmic scale. We use a logarithmic scale because the number of classifiers before subsuming is much larger than afterwards. The white bar depicts the number of classifiers before subsuming, the gray bar after subsuming, the black bar shows the number of classifiers from [9], and the patterned bar refers to the number of optimal classifiers. In [9], the XCS is applied to the same problem instances with the two methods *GA subsumption* and *Action Set Subsumption* activated; they both subsume during learning.

We can see that the subsuming method reduces the number of classifiers significantly. For the problems $(10, i), i = 1, \ldots, 6$, the number of classifiers after subsuming is a bit larger than the number of optimal classifiers.

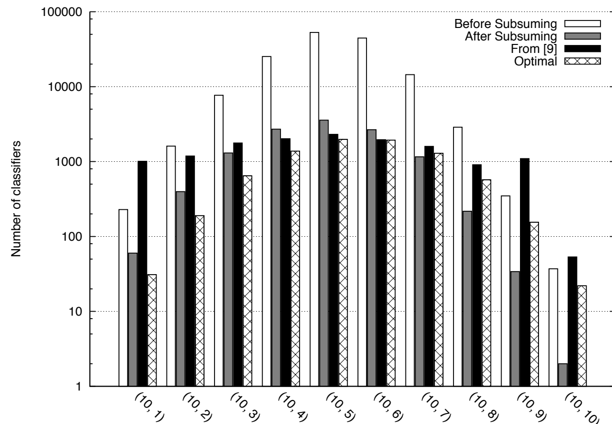

Figure 2: Number of classifiers

For the problems $(10, i), i = 7, \ldots, 10$, the number of subsumed classifiers is even smaller than the number of optimal classifiers. Once again, this can be attributed to action zero. With i close to L, $P(\text{Action } 0 \text{ gets reward } 1000)$ increases (see Table 2). Because more and more input states require action zero, not all optimal classifiers are still required, and thus the number of classifiers drops below the number of optimal classifiers.

As we can see, for problems with small and large i, the XCS with the settings from [9] requires more classifiers than we need after subsuming, but less classifiers than we need before subsuming. For medium-sized i, the number of classifiers is nearly the same for subsuming during and after learning.

The results remain the same for problems with $L < 10$.

In the worst case, problem $(2, 1)$, 17% of the classifiers, in the best case, problem $(10, 10)$, even 95% of all classifiers are superfluous. On average, the number of classifiers can be reduced to 27% of classifiers before subsuming.

To sum up, subsuming after learning reduces the number of classifiers to the dimension of optimal classifiers. When we use the methods *GA subsumption* and *Action Set Subsumption*, we need at least as many classifiers as we require when we subsume after learning. For many problem instances, subsuming after learning reduces the number of classifiers more than subsuming during learning. In the next section, we will see, that our newly introduced subsuming method additionally offers a higher level of reliability.

7.3 Core allocation problem after subsuming

In the next experiment set, we use the classifiers from Figure 2 as initial classifiers.

In addition, we turn the GA off; this has several reasons: First, the XCS does not need the GA component anymore, so we can save space. Second, since we assume that we already have optimal classifiers, there is no need to search any

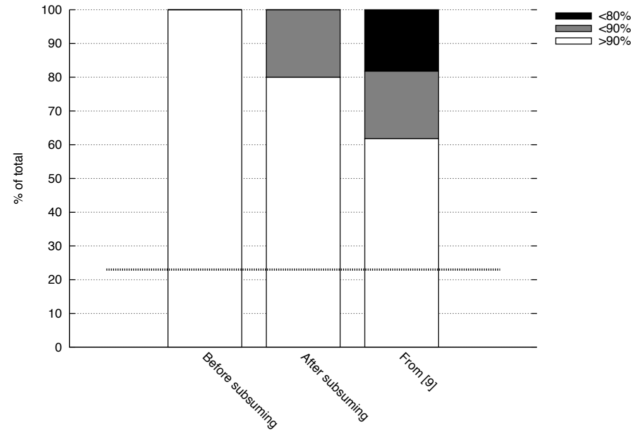

Figure 3: Correctness with subsumed classifiers

further. Since the classifiers can still adjust their reward, they are also still able to adapt to unforeseen events as we will see in the next section. Third, if we allow the GA to take place, the classifiers are not static; new classifiers are discovered and old ones are deleted. Hence, if we turn the GA off, no good classifier can be lost, and we do not need additional space for the new classifiers that the GA discovers.

The Figure 3 shows the correctness rates without subsuming, with subsuming after learning and with subsuming with the methods *GA subsumption* and *Action Set Subsumption* [9].

Obviously, the performance is best without subsuming. As we can see in the chart, when we subsume after learning, the XCS worsens slightly since in 20% of the problems the correctness rate is between 80% and 90%. When we subsume during learning, the correctness rate is in 20% of the problems between 80% and 90%, in 20% even below 80%.

In conclusion, subsuming after learning requires at least as many classifiers as subsuming during learning, but the performance remains on a higher level. Comparing the results between no subsuming at all and subsuming after learning, we come to accept the slight loss of performance in favor of reducing the number of classifiers.

7.4 Self-adaptation to failing cores

Next, we analyze how well the XCS can self-adapt after subsuming. In [9] is shown that the XCS is able to self-adapt when *GA subsumption* and *Action Set Subsumption* are activated. We test now if the XCS maintains this capacity when the newly introduced subsuming method is used instead.

Therefore, one or more cores fail, and the XCS has to adapt to the new situation. A failed core is still monitored as free, but cannot be allocated.

We take the same experimental setup as in Section 7.3. The XCS runs the first

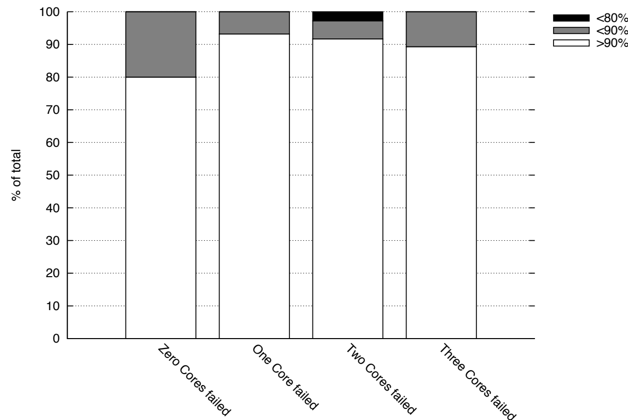

Figure 4: Correctness with core failure

150000 time steps normally before the first core fails. After another 100000 time steps, the next core fails and so forth. In the problem (L, i) maximum $L - i$ cores can fail since there is no valid allocation possible otherwise.

If for a problem all optimal classifiers exist, the core failure does not influence the performance of the XCS; the reward of the classifiers which want to allocate the failed core simply adjusts to zero.

As we see in Figure 4, the performance of the XCS remains nearly the same after one core has failed. The performance is even slightly better compared to before; the number of problems whose correctness rate is below 90% drops from 11 to below 5. The average overall performance is between 94% and 97% in all experiment sets. Hence, the new subsuming method does not affect the self-adaptation capacities of the XCS.

8 Conclusion

In this report, we showed how to estimate the population size for complex problems and how to reduce the number of classifiers after learning.

Butz et al. already estimated the population size for regular problems [10]. We showed how k can be approximated for complex problems as a weighted average of all occurring problem dimensions. The weights are calculated according to how often the classifiers can be applied, how large the rewards they gain are and with how many classifiers they are in the action set. We showed that the performance of the XCS with our estimated population size N is always above 90% with an average value of 96%. When we reduce the population size by one half, the correctness rate drops severely.

Since the classifiers which evolve during learning are only partially required - some are too specific and thus already covered by more general classifiers and

others are not accurate as they are newly discovered in the GA - we introduced a subsuming method to eliminate the superfluous classifiers. In so doing, we could reduce the number of required classifiers to 27% and needed at most slightly more classifiers than optimal classifiers exist.

After subsuming, we decided to turn off the GA, so we could work with a fixed number of classifiers and without having to fear to loose any good classifiers. A few problems had a performance loss of at least 5%, but for the majority of problems the performance did not suffer at all. We accept the slight degradation of performance in favor of reducing the number of classifiers.

Then, we showed that even without the GA the subsumed classifiers are able to adapt to unforeseen events because the classifiers can adapt to failing cores with only adjusting their rewards.

To sum up, the results show that the population size of the XCS for learning can be estimated even for complex problems and the number of classifiers after learning can be reduced to the dimension of the number of optimal classifiers.

Acknowledgment

B.R. thanks Meike Sprecher for her help in constructing English sentences, as correct as possible.

References

[1] International Roadmap Committee, “International Technology Roadmap for Semiconductors,” http://www.itrs.net/reports.html, 2008.

[2] K. Bernstein, D. Frank, A. Gattiker, W. Haensch, B. Ji, S. Nassif, E. Nowak, D. Pearson, and N. Rohrer, “High-performance CMOS variability in the 65-nm regime and beyond,” *IBM Journal of Research and Development*, vol. 50, no. 4/5, p. 433, 2006.

[3] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” in *DAC ’03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 338–342.

[4] A. Agarwal, V. Zolotov, and D. Blaauw, “Statistical clock skew analysis considering intra-die process variations,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1231 –1242, 2004.

[5] C. Schlunder, R. Brederlow, B. Ankele, A. Lill, K. Goser, R. Thewes, I. Technol, and G. Munich, “On the degradation of p-MOSFETs in analog and RF

circuits under inhomogeneous negative bias temperature stress," in *41st Annual IEEE International Reliability Physics Symposium Proceedings*, 2003, pp. 5–10.

[6] V. Narayanan and Y. Xie, "Reliability Concerns in Embedded System Designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006. [Online]. Available: http://www.cse.psu.edu/~yuanxie/Papers/ComputerMag2006.pdf

[7] A. Bernauer, D. Fritz, and W. Rosenstiel, "Evaluation of the Learning Classifier System XCS for SoC run-time control," in *Lecture Notes in Informatics*, vol. 134, Gesellschaft für Informatik. Springer, 10 2008, pp. 761–768.

[8] S. W. Wilson, "Generalization in the XCS Classifier System," 1998.

[9] A. Bernauer, O. Bringmann, and W. Rosenstiel, "Generic Self-Adaptation to Reduce Design Effort for System-on-Chip," in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, San Francisco, 09 2009, pp. 126–135.

[10] M. V. Butz, D. E. Goldberg, and K. Tharakunnel, "Analysis and improvement of fitness exploitation in XCS: bounding models, tournament selection, and bilateral accuracy," *Evol. Comput.*, vol. 11, no. 3, 2003.

[11] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson, "Toward a Theory of Generalization and Learning in XCS," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 1, 2004.

[12] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer, and H. Schmeck, "Organic control of traffic lights," in *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, ser. LNCS, C. Rong, M. G. Jaatun, F. E. Sandnes, L. T. Yang, , and J. Ma, Eds., vol. 5060. Springer, 2008, pp. 219–233.

[13] A. Scheidler and M. Middendorf, "Evolved cooperation and emergent communication structures in learning classifier based organic computing systems," in *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2009, pp. 2633–2640.

[14] U. Richter, "Controlled self-organisation using learning classifier systems," PhD thesis, Institut für Angewandte Informatik und Formale Beschreiungsverfahren, Universität Karlsruhe (TH), Karlsruhe, Germany, 2009.

[15] C. Müller-Schloer, "Organic computing: on the feasibility of controlled emergence," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS 2004)*. ACM New York, NY, USA, 2004, pp. 2–5.

[16] J. Zeppenfeld, A. Bouajila, W. Stechele, and A. Herkersdorf, "Learning Classifier Tables for Autonomic Systems on Chip." in *GI Jahrestagung (2)*, ser. LNI, H.-G. Hegering, A. Lehmann, H. J. Ohlbach, and C. Scheideler, Eds., vol. 134. GI, 2008, pp. 771–778.

[17] S. W. Wilson, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, no. 2, 1995.

[18] M. V. Butz and S. W. Wilson, "An Algorithmic Description of XCS," in *IWLCS '00: Revised Papers from the Third International Workshop on Advances in Learning Classifier Systems*, London, 2001.

[19] M. V. Butz, D. E. Goldberg, and P. L. Lanzi, "Bounding Learning Time in XCS," Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, IlliGAL report 2004003, 2004.

[20] J. McCaffrey, "Generating the mth Lexicographical Element of a Mathematical Combination," CTAN: http://msdn.microsoft.com/en-us/library/aa289166\%28VS.71\%29.aspx, 2005.

[21] M. V. Butz, "An Implementation of the XCS classifier system in C," The Illinois Genetic Algorithms Laboratory, Tech. Rep. 99021, 1999.

Appendix

Derivation of the optimal start classifiers

In our report, for the core allocation problem (L, i) we set the optimal start classifiers equivalent to the optimal classifiers; the optimal start classifiers must ensure that the optimal classifiers can evolve. In the following, we will prove that the optimal classifiers for the core allocation problem (L, i) can only evolve out of themselves:

Let us consider a classifier cl_1 having i_1 zeros on the positions corresponding to the allocation with $i_1 < i$. We assume further that all other classifiers with the same action have less zeros on the corresponding positions. We would like that the GA selects cl_1 because it is closest to the optimal classifier since it shares more positions with the corresponding optimal classifier than any other classifier in $[A]$. Hence, cl_1 needs fewer alterations to evolve to the optimal classifier and is thus the best choice as parental individuum. Therefore, the fitness of cl_1 must be greater than the fitness of all other classifiers in $[A]$. Since fitness behaves inversely to the reward prediction error ϵ, $cl_1.\epsilon$ must be smaller than the prediction errors of all other classifiers in $[A]$.

In the following, we will show that cl_1 has the largest prediction error ϵ in $[A]$, and is hence not selected in the GA, which explains why it is unlikely that the optimal classifiers evolve from classifiers with less zeros.

In [10] Butz et al. show that the prediction error ϵ of a classifier can be calculated as follows when only two reward levels (0 and 1000) are possible:

$$cl.\epsilon = 2000(P_c(cl) - P_c^2(xl)).$$

$P_c(cl)$ is the probability of the classifier cl predicting the correct reward. As we can see, $cl.\epsilon$ acts like a reverse parable with its maximum at $P_c(cl) = 0.5$.

The classifier cl_1 only gains the reward 1000 when all other $i - i_1$ bits are zero:

$$P_c(cl_1) = \frac{1}{2^{i-i_1}} \leq \frac{1}{2}$$

All other classifiers have at least $i_2 = i_1 - 1$ zeros on the corresponding positions. Consider a classifier cl_2 with i_2 zeros:

$$P_c(cl_2) = \frac{1}{2^{i-i_2}} = \frac{1}{2^{i-(i_1-1)}} = \frac{1}{2}\frac{1}{2^{i-i_1}} = \frac{1}{2}P_c(cl_1)$$

As $P_c(cl_2) < P_c(cl_1) \leq \frac{1}{2}$ holds, $cl_2.\epsilon < cl_1.\epsilon$. Consequently, cl_1 has the largest prediction error in $[A]$.

The same applies for the optimal classifiers with $(L - i + 1)$ ones.

Results in Numbers

The following table lists the numerical results for the problem (L, i) with $1 \leq i \leq L \leq 10$. The shortcut bs stands for before subsuming, as for after subsuming, opt for optimal and cl for classifiers.

(L, i)	N	p	corr(N)	corr($.75N$)	corr($.5N$)	corr(as)	$\#_{cl}(bs)$	$\#_{cl}(as)$	$\#_{cl}(opt)$
(1,1)	11	.2	1.0	1.0	**0.82**	1.0	5	4	4
(2,1)	23	.2	1.0	0.95	**0.86**	1.0	12	10	7
(2,2)	16	.2	0.97	0.95	**0.78**	1.0	8	6	6
(3,1)	37	.2	0.99	**0.88**	**0.75**	**0.87**	18	6	10
(3,2)	54	.2	0.99	**0.8**	**0.74**	1.0	28	15	15
(3,3)	19	.2	0.92	0.9	**0.88**	**0.88**	10	7	8
(4,1)	48	.2	0.9	**0.76**	**0.43**	0.99	40	26	13
(4,2)	132	.2	0.99	0.91	**0.71**	1.0	72	30	28
(4,3)	97	.2	0.95	**0.88**	**0.76**	1.0	63	24	26
(4,4)	20	.2	0.95	0.93	0.93	0.95	13	7	10
(5,1)	68	.3	0.98	**0.86**	**0.43**	0.97	48	19	16

(L,i)	N	p	corr(N)	corr($.75N$)	corr($.5N$)	corr(as)	$\#_{cl}(bs)$	$\#_{cl}(as)$	$\#_{cl}(opt)$
(5,2)	250	.2	0.98	**0.88**	**0.63**	0.97	171	50	45
(5,3)	354	.2	0.97	**0.89**	**0.78**	0.95	207	53	60
(5,4)	141	.2	0.92	**0.88**	**0.85**	0.91	87	35	40
(5,5)	24	.3	0.97	0.97	0.97	0.97	17	10	12
(6,1)	91	.4	0.98	0.92	**0.5**	0.97	67	25	19
(6,2)	384	.2	0.93	**0.89**	**0.29**	**0.88**	268	62	66
(6,3)	962	.2	0.97	0.93	**0.8**	0.97	589	139	115
(6,4)	741	.2	0.95	**0.85**	**0.8**	0.94	461	92	110
(6,5)	174	.2	0.92	**0.87**	**0.87**	**0.89**	127	26	57
(6,6)	27	.4	0.98	0.98	0.98	0.98	14	4	14
(7,1)	121	.5	0.97	0.92	**0.81**	0.98	87	26	22
(7,2)	616	.2	0.95	**0.34**	**0.22**	**0.88**	422	123	91
(7,3)	2030	.2	0.97	0.93	**0.81**	0.93	1332	191	196
(7,4)	2800	.2	0.96	0.94	**0.84**	0.96	1827	261	245
(7,5)	1258	.2	0.91	**0.87**	**0.84**	**0.89**	885	126	182
(7,6)	228	.2	0.94	0.93	0.93	0.94	164	33	77
(7,7)	31	.5	0.99	0.99	0.99	0.99	18	2	16
(8,1)	159	.6	0.98	0.96	**0.84**	0.98	122	45	25
(8,2)	945	.3	0.95	**0.89**	**0.26**	0.93	688	187	120
(8,3)	3493	.2	0.96	0.91	**0.82**	0.95	2431	382	308
(8,4)	7969	.2	0.97	0.95	**0.88**	0.94	5224	700	476
(8,5)	6426	.2	0.95	0.92	**0.86**	0.94	4332	528	462
(8,6)	1800	.2	0.91	0.9	**0.82**	**0.89**	1299	172	280
(8,7)	297	.3	0.96	0.96	0.95	0.96	219	36	100
(8,8)	36	.6	1.0	1.0	1.0	0.98	29	17	18
(9,1)	208	.7	0.99	0.96	0.91	1.0	162	61	28
(9,2)	1391	.4	0.96	0.92	**0.53**	0.95	1090	297	153
(9,3)	5942	.2	0.96	0.91	**0.25**	0.95	4386	792	456
(9,4)	17888	.2	0.97	0.95	**0.89**	0.95	12835	1487	840
(9,5)	24176	.2	0.95	0.94	0.9	0.91	17452	1508	1008

(L,i)	N	p	corr(N)	corr($.75N$)	corr($.5N$)	corr(as)	$\#_{cl}(bs)$	$\#_{cl}(as)$	$\#_{cl}(opt)$
(9,6)	12039	.2	0.91	0.9	**0.87**	**0.88**	8862	881	798
(9,7)	2600	.2	0.93	0.93	**0.86**	0.93	1921	283	408
(9,8)	378	.4	0.98	0.98	0.94	0.98	292	41	126
(9,9)	43	.7	1.0	1.0	1.0	1.0	31	2	20
(10,1)	274	.7	0.99	0.99	0.91	1.0	229	60	31
(10,2)	2003	.4	0.96	0.93	**0.4**	0.96	1607	397	190
(10,3)	9927	.2	0.96	0.93	**0.25**	0.95	7686	1301	645
(10,4)	33037	.2	0.95	0.93	**0.88**	**0.89**	25274	2709	1380
(10,5)	70198	.2	0.93	0.9	**0.88**	**0.83**	52912	3573	1974
(10,6)	58653	.2	0.92	0.9	**0.87**	**0.85**	44527	2664	1932
(10,7)	19036	.2	0.91	0.9	**0.89**	0.9	14448	1157	1290
(10,8)	3782	.2	0.95	0.94	**0.88**	0.95	2879	217	570
(10,9)	480	.4	0.99	0.99	0.95	0.99	348	34	155
(10,10)	51	.7	1.0	1.0	1.0	1.0	37	2	22

Parameter Settings

In the implementation we used [21], the XCS parameters can be set with a configuration file. The parameters *maxPopSize* and *dontCareProb* are set for each experiment individually (see N and p in the previous subsection), the other parameters are set as follows:

```
nrExps 1
maxNrSteps 250000
testFrequency 500
maxPopSize  9927
initializePopulation 0
alpha 0.1
beta 0.2
gamma 0.8
epsilon0 0.01
nu 5

thetaGA 250
fitnessReduction 0.5
tournamentSize 0.4
selectTolerance 0.001
crossoverType 0
forceDifferentInTournament 0
chiGA 0.5
muGA 0.1
doGeneralizationMutation 0
```

```
doNicheMutation 1

doMAM 1
doGAErrorBasedSelect 0

delta 0.1
thetaDel 20
deletionType 1
dontCareProb 0.20
doGASubsumption 0
doActionSetSubsumption 0
thetaSub 200
exploreProb 0.2
```