XCS Cannot Learn All Boolean Functions

Charalambos Ioannides
Industrial Doctorate Centre in Systems
University of Bristol
Queen's Building, University Walk
Bristol BS8 1TR, UK
+44 (0)117 331 5421
charalambos.ioannides@bristol.ac.uk

Geoff Barrett
Broadcom BBE BU, Broadcom Corporation
220 Bristol Business Park,
Coldharbour Lane
Bristol BS16 1FJ, UK
+44 (0)117 906 2750
gbarrett@broadcom.com

Kerstin Eder
Department of Computer Science,
University of Bristol
MVB, Woodland Road
Bristol BS8 1UB, UK
+44 (0)117 954 5146
kerstin.eder@bristol.ac.uk

ABSTRACT
In this paper we applied the eXtended Classifier System (XCS) on a novel real world problem, namely digital Design Verification (DV). We witnessed the inadequacy of XCS on binary problems that contain high overlap between optimal rules especially when the focus is on population and not system level performance. The literature attempts to underplay the importance of the aforementioned weakness and in short, supports that a) XCS can potentially learn any Boolean function given enough resources are allocated (right parameters used) and b) the main metric deciding the learning difficulty of a Boolean function is the amount of classifiers required to represent it (i.e. ‖[O]‖). With this work we experimentally refuted the aforementioned propositions and as a result of the work, we introduce new insights on the behavior of XCS when solving two-valued Boolean functions using a binary reward scheme (1000/0). We also introduce a new population metric (%[EPI]) that should necessarily be used to guide future research on improving XCS performance on the aforementioned problems.

Categories and Subject Descriptors
I.2.6 [**Artificial Intelligence**]: Learning – *knowledge acquisition*

General Terms
Algorithms, Performance, Design, Experimentation

Keywords
Learning Classifier Systems (LCS), XCS, Logic Optimization

1. INTRODUCTION
Machine Learning (ML) techniques aim to solve real-world problems and constitute an essential tool for understanding, learning, and inferring knowledge from the deluge of information readily available in today's scientific, industrial, and societal systems. These intelligent techniques are formed and improved either through scientific exploration or through industrial application and experimentation. This paper describes how the adoption of Learning Classifier Systems (LCS) and, more specifically, accuracy-based fitness LCS (a.k.a. XCS), in the digital design verification (DV) field has exposed a weakness, which instigated experimentation that intended to draw a more complete picture of the true properties of these systems. It is hoped that future improvements in XCS will follow based on the work presented here.

In digital design verification, engineers are interested in discovering as many bugs related to a digital design as possible before it is manufactured. Two prominent methodologies are followed: The first uses formal verification to prove formal properties of designs. The second uses simulation-based verification to produce tests that exercise the functionality of the designs to achieve a sufficient degree of coverage; this methodology is relevant to the study presented in this paper.

In its lifecycle, a design can be viewed at differing levels of abstraction. Relevant to this work (i.e., simulation-based verification) is the code or register transfer level (RTL), as realized through the use of hardware description languages such as Verilog or VHDL. To test such a design, a testbench is constructed that allows test code to be sent to the design and feedback from the process to be stored for later processing. In assessing the completeness of the verification process, various metrics are used. These include code coverage metrics, such as branch, expression, and toggle coverage, along with custom declared metrics, such as functional coverage [1].

Coverage Directed Generation (CDG) of tests is a technique that aims to automate obtaining full coverage of digital designs through the use of ML methods. The assumption underlying CDG is that the learning mechanism can identify, from existing tests and coverage data, how best to bias stimulus generation such that the resulting new tests when run on the design under verification (DUV) can reach outstanding coverage. This is not as straight-forward as it seems because designs nowadays are significantly complex systems, requiring equally elaborate tests in order for 100% of their structure and functionality to be exercised. Furthermore, due to delays of variable time length between an input and a potential output ports, any cause and effect relationships are difficult to spot and learn.

A very important aspect in CDG is achieving 100% coverage as soon as possible and with as few resources as possible. A key in achieving the latter is to find a way to balance coverage achieved over all cover points as dictated by the chosen coverage model. In fulfilling this requirement, it is necessary to form a complete mapping between the test input biases and the coverage they achieve. The extent to which coverage can be balanced between the cover points in the coverage model is very much dependent on the completeness and accuracy of the mapping learnt.

The XCS variant [2], has been shown to develop rules forming a complete, accurate, minimal, and non-overlapping mapping between the input and output space of Boolean functions [3]. The aforementioned four properties of solutions as they are discovered by XCS have made them a very attractive ML technique to try out in solving the aforementioned DV problem. The only prerequisite is the representation of the problem using a ternary alphabet (i.e., {0, 1, #}) while treating it as a single-step problem.

Although the original intent of this work was to use XCS in solving the DV and coverage balancing problem, its inability to do so for a very simplified example has led to further investigation to identify the underlying causes. This investigation has allowed for a better understanding of the true properties of XCS and its abilities when utilizing a ternary representation and binary reward scheme in solving Boolean functions.

The structure of the paper is as follows: In Section 2, a background on XCS and Boolean function minimization is presented. Section 3 describes the DV problem as formulated for solution via XCS. Section 4 includes the initial experiments performed and the findings that required further investigation. Section 5 presents the new insights gained and the hypotheses formed on suboptimal XCS behavior. Section 6 provides evidence for the claims made in this paper, and Sections 7 and 8 summarize findings and conclude this paper by providing an overview of the work and future research topics.

2. BACKGROUND

To follow the material and ideas discussed in this paper, some background in LCS, and Boolean Function representation and minimization is required.

2.1 XCS

The eXtended Classifier System (XCS), as first introduced in [2], is a very popular LCS variant. LCS is an adaptive rule-based ML technique that usually combines Reinforcement Learning (RL) and Genetic Algorithms (GA) to derive a set of production (IF condition THEN action) rules (or *classifiers*) that form the solution to presented problems. Problems are categorized as single-step, i.e., requiring one action for their solution, or multi-step, requiring a series of jointly optimal actions.

XCS is popular because it uses a novel classifier fitness function that is based on the accuracy of the prediction of the classifiers in the evolved populations (accuracy-based fitness), rather than on the reward prediction (strength-based fitness). This type of classifier fitness function is one of the mechanisms that apply pressure towards complete, accurate, minimal, and non-overlapping populations of classifiers [4], [5]. A very good overview of the evolution of classifier systems is found in [6].

Many classification studies with XCS have dealt with Boolean functions, for which classifiers typically use a ternary representation of $\{0, 1, \#\}^n$ for the n-bit Condition part and the binary strings $\{0, 1\}^m$ for the m-bit Action part. The # symbol represents generalization over ‘0’ and ‘1’ values. A very popular Boolean function, which helped in evolving and evaluating the performance of LCS and XCS implementations, is the *multiplexer problem* described in the following subsection.

2.1.1 Multiplexer Problem

Using the operation of a well-known digital design component, this problem is related to finding which bit in an input bit string is to be selected and propagated to the output of the component. The input bit string received from the ‘environment’ of XCS comprises the address and data parts as they would be inputted to a multiplexer component. XCS must learn the correct data bit to be sent to the output given a particular input. The formula that gives the length L of a problem is $L = k + 2^k$, where k is the number of the address bits. Figure 1 illustrates the 6-MUX and 11-MUX problems, with the 11-MUX depicted on the left.

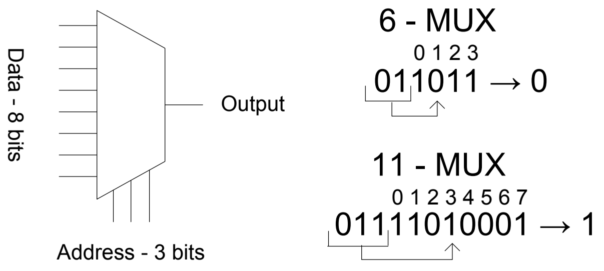


Figure 1. The Multiplexer problem (11-MUX)

These types of problems have been popular in the LCS community due to their properties of multi-modality (more than one solution can be assigned maximum reward), scalability (the size of the problem can grow with the address bits), and epistasis (the importance of a bit in solving the problem is affected by other bits in the input string).

2.2 Boolean Functions and Minimization

A Boolean function is a mapping between two Boolean spaces. If n bits define the input and m bits define the output spaces, then the mapping is formally noted as $\mathbf{f} : B^n \rightarrow B^m$. When $m = 1$, Boolean functions are called *single-output*; otherwise, when $m > 1$, they are called *multiple-output*. An input variable or its complement is called a *literal*. A product term in which no variable appears more than once is called a *normal product term*. Boolean functions can be expressed as either the sum of products of n literals, called *minterms*, or the product of sums of n literals, called *maxterms*. An alternative definition is that every value assignment of the n-bit input literals that corresponds to a ‘1’ in the output of single-output functions is a minterm, while those with an output of ‘0’ are the maxterms of the function.

2.2.1 Boolean Function Representation

Boolean functions can be represented in tabular form, as logic expressions or as binary decision diagrams. The simplest of tabular forms is the two-column truth table. The first column contains an ordered listing of all possible Boolean input vectors, and the second column contains the corresponding output vectors.

Logic expressions are formed by conjunction (· or AND), disjunction (+ or OR) and negation (’ or NOT) Boolean operators, when applied to the n input literals of a Boolean function. As previously mentioned, when the expression is in terms of a sum of products (SOP), then these products are called minterms, and when it is in terms of a product of sums (POS), these sums are called maxterms.

Binary decision diagrams (BDD) are an alternative way to represent Boolean functions. They are undirected graphs, with each vertex being an input variable, and thus a decision point, in predicting the output of the function (e.g., either ‘0’ or ‘1’ in single-output functions).

Another, more compact way to represent Boolean functions is the Sigma notation. If each of the rows in a truth table is indexed and grouped into minterms (onset) and maxterms (offset), the Sigma notation defines the function by listing its onset indices. For example, the function F = A·B + A·B’ can be represented in Sigma notation as Σ(2,3). Figure 2 depicts the 3-MUX problem in all the aforementioned forms of representation.

Truth Table

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Logic Expression (SOP)

F = (A’·B·C’) + (A’·B·C) + (A·B’·C) + (A·B·C)

Minimized Logic Expression (SOP)

F = (A’·B) + (A·C)

Sigma Notation

Σ(2,3,5,7)

BDD

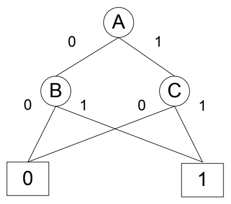


Figure 2. Representations of 3-MUX function

2.2.2 *Boolean Function Minimization*

The primary use of Boolean function optimization is for logic circuit design and synthesis. In that context, Boolean functions are converted to circuits. Minimizing logic gate count and the connections between gates and input signals (fan-in) reduces the silicon area (hence cost) and improves signal propagation delays.

The discussion of optimization here is limited to single output functions expressed as SOPs. The following are relevant definitions: A *minimal sum* is a SOP expression such that no other expression contains fewer product terms. An *implicant* is a normal product term that implies the function F; e.g., F = A·B’ + A·B’·C + A·C contains three implicants. A *prime implicant* (PI) is an implicant of F, such that if a variable is removed from the implicant, it no longer implies F; in other words, it is a maximally general term that implies F. In the previous example, only the implicants A·B’ and A·C are prime. An *essential prime implicant* (EPI) is one which implies one or more minterms implied by no other prime implicant. A *minimal cover* of a function is the set of prime implicants from which any further removal does not allow coverage of the function. The set with the aforementioned properties is called a *minimal prime implicant* (MPI) set. Those necessarily include all essential prime implicants plus any other prime implicants that cover the remaining minterms of the function, see Figure 3. The aforementioned types of prime implicants form sets, now on being denoted as [PI], [EPI], [MPI].

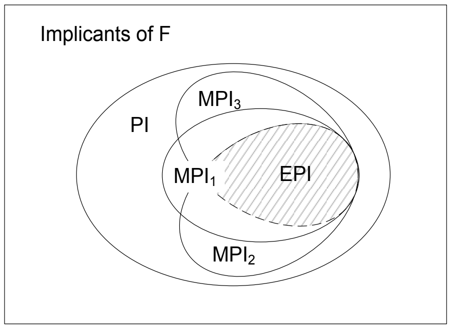


Figure 3. Venn diagram of Implicant sets

A Boolean function can have one [PI], one [EPI] but potentially many [MPI] sets. This is because some minterms can be covered by more than one prime implicants and therefore the number of total [MPI] per function is relative to the different combinations of prime implicants covering those minterms.

All [MPI] of a given function are of equal cardinality. It is also possible to have a function with the property [MPI]=[EPI], in that case the cover is said to be the *minimum cover* of the function.

3. PROBLEM DESCRIPTION

Following investigations on the properties of XCS, e.g., [5] and [7], especially for single-step problems with ternary representation, we decided to use XCS to learn the relationship between biases of a test generator and the coverage thereby achieved, after describing the problem in Boolean function form. To the best of our knowledge this was the first time any type of LCS had been used in solving DV problems.

The expectations, therefore, were that XCS would manage to learn a complete, accurate, minimal, and potentially non-overlapping mapping between biases and the coverage they achieve. The ultimate goal was to rely on the learnt population to guide effective test generation so that balanced and efficient coverage is achieved.

3.1 Design Under Verification

The DUV chosen for this work was a digital signal processor called FirePath, at the accumulator stage in the long pipeline of its design. Originally, we decided that the Bias bits would represent the presence or absence of specific types of opcodes in the tests simulated on the DUV. The number of such bits was decided to be 21, enough to represent all types of opcodes.

The coverage model chosen was the toggle coverage of all control signals in the accumulator stage of the long pipe. Toggle coverage refers to whether a signal’s value has been changed from zero to one and back to zero, i.e., a full period, during simulation. The control signals of interest required 42 bits to be represented.

3.2 Proof of Concept Setup

Before attempting to solve the entire problem as described above, we decided to first test XCS abilities in a much smaller example, to test the validity of our expectations. The smaller example grouped bias controls in a more compact way, thus requiring only 7 bits to represent the conditions of classifiers. Accordingly, and trying to mimic a traditional single-output Boolean function, only one control signal was chosen to form the coverage model and, therefore, the Action part of classifiers.

4. METHODOLOGY

In the experimental setup, XCS alternates between explore and exploit trials. During explore trials the classifiers update their statistics while during exploit trials the system performance is recorded. XCS receives random bias strings from the ‘environment’ (i.e., a pseudorandom bit string generator) and a test is generated accordingly. Next, the test is simulated on the DUV. Depending on the coverage results achieved, XCS is expected to learn which bias strings (conditions) toggle the chosen signal (action). The test generator used in our setup is called FireDrill, and it constructed tests containing 100 instructions. If tests predict correctly whether they toggle the signal or not, they are assigned a reward of 1000, if not they are assigned 0.

The experimental loop depicted in Figure 4 was used in the beginning of the experiments. However, it had to be emulated to eliminate the repeated computationally expensive calls to the simulator. Learning epochs usually allocated for XCS systems range from a few thousands to hundreds of thousands; therefore, given that each call to the simulator required ~2 minutes of runtime, we decided to first simulate all possible tests ($2^7 = 128$ tests), store their coverage results, and then allow XCS to treat them as a lookup table. This way, experimental runs were sped up by three orders of magnitude.

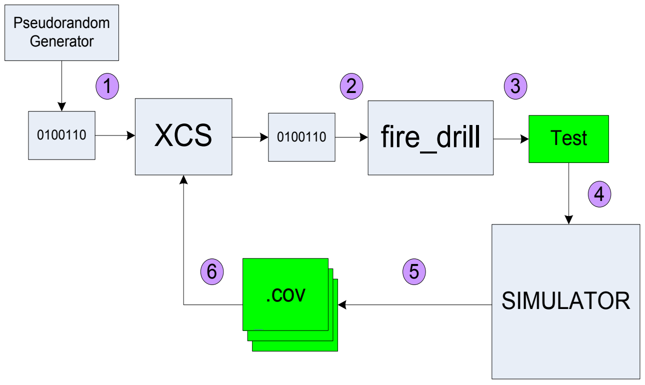


Figure 4. The main experimental setup

The results of the aforementioned exhaustive simulation formed a Boolean function henceforth named DV1, with the following sigma notation: Σ(1, 2, 3, 8, 9, 10, 11, 13, 14, 24, 25, 26, 27, 28, 30, 40, 41, 42, 43, 46, 47, 56, 57, 58, 59, 61, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 88, 89, 90, 91, 93, 94, 95, 97, 98, 99, 101, 102, 103, 104, 105, 106, 107, 109, 110, 113, 114, 115, 117, 118, 121, 122, 123, 125, 126, 127).

The XCS system used is the XCSlib v.1.1 developed at the Politecnico di Milano by Prof. Pier Luca Lanzi et al. [8]. The parameter settings used for all experiments in this work, unless otherwise stated, were $N = 1000$, $P_\# = 0.3$, $\alpha = 0.1$, $\beta = 0.2$, $\chi = 0.8$, $\mu = 0.04$, $\nu = 5$, $\varepsilon_0 = 10$, $p_I = 10$, $\varepsilon_I = 0$, $f_I = 0.01$, $\theta_{GA} = 25$, $\theta_{del} = 20$, $\theta_{GAsub} = 20$, $\theta_{ASsub} = 100$, 1-point x-over = on, AS subsumption = on, GA subsumption = on, GA tournament selection = off.

The statistics reported are the system performance as a percentage of correct predictions, the system error as the difference between predicted and actually achieved reward by XCS and finally the population which is the percentage of N (population size). The three main metrics, as shown on graphs and for each exploit problem, denote the moving average of the last 100 exploit trials.

4.1 Initial Results

Judging from the literature [5], [7], [9], the DV1 function would be sufficiently small ($3^7 \times 2 = 4374$ possible unique rules and only $2^7 = 128$ input examples) to be fully learnt within 2×10^6 exploit trials, i.e. achieve 100% performance and almost zero error.

However, as seen in Figure 5, the system performance fluctuated sub-optimally with the system error being relatively high, never settling throughout the experiment.

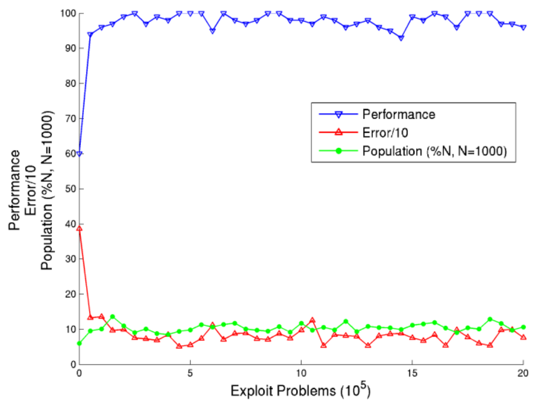
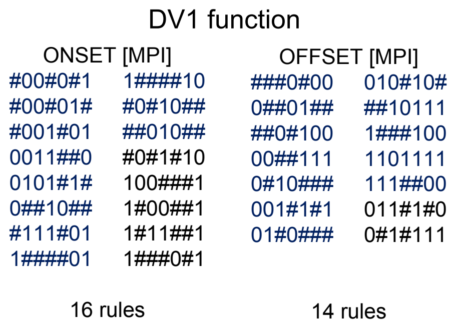


Figure 5. XCS performance on DV1 function[1]

4.2 Further Investigation

To investigate the reasons behind the suboptimal behavior, it was necessary to learn which rules should exist in the final population to achieve optimal performance. To this end, we decided to use a technique from the Logic Optimization literature, the Quine-McCluskey (Q-M) algorithm [10]. This is an *exact* Boolean function optimization algorithm which means that it calculates all prime implicants (i.e. the [PI] set), all essential prime implicants (i.e. the [EPI] set) and one of the possible minimal prime implicant sets (i.e. an [MPI] set) of a Boolean function. When applying this technique to the onset as well as the offset of the DV1 function and combining the two resulting MPI sets, it is possible to discover the minimal set of rules that fully and accurately cover it.



DV1 function

ONSET [MPI]		OFFSET [MPI]	
#00#0#1	1####10	###0#00	010#10#
#00#01#	#0#10##	0##01##	##10111
#001#01	##010##	##0#100	1###100
0011##0	#0#1#10	00##111	1101111
0101#1#	100###1	0#10###	111##00
0##10##	1#00##1	001#1#1	011#1#0
#111#01	1#11##1	01#0###	0#1#111
1####01	1###0#1		
16 rules		14 rules	

Figure 6. DV1 MPI set

It was believed that this [MPI], see Figure 6, would be equivalent to what was defined and termed in [9] as [O] (complete, accurate, minimal, and non-overlapping population of rules). Therefore, it was expected that the population would need to converge to this [MPI], if XCS was expected to fully solve/learn it. In blue are all the essential prime implicants of the function, while in black, the list of the prime implicants that cover the remaining minterms of the DV1 function, thus forming an [MPI], as explained in Section 2.2.2.

To test whether XCS would fundamentally end up with this 'optimal' population, we seeded the population with these rules at the beginning of the experiment and observed whether the rules would survive in it, having the GA and Deletion mechanisms on.

[1] All graphs in the paper are in color for better readability

The results of this experiment, as discussed in more detail in Section 6, were contrary to expectations. Only a few of these rules survived, while a limited set of those, in the course of the 2×10^6 learning epochs, were deleted and rediscovered. In addition, many of these rules, although containing correct Reward and Error statistics, they had Fitness that was far from optimal. Their suboptimal fitness would be a plausible reason for their deletion. This is because it would increase the probability of deletion and reduce reproduction opportunities of the corresponding rules, as identified and investigated in [5] and [11].

The findings on the DV1 problem call for more investigation on the claims made in [7], [9], and [12], as XCS was clearly unable to retain the rules that optimally cover the DV1 function.

5. NEW INSIGHTS

Having a closer look at the prime implicants in the [MPI] of the DV1 function, we discovered that most of these overlapped, which explained the suboptimal Fitness statistics and subsequent system performance. The question then was, "Can these overlaps be avoided?" or "Is there a set of optimal and non-overlapping classifiers for this problem?"

Looking into these questions using a free online logic minimization tool [13] and testing various Boolean functions of the same or smaller size of DV1, we discovered that overlap of essential prime implicants in [MPI] of functions was unavoidable and constituted the norm in the cases investigated. These findings are due to the properties of the Boolean function representation chosen and therefore the generalization properties of disjunctive or conjunctive normal forms (DNF or CNF).

Previous work [12] has suggested the difficulty of learning a Boolean function is related to its dimensionality, the mean hamming distance (MHD) of classifiers in [MPI], and the size of [MPI]. We now propose that overlaps between high-fitness rules, especially when members of [EPI], are a major factor. Another implication of these results is that XCS performs logic minimization from two ends at once, discovering both prime implicants as well as *prime implicates* (i.e. maximally general rules implying the maxterms of a function) when using the ternary representation {0,1,#} and a binary reward scheme (1000/0).

Additionally, using [MPI] to denote the population to be learnt is better than [O]. The latter contains the condition of disjunctiveness of classifier conditions, something that, as previously stated, is not always possible. In fact, the vast majority of Boolean functions of any arbitrary length contain overlaps in their ternary (or DNF) rule representations, thus making the use of [O] as a population metric non-realistic.

We analyzed the multiplexer function, which was heavily used as a test in the development and analysis of XCS, and the parity function, that was previously claimed to be the most difficult Boolean function of a given length for XCS to learn [12]. Both belong to the class of functions that have [MPI]=[EPI] and do not contain overlapping prime implicants (or classifiers) in [MPI]. We propose that this class of functions is easier for XCS to learn than those which contain overlaps in [MPI]. Additionally, tests using Q-M have pointed out that the functions with these properties, given the set of all possible n-bit functions, are vastly fewer in comparison to those containing overlaps. To begin realigning the view in the literature on what XCS can and cannot learn with respect to the above findings, we attempt to succinctly summarize the literature's views and then experimentally refute them. These views on XCS's capabilities on single-step Boolean functions have been investigated in [7], [9], and [12] and are summarized in the following two propositions:

1. XCS can potentially learn any Boolean function if enough resources are allocated (the right parameters are used).
2. The main metric in determining the difficulty of learning a Boolean function via XCS is the amount of classifiers required to represent the function (i.e., |[O]|).

The following two experiments have been devised to refute the aforementioned and corresponding propositions.

Experiment 1:

One can experimentally show that XCS will never completely solve a binary problem with overlapping EPIs (i.e., achieve 100% system and population performance), if:

a. Seeding XCS with [MPI], with GA discovery off, the rules obtain the correct reward error and fitness values with no more rules being introduced via the covering mechanism [2].

b. Seeding the starting population with [MPI], with GA discovery on, some rules are eventually deleted. They may be rediscovered and deleted but in general the cardinality of [MPI] remains below the optimum.

Experiment 2:

Choose a 6-bit Boolean function that, although requires fewer prime implicants to form a complete cover than the 6-MUX problem, the implicants happen to overlap. One such generic problem is Σ(8,9,10,11,16,17,18,19,24,25,26,27), henceforth referred to as GEN. The idea is to perform an experiment similar to Experiment 1 on GEN and MUX, and show that, with GA discovery on, rules are deleted and rediscovered, never attaining maximum %[MPI][2].

GEN function		6-MUX function	
ONSET [MPI]	OFFSET [MPI]	ONSET [MPI]	OFFSET [MPI]
0#10##	#00###	001###	000###
01#0##	###1##	01#1##	01#0##
	1#####	10##1#	10##0#
		11###1	11###0
2 rules	3 rules	4 rules	4 rules

Figure 7. GEN and 6-MUX MPI set

6. EVALUATION

In this section, the results of the two aforementioned experiments are presented. In addition to the three classical performance metrics, the three population performance metrics, namely %[PI], %[EPI] and %[MPI] are also included. These are not averaged over the last 100 exploit trials but correspond to the percentages recorded at each exploit trial. All results are averaged over 10 experiment runs unless otherwise stated.

[2] The %[X] population performance metric ([X] being some arbitrary set of classifiers), has been defined in [11] as the proportion of rules in [X] existing in [P].

6.1 Experiment 1 – Results

The first part of this experiment is to confirm that the [MPI] constitutes a maximal, accurate and complete coverage of the function. If this was not the case, then, even with GA off, the covering mechanism would introduce new classifiers.

The classifiers forming the [MPI] set of the DV1 function are 60. The onset of the function is covered by 16 members of [MPI], while the offset is covered by another 14. The final number was obtained by doubling the aforementioned sum as both actions in XCS must be represented for each rule [2].

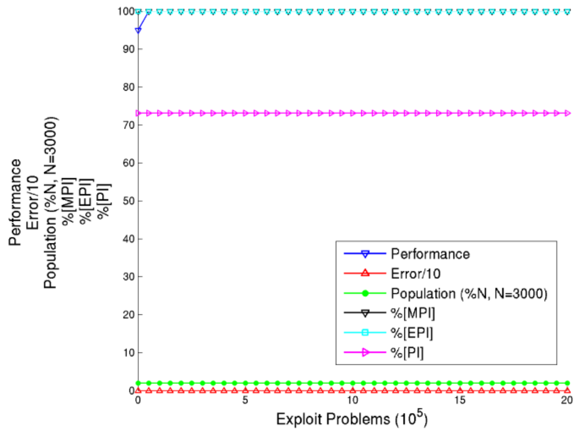


Figure 8. Results of Experiment 1a on DV1 problem

After performing this first part of the experiment on 2×10^6 exploit cycles with N=3000 and the rest of the parameters as stated in Section 4, no new classifiers were introduced in the population [P]. Furthermore, all classifiers attained the correct statistics on Reward (R) and Error (E), see Figure 8, but not on the Fitness (F) metric due to overlaps in [EPI] and therefore in [MPI].

Table 1. Sample of classifiers in [MPI] of DV1 with GA off

C : A	R	E	F	AS	EXP	NUM
1101111 : 0	1000	0.00	1.00	1.00	216	1
1101111 : 1	0	0.00	1.00	1.00	82	1
1####10 : 1	1000	0.00	0.88	1.25	3825	1
1####10 : 0	0	0.00	0.82	1.57	1241	1
##10111 : 1	0	0.00	0.78	2.27	272	1
0101#1# : 0	0	0.00	0.69	1.93	330	1
0011##0 : 0	0	0.00	0.68	1.76	316	1
00##111 : 0	1000	0.00	0.67	2.33	972	1
01#0### : 0	1000	0.00	0.66	2.00	3716	1
011#1#0 : 1	0	0.00	0.65	2.46	302	1

Table 1 contains a sample of the [MPI] population sorted in descending fitness order. As can be seen, only the maximally specific rules obtain full fitness. The results of the second part of the experiment are illustrated in Figure 9. The only difference with the previous experiment is that GA discovery is on. The purpose of this experiment is to show that XCSs are incapable of sustaining the [MPI], which forces system performance to behave sub-optimally. Also, the deletion and rediscovery mechanism is evident from a sample of rules in Table 2.

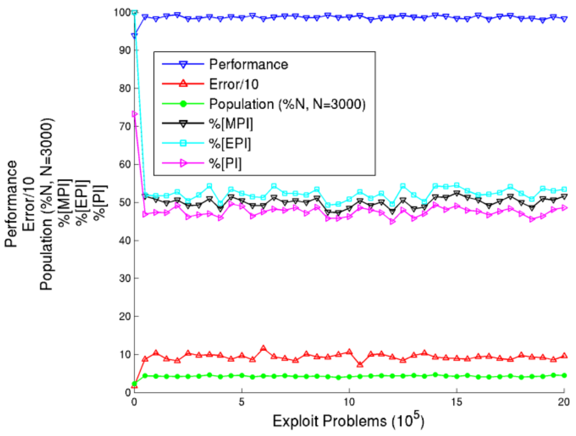


Figure 9. Results of Experiment 1b on DV1 problem

Commenting on Figure 9, the system performance, although not poor, never attains maximum value. On the other hand, there is a sharp decrease of the three population performance metrics which remain ~50% throughout the 2×10^6 explore/exploit trials. This is due to two reasons, the second being the consequence of the first: Initially fitness declines on overlapping EPI rules (fitness sharing). Consequently, these rules are less likely to be picked as parents during GA application.

The system performance is not as affected, however, since suboptimally-general classifiers in the population compensate for the rules missing from [MPI]. For some problems, this behavior is acceptable, but for the design verification field, it is very important to learn the full and accurate mapping between biases and coverage points.

Table 2. Sample of classifiers in [MPI] of DV1 with GA on

C : A	R	E	F	AS	EXP	NUM
###0#00 : 1	*0*	*0*	*0.67*	*450.67*	*124994*	*169*
1#0#### : 1	799	205	0.62	183.42	296	5
111##10 : 1	1000	0	0.62	145.14	46	2
11##110 : 1	1000	0	0.62	151.69	73	1
0#10### : 1	***0***	***0***	***0.61***	***428.02***	***25290***	***130***
1##0#10 : 1	1000	0	0.59	153.15	53	1
#00#01# : 0	*0*	*0*	*0.55*	*302.22*	*64546*	*53*
01#0### : 1	***0***	***0***	***0.53***	***464.49***	***25405***	***132***
011#11# : 1	0	0	0.51	391.69	26758	32
#111#01 : 0	*0*	*0*	*0.47*	*304.48*	*31327*	*57*

In Table 2 the rules that are part of [MPI] are in italics, while the deleted and rediscovered classifiers of [MPI] are in bold. Clear differences can be seen in the Experience (EXP) and Numerosity (NUM) stats between the classifiers that were rediscovered and those that survived. Out of the 60 classifiers in [MPI], only 33 exist in the final population, with 20 of those rediscovered in recent trials.

Another point to notice is that some of the non-[MPI] rules have higher fitness than those in [MPI]. As ranking of fitness is one indicator distinguishing optimal from non optimal rules [7], especially if the former are unknown, the aforementioned behavior makes it harder to see how changing parameters in XCS would allow the correct ranking to be achieved and hence problem solution to be potentially reached.

6.2 Experiment 2 – Results

The purpose of the experiments here was to show that the difficulty of learning a binary problem with XCS is not primarily related to the cardinality of the optimal rule set to be learnt, but to the amount of overlap between those rules.

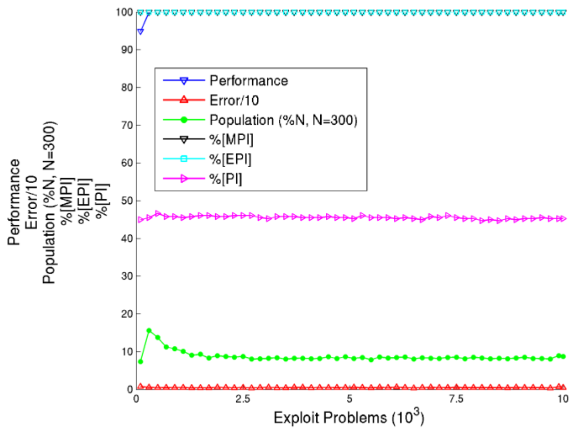


Figure 10. Results of Experiment 2 on 6-MUX problem

It is argued in [12] that the Parity problem would be most difficult in the case where |[O]| was the primary problem difficulty metric (i.e., 2^l classifiers in [O], l being the length of the input bit string). The differing settings for this experiment were N = 300 and learning problems = 10,000. Figure 10 provides results for the 6-MUX problem. After seeding the initial population with [MPI], system performance reaches 100% quickly but not immediately (due to time taken to assign correct statistics to classifiers), while %[MPI] is always at maximum.

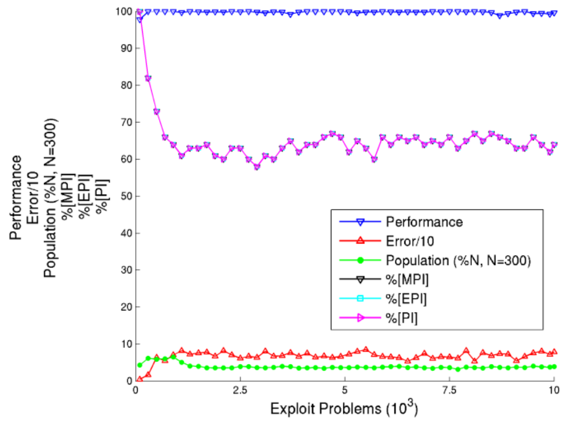


Figure 11. Results of Experiment 2 on GEN problem

The results are different with the GEN problem, as seen in Figure 11. Though |[MPI]| = |[PI]| = 10, the system performance is unable to stabilize at 100%, while %[MPI] declines to ~60%. As a result, the error is high enough to indicate incomplete learning after 10,000 exploit trials, even with $|[MPI]|_{GEN} < |[MPI]|_{6MUX}$.

7. SUMMARY

It has been shown experimentally that XCS cannot solve all single-output and in consequence *n*-output Boolean functions, regardless of the resources allowed. This is due to XCS's fitness regime, which only allocates appropriate fitness when the solution consists of disjunctive rules. But more fundamentally, the reason behind the problem witnessed is the inevitability of overlap between maximally general and accurate rules (i.e. prime implicants), when using a ternary generalization representation.

In addition, it has been shown that the population metric %[O] is not a realistic measure on the majority of Boolean functions as it implies not only completeness, accuracy and minimality but also disjunctiveness of rules. Instead, using terms known in the Boolean function optimization literature, e.g. sets such as [MPI], [EPI] and [PI] along with their percentage metrics, would be more appropriate as their definitions map one-to-one to the ternary generalization representation of Boolean functions that XCS uses.

In [11], %[m-DNF] (i.e. %[MPI]) and %[PI] are first introduced, but the %[EPI] metric is absent. This metric is a necessary measure towards obtaining any of the potentially many minimal representations a Boolean function can have, as any of those will need to include all members of [EPI]. A potential overlap of rules in [EPI] is expected to render the problem particularly difficult for XCS to solve.

The problems with fitness sharing have also been identified in [14]. There, although Boolean functions are separated in three classes (i.e., requiring only overlapping, only non-overlapping and at least one non-overlapping solution), the authors seem confident that there can be a potential setup of XCS from which optimal behavior can be obtained, even if there are overlaps in the [O] solution. In contrast, the results presented in this paper clearly demonstrate that given a problem's population solution is more important than the system performance and, given an overlap of the EPI, it is impossible for XCS in its current form to provide a stable solution.

8. CONCLUSIONS

When XCS is used on single-step Boolean function problems that use ternary conditions and a binary reward scheme (1000/0), then its operation is that of a logic optimizer. In this case, each optimal classifier discovered is effectively a PI of the function, with a good chance of belonging to the [EPI] and [MPI] as well. XCS tries to find minimal, accurate and complete equivalents of the function presented to it (in parallel both on the onset and the offset, something which does not happen with deterministic algorithms such as the Q-M method).

What does not help its operation is the fitness function traditionally used, which shares the overall fitness of a niche to its members [2]. Though this would not be a problem if all Boolean functions could be minimally represented by maximally general classifiers without overlaps in their conditions, this is not the case in reality. Although a Boolean function can potentially have more than one [MPI], it can only have one [EPI], and when essential implicant overlap is inevitable, then XCS performance suffers.

As a natural consequence of the above, the Parity function cannot be considered the hardest binary problem to be presented to XCS. Its [MPI] members, although as numerous as possible, are non-overlapping; therefore, a complete solution would be expected, depending on the parameters chosen. In fact, the Parity problem is the hardest of the class of easy binary problems (i.e. problems with disjoint solutions), as it is time-consuming for XCS to discover many and fully specific rules (due to its natural generalization pressure). This motivates us to discover classes of Boolean functions, rather than specific cases, on which XCS exhibits similar behavior.

Requiring XCS to be good at finding minimal and thus potentially non-overlapping (which is not always possible) solutions is making the system behave sub-optimally in terms of system level performance. Hence there is a trade-off issue here. It has to do with the restrictions imposed on XCS' performance by the way the algorithm has been structured. So, as aforementioned, requiring disjunctiveness as well as completeness, minimality and accuracy, is making the system behave non-optimally, as disjunctiveness of rules is a very rare property of Boolean functions.

For future work, judging from more recent attempts to analyze the effect of overlapping classifiers in the final solution representation [15] and work related to classifying XCS as a probably approximately correct (PAC) learner for k-DNF functions given non-severe solution overlap [16], these could be used to investigate the findings of this paper from a different perspective. This should be related to what constitutes a hard problem for XCS and what are the most important or influential criteria for learning difficulty. It is the belief of the authors that overlap of classifiers has been so far underestimated in both its effects and implications.

Finally, the link between Boolean function classes and the difficulty of using XCS to solve them (w.r.t. the overlapping [EPI] problem) should be investigated. In light of this new population metric, future experiments that investigate the effects or severity of overlaps should be made to include it as standard practice.

Another valuable direction of future research would be to propose a new fitness function or XCS setup that allows the coexistence of overlapping EPIs. If accomplished, XCS would be made more robust, adding to its reputation of being a versatile genetics-based ML technique that can handle on-line learning for single-step and multi-step problems.

9. ACKNOWLEDGMENTS

The authors would like to thank Dr. Tim Kovacs for his invaluable support and comments towards the completion of this work. Charalambos Ioannides is an EPSRC and Broadcom Corporation funded research engineer at the IDC in Systems, University of Bristol.

10. REFERENCES

[1] A. Piziali, *Functional verification coverage measurement and analysis*. Berlin: Springer, 2007.

[2] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149-175, 1995.

[3] T. Kovacs, "What Should a Classifier System Learn?," *Evolutionary Computation*, vol. 2, pp. 775 - 782, 2001.

[4] T. Kovacs, "Strength or Accuracy? Fitness Calculation in Learning Classifier Systems," in *Learning Classifier Systems, From Foundations to Applications*, London, UK, 2000, p. 143–160.

[5] M. V. Butz, *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*, 1st ed. Springer, 2005.

[6] R. J. Urbanowicz and J. H. Moore, "Learning classifier systems: a complete introduction, review, and roadmap," *J. Artif. Evol. App.*, vol. 2009, p. 1:1–1:25, 2009.

[7] T. Kovacs, "A Comparison of Strength and Accuracy-Based Fitness in Learning Classifier Systems," PhD thesis, University of Birmingham, 2001.

[8] P. L. Lanzi and D. Loiacono, *XCSLib: The XCS Classifier System Library*. Illinois Genetic Algorithms Lab: University of Illinois, 2009.

[9] T. Kovacs, "XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions," in *Soft Computing in Engineering Design and Manufacturing*, Springer, 1997, pp. 59-68.

[10] W. V. Quine, "The Problem of Simplifying Truth Functions," *The American Mathematical Monthly*, vol. 59, no. 8, pp. 521-531, Oct. 1952.

[11] T. Kovacs, "Performance and population state metrics for rule-based learning systems," in *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, Washington, DC, USA, 2002, p. 1781–1786.

[12] T. Kovacs and M. Kerber, "What Makes a Problem Hard for XCS?," in *Revised Papers from the Third International Workshop on Advances in Learning Classifier Systems*, London, UK, 2001, p. 80–102.

[13] N. UFRGS Research Lab, "KARMA3."[Online]. Available: http://www.inf.ufrgs.br/logics/docman/karma/. [Accessed: 01-Jul-2011].

[14] M. V. Butz, D. E. Goldberg, and K. Tharakunnel, "Analysis and improvement of fitness exploitation in XCS: bounding models, tournament selection, and bilateral accuracy," *Evolutionary Computation*, vol. 11, no. 3, pp. 239-277, 2003.

[15] M. V. Butz, D. E. Goldberg, P. L. Lanzi, and K. Sastry, "Problem solution sustenance in XCS: Markov chain analysis of niche support distributions and the impact on computational complexity," *Genetic Programming and Evolvable Machines*, vol. 8, p. 5–37, Mar. 2007.

[16] M. V. Butz, D. E. Goldberg, and P. L. Lanzi, "Computational Complexity of the XCS Classifier System," in *Foundations of Learning Classifier Systems*, vol. 183, L. Bull and T. Kovacs, Eds. Springer Berlin / Heidelberg, 2005, pp. 914-914.