Introducing XCS to Coverage Directed Test Generation

Charalambos Ioannides
Industrial Doctorate Centre in Systems
University of Bristol
Bristol, UK
charalambos.ioannides@bristol.ac.uk

Geoff Barrett
Broadcom BBE BU
Broadcom Corporation
Bristol, UK
gbarrett@broadcom.com

Kerstin Eder
Department of Computer Science
University of Bristol
Bristol, UK
kerstin.eder@bristol.ac.uk

Abstract— **Coverage Directed test Generation (CDG) is rife with challenges and problems, despite the relative successes of machine learning methodologies over the years in automating it. This paper introduces the use of the eXtended Classifier System (XCS) in simulation-based digital design verification. It argues for the use of this novel genetics-based machine learning technique to perform effective CDG by learning the full mapping between coverage results and test generator directives. Using the resulting production rules, efficient test suites can be constructed, and inference on the validity of the verification environment can be made. There is great potential in using XCS for design verification and this paper forms an initial attempt to highlight the associated advantages. The technique requires no domain knowledge to setup and satisfies important CDG requirements. Once matured, it is expected to be utilized seamlessly in any industrial level simulation-based verification process.**

Electronic Design Automation and Methodology, Digital Simulation, Learning Systems, Learning Classifier Systems, XCS

I. INTRODUCTION

In an industrial context, the verification process is still dominated by the application of dynamic verification or verification by simulation for ensuring the equivalence of the specification of a hardware design with respect to its implementation. An alternative and complementary process also used today is formal verification. As designs become more complex due to (i) the increasing miniaturization level achievable, (ii) the increasing functionality provided driven by market competition, and (iii) the development of more automated design tools, the design verification (DV) stage of the hardware development lifecycle is ever more pressed to provide an effective framework for the development of functionally correct semiconductor designs.

Automating dynamic verification is the only cost-effective way of coping with the increasing complexity, and a step closer to this is automating pseudorandom test generation, or more specifically, automating the manual work involved with analyzing coverage reports and accordingly altering the directives that guide the pseudorandom test generator used. Coverage directed test generation is an effective technique that is aimed at closing the loop between coverage analysis and test generation.

In this respect, academics and industrialists have been working in the past couple of decades on automating CDG with the use of machine learning (ML) techniques [1]. A comprehensive presentation and comparison between the most prominent methodologies utilizing ML techniques on CDG is given in [2].

Although all methodologies currently in the literature achieve good performance on most aspects of CDG, there is still plenty of room for improvement. Inevitably, the solutions currently proposed by researchers introduce new challenges for automated CDG. In addition, experience with working in an industrial context has pointed out the needs and limitations related to the most important element in any contemporary design verification project, which is the verification engineer. The challenges mentioned earlier have influenced the new approach to CDG presented in this work.

Charalambos Ioannides is an EPSRC funded research engineer at the IDC in Systems, University of Bristol.

In this paper, the use of Learning Classifier Systems (LCS) [3] on CDG is presented for the first time as a concept, and a small experimental example is given as a more practical demonstration of this newly proposed methodology. Learning Classifier Systems are chosen because they have the necessary learning mechanisms and functionality that allow them to form a minimal, accurate, and complete mapping between the test generator directive space and the coverage model space, forming rules that indicate their relations in a compact and readable manner,. These can then be used to construct effective regression test suites that can target balancing the coverage attained during test simulation. The latter is a crucial prerequisite for successfully applying CDG on a simulation-based verification project, which has to do with minimizing the resources during the verification process and ensuring multipath coverage on the verification tasks set.

The structure of the paper is as follows: Section II provides background on CDG and LCS. Section III introduces the stated CDG challenges and the attempts to address them with this work. Section IV provides the main argument line justifying the use of XCS in design verification, whereas Section V talks about the experimental methodology that was followed. Sections VI and VII respectively provide the results from the experiments performed, a discussion of the results and the use of XCS on design verification. Finally, Section VIII provides an overview of the work presented and indicates the future steps to be taken regarding this work.

II. BACKGROUND

To follow the material and ideas discussed in this paper, it is necessary to give some background in CDG, LCS and, more specifically, a variant of those, i.e., XCS.

A. Coverage Directed Test Generation

Coverage Directed test Generation is a simulation-based verification methodology that aims to automate the coverage closure process by providing test directives that are based on the coverage feedback attained. It does so by using intelligent algorithms that analyse the coverage report after each simulation phase and direct the next round of test generation towards uncovered events. There are two main approaches towards CDG: one is *by construction* using formal methods, and the other is based *on feedback*. Examples of the former [4] require a formal model of the design under verification (DUV) (e.g. an FSM), which is traversed to derive constraints for test generation that accurately hit a specific coverage task. This approach has inherent practical limitations because the formal models can be particularly large especially as designs become more complex. In contrast, CDG *by feedback* techniques, e.g., [5] or [6], employ ML methods to close the loop between coverage analysis and test generation. This is achieved by learning from the existing tests and the achieved coverage the cause and effect relationships between tests, or test generation constraints, called *biases*, and the resulting coverage. These relationships are then utilized to construct new tests or to create new constraints for a test generator (TG) in such a way that coverage closure is achieved faster and more reliably.

B. Learning Classifier Systems and XCS

Learning Classifier Systems (LCS) [3] is an adaptive rule-based ML technique that 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 multistep, requiring a series of jointly optimal actions. The eXtended Classifier System (XCS), as first introduced in [7], is a very popular LCS variant.

XCS is known to learn a complete and accurate mapping, f_{XCS}: $X \times A \rightarrow P$, from states (X) and actions (A) to payoff predictions (P) for any given problem. According to the *generalization hypothesis* [7], XCS evolves classifiers that are as general as possible without losing accuracy. Furthermore, the *optimality hypothesis* [8] states that XCS develops, given enough resources (i.e., sufficient population N and training cycles), minimal representations of optimal solutions, i.e., the optimal population termed as [O].

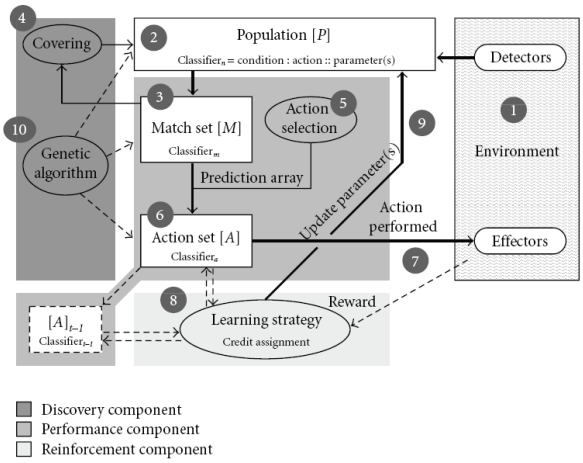

Figure 1. Overview of the eXtended Classifier System [11]

Fig. 1 gives an overview of the mechanisms involved in XCS operation. At first the environment message, i.e., the problem example, is detected and fed into the system. The environment message is a feature vector in ML terms, which is usually a bit vector. The next step is to find any classifiers whose condition part is matching the message, thus forming the Match set [M]. Depending on whether the system is exploring or exploiting its environment, it either chooses a random action or an action expected to fetch the highest reward. From this process, the Action set [A] is created based on classifiers in [M] that propose the chosen action. According to the effectiveness of the proposed action, a reward value is fed back from the environment to the system, which is used to update the statistics kept for each member of [A]. These statistics determine the usefulness of each classifier toward solving the problem posed. This learning cycle is called a learning epoch and is repeated according to a parameter set before each experiment. To explore the problem space and find better classifiers, the genetic algorithm mechanism is activated at fixed intervals. The second discovery component

is 'covering,' which is applied when [M] contains classifiers that propose some, but not all possible, system actions. Therefore, 'covering' creates new random classifiers that propose the missing 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 [9]. A very good overview of the evolution of classifier systems is given in [10].

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 '*don't care*' symbol (#) represents generalization over '0' and '1' values.

III. CDG CHALLENGES

According to [11], the main aims of CDG are to improve the coverage rate, to help reaching uncovered tasks, and to provide many different ways to reach a given coverage task. Linked to the latter aim is the ability to expose bugs more easily, and also to be in the favourable position of balancing the coverage achieved by the tests generated; this in turn optimizes the time and effort required per coverage task.

CDG involves the use of biased pseudorandom test generation. Though this process tends to produce many tests effortlessly, there is no 'memory' or any learning process occurring that would prohibit the repetition of directives already executed and, consequently, reduce the coverage hit count of tasks already reached. Although heuristics have been developed [12] as part of the CDG process, they require guidance by engineers and are not built into a complete and automated CDG method.

To solve some of these problems and in further automating CDG, several ML techniques have been used to help close the loop between coverage results obtained and test directives run on the design under verification (DUV). Despite the overall successful application of these techniques, some additional challenges remain that require either their improvement or the search and use of another methodology.

For example, the application of genetic algorithm (GA) [13], [14] or genetic programming (GP) techniques [15], [16] has produced better results than random test generation in terms of percentage of coverage closure and coverage progress rate. The solution sought by either technique, however, was usually in terms of a single 'rule' or 'test' that would attempt to cover the entire coverage space chosen. As expected, though, the design verification process is never as straightforward. It requires different rules or tests to exercise different tasks in the coverage model used.

Graph-based learning techniques such as Bayesian networks (BN) [11], [17] and Markov models (MM) [18], [19], successfully develop a probabilistic inference network that is very effective for the entire coverage model used, even for guessing directives that might potentially cover previously uncovered tasks. They also, however, require additional effort to use. This is in terms of background knowledge required on the techniques, as well as effort in constructing the networks themselves (especially in the case of BNs and when manual network construction is preferred over automatic). In effect, the verification engineer is required to be an expert in both the DUV and the ML technique and be able to manipulate either or both sides to exploit the latter's full potential.

This fact points to the importance of the human factor when choosing an automated methodology. The typical verification engineer is unaware of these learning techniques, how they work, and how they can best be used. Verification engineers would be better left to influence the elements of the DV process that they have traditionally been influencing (i.e., test generators, coverage models, etc.). Personal experience in working with engineers has shown that they prefer any automation to deal with optimizing the TG and coverage closure process without requiring their immediate attention. Their expertise should be used to set up the TG, choose the coverage model, and facilitate rather than closely monitor the verification process and, more importantly, the coverage closure of tasks.

A final challenge does not come from CDG, DUVs, or engineers, but rather from the verification process used. Today's DV projects require the use of regression suites to test between multiple versions of DUVs as they evolve during a project. Therefore, developing and maintaining a set of rules that are used to construct a regression suite, instead of the regression suite itself, can minimize the effort and resources required to maintain and run regressions as the DUV is modified. This way, a layered and structured design development and verification can be facilitated, building upon previously learnt relationships between test generation and its coverage effectiveness.

IV. MOTIVATION FOR XCS

All of the aforementioned challenges can be met by choosing XCS as the ML technique to be used in further helping CDG. The discussion in this section attempts to explain why.

As introduced in Section II.B, XCS is capable of learning the complete, accurate, and minimal mapping between two spaces, namely the input (feature) and output (concept) space, as used in any classification task in ML. In terms of CDG and the design verification problem in general, the input space would be the directives driving a test generator, and the output space would be the coverage space corresponding to either a simpler code coverage model or a functional coverage model. The task of XCS would be to find the complete and minimal mapping between the two spaces, i.e., find some generalization of test generator directives that cover similar groups of tasks in the coverage model used.

After evolving these rules via XCS, it would be possible to use them to find out in how many ways a particular coverage task can be reached and to plan a regression suite that balances, as much as possible, the coverage between the tasks in the

coverage model. Additionally, each rule would constitute a niche in the test directive space that groups highly effective directives that reach specific coverage tasks. This way, different solutions (test directives) would be learnt for different problem subspaces (coverage tasks).

Also, using XCS on DV means that the verification engineer needs only to construct the test generator, choose the directives, choose the coverage model, and make sure that the feedback reaches XCS for its evaluation mechanism to work properly. Other than this, no knowledge on how XCS works is required. Effectively, what the verification engineer influences is only tasks he/she has already been dealing with traditionally, but now with the added value that cause-and-effect relationships during the verification phase will be represented in the rules learnt by XCS.

The rules developed by XCS can be used throughout the hardware development lifecycle because they form the basis of how design artifacts (DUV structure) correspond to/are affected by input stimuli coming from a test generator. Even if there are changes in the DUV, as functionality is added or debugged during a project, the rules provide a basis from which the adaptive mechanism in XCS can work to reach a correct cause-effect rule, avoiding a completely new search process.

The only prerequisites are that the test generation process uses a distinct learnt XCS population of rules per seed for the test generator, and that the length of the tests developed by the directives learnt are of fixed size. These constraints are to help the one-to-one cause-and-effect relationship between directives and coverage to be discovered and maintained, especially since pseudorandom test generation and coverage attained by it is, in principle, a stochastic process.

Additionally, although the rules developed are not in natural language, they are still in the technical language the engineer chose to represent the bias and coverage spaces. Therefore, they should not be particularly difficult to interpret into useful CDG process and design knowledge.

As an added value, the design verification problem can be formulated in more than one way using XCS. For example, XCS can deal with single as well as multistep problems. This means that learning could be performed at the test directive level after simulating the tests developed (single step), but it could also be performed at the instruction level, online, as individual input vectors are fed to the DUV during the simulation process (multistep). Any work performed on one problem formulation can provide benefit for the other because the learning mechanisms in XCS are almost identical.

The aforementioned advantages motivated this work and indicate the potential of this technique. This paper gives the first step toward achieving this potential, and the next section explains the initial investigation taken.

V. METHODOLOGY

In this section, the DUV, the problem representation, and the experimental setup details are presented in an attempt to explain the approach followed.

A. Design Under Verification

FirePath is a fully featured 64-bit LIW DSP processor [20]. It consists of a large register file, 4 to 6 SIMD execution pipes and 2 load/store pipes (of which a total of 4 can be activated by a single instruction). The design has been in production for 8 years and is the main processing element in Broadcom's Central Office DSL modem chips. This chip is installed in the telephone exchange and is estimated to terminate around 50% of all DSL connections worldwide. The processor is in continuous development as new demands for functionality materialise and as new ideas for improvements in speed, area, or power are implemented. Although the verification environment is very thorough, its maintenance is a labour-intensive process that will benefit from greater automation.

As the size and complexity of FirePath is beyond the current capabilities of XCS, it was decided to target for verification a part of its functionality. Therefore, the particular DUV chosen for this work was the accumulator stage of the long pipeline of FirePath. More specifically, the task will be to discover the relation between biases affecting the opcodes used in a test to toggle the control signals of the accumulator stage of the long pipeline. Control signals are deemed important to toggle, as experience has shown that their full exercise can maximize the code coverage on the DUV RTL code.

B. Problem Representation

XCS can handle different types of problem representations. It has traditionally been targeted on Boolean problems, using the ternary representation but has also been applied on function approximation problems that involved real number and interval representations [21].

The ultimate goal when applying XCS on design verification, and, more specifically, at the bias-coverage abstraction level, is using a real or integer number representation to learn the connection between biases expressed as numbers (probability distributions) and the coverage count for each of the tasks chosen. For practical reasons though, as this is the first time XCS has been applied on such a problem, it was decided to first formulate the problem on a representation closer to the more classical representations XCS deals with. Therefore, the ternary alphabet was chosen, where the biases and coverage feedback are bit strings of fixed length.

Originally, it was 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 21, enough to represent all types of FirePath opcodes.

The cross-product coverage model chosen was the toggle coverage of all control signals in the accumulator stage of the long pipeline. Toggle coverage refers to whether a value of a signal 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 *coverage* bits to be represented.

Before attempting to solve the entire problem as described, it was decided to use XCS on a smaller proof of concept scenario. The smaller example grouped bias controls in a more compact way, thus requiring only 7 bits to represent the

conditions of classifiers. Accordingly, the coverage space was minimized using only 4 bits for representing classifier actions.

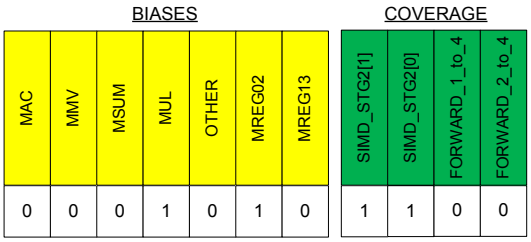

Figure 2. DV Problem Representation

Explaining the bias bit string syntax in Fig. 2, the first 4 bits belong to groups of opcodes that are executed in the long pipeline of FirePath, whereas the fifth bit corresponds to all other opcodes in the instruction set architecture. The last two bits of the bias string control the size of the data inside 4 special registers related to the execution of the long pipeline. On the coverage bits, the first two belong to a 2-bit wide signal indicating the width of the data moving down the long execution pipeline. The third and fourth signals are indicators on whether pipe forwarding is permissible from stage 1 of the long pipeline to stage 4, and stages 2 to 4, respectively.

C. *Test Generator*

FireDrill is a test generator specifically developed to produce instruction streams for the FirePath verification environment. The behavioural model of Firepath is integrated with the test generator so that the generator has full visibility of the architectural state of the processor while generating each instruction. The generator manages constraints governing the correct formation of instructions (e.g., the operations that can be combined in the same packet, valid combinations of destination registers and addresses, etc.) and ensures that each generated test will execute correctly and terminate (loops are bounded, code and data regions are correctly separated, etc.).

Test suites are generated to target various functional coverage metrics [22] using handwritten bias files that can specify instruction sequences designed to increase the likelihood of covering certain corner cases. Each time new features are added to the processor, more metrics and associated bias files are created to cover the new features. Coding the metrics, debugging them, and analysing the coverage and biasing into the gaps consume most of the verification engineers' time and are on the critical path of RTL delivery.

D. *Experimental Setup*

In the experimental setup, XCS alternates between *explore* and *exploit* trials. During explore trials, the classifiers update their statistics, whereas 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 by FireDrill. 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 signals (actions). The test generator in our setup constructs tests containing 100 instructions using a fixed seed. Rules that correctly predict the toggle coverage feedback on the control signals are assigned a reward of 1000, and the incorrect are assigned a reward of 0. The experimental loop, as explained above, is shown in Fig. 3.

According to the learning process mentioned earlier, there would need to be a call to the simulator at every learning epoch. On average, every call requires 90 seconds of simulation, which is quite expensive, especially because a problem of the size as the one chosen would require several tens of thousands of learning epochs. To tackle the time problem, it was decided to simulate only newly proposed biases and save their coverage results in an internal lookup table. This way, unnecessary calls to the simulator were avoided, and experimental runs were sped up by three orders of magnitude.

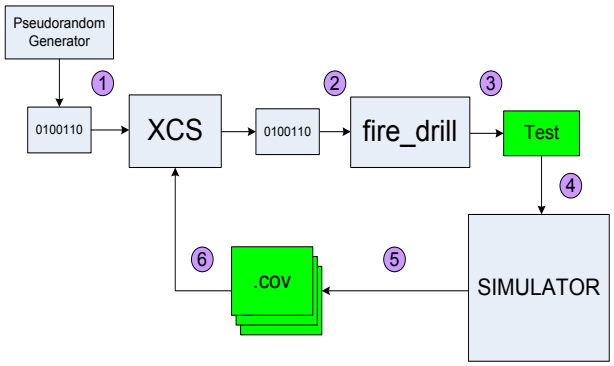

Figure 3. Main experimental setup

The XCS system used is the XCSlib v.1.1 developed at the Politecnico di Milano by Prof. Pier Luca Lanzi et al. [23]. The parameter settings used for all experiments in this work are the standard ones used by XCSlib.

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 size of the population of rules, which is the percentage of N (population size). The three main metrics, as shown in Fig. 4 and for each exploit problem, denote the moving average of the last 100 exploit problems.

VI. RESULTS

During the experiments, all signals were toggled, though not by using a single bias vector. It was discovered that, for full coverage, more than one vector must be used. This is fundamentally a property of the relationship between the specific biases, test generator, and control signals chosen as the coverage model. Given this relationship, XCS learnt most of the rules that describe it. Using these rules, engineers can plan test suites that either achieve full coverage in the most time-efficient manner or test suites that attempt to reach each cover point in more than one way.

The system-level performance, which corresponds to the ability of the system to predict the coverage achieved by the simulation process varies between 80 and 90% for the majority

of the experiments. The average performance out of 10 experimental runs is shown in Fig. 4.

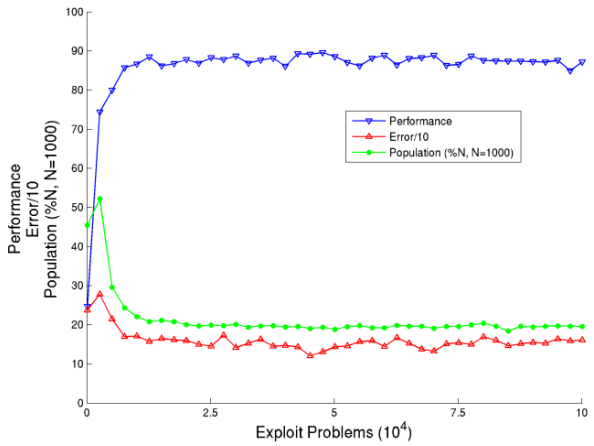

Figure 4. XCS average system level performance

The inability to go above 90% on system-level performance is due to the heavy overlap between the conditions of the optimal rules that describe the problem. This overlap is not allowing XCS to retain all optimal rules, i.e., all rules that minimally describe the relationship between biases and coverage. Therefore, on some occasions, i.e., for some bias vectors, the system is unable to predict the correct coverage they achieve.

After 10^5 learning epochs and ~400 minutes per experiment, the average system-level performance is 86.9%, the average system error is 15.7%, and the average population is 216.5 classifiers. As can be seen on the graph, the system performance reaches the average 86.9% around epoch 5200. Learning could have stopped there, i.e., after approximately 20 minutes without giving much more in terms of performance, but it was allowed to continue in case better results were achieved.

TABLE I. DV PROBLEM - OPTIMAL POPULATION SAMPLE

Classifiers							
ID	**Cond. : Action**	**R**	**E**	**F**	**AS**	**EXP**	**NUM**
1	0###1## : 0000	1000	0	1.00	26.98	22805	21
2	###10## : 0000	0	0	0.48	31.76	390	6
3	0#110## : 1110	1000	0	0.63	23.31	803	6
4	01#10## : 1110	1000	0	0.58	26.06	392	2
5	1#01100 : 0001	1000	0	0.42	9.34	102	3
6	0#100## : 0010	1000	0	0.50	13.29	1534	2
7	01#00## : 0010	1000	0	0.82	17.97	3520	8
8	1###0#0 : 1100	1000	0	0.62	20.85	403	4
9	####### : 0011	0	0	1.00	36.16	6285	36
10	####### : 0110	0	0	1.00	44.35	6157	30

In terms of population performance collectively out of all 10 experiments, XCS learnt 69 out of the 113 optimal rules (~61%), though as mentioned earlier, due to the high overlap between classifier conditions, it remained at this suboptimal performance. As an example, Table I contains a small sample of optimal rules learnt.

The important data to look at are in columns 2 to 4; these data are the condition in which the classifier is applied as a rule and the action that it proposes (Cond. : Action), the expected reward (R), and the expected error (E) if the classifier action is proposed in that condition. The rest of the columns shown are statistics [7] kept for each classifier within XCS to guide learning towards optimal classifiers. They are included here for completeness to indicate how learnt populations look.

VII. DISCUSSION

A. *On Experimental Results*

Full system-level performance is never achieved because when optimal classifier conditions overlap, the fitness (F) they are assigned is shared between them. If the fitness of a classifier is less than 1, it is prone to deletion from the population. Due to this mechanism, optimal rules are learnt, deleted, and eventually rediscovered - something that has the effect of a fluctuating system-level performance, as seen in Fig. 4. Although this is a known weakness of XCS, see [24] and [25], results from active research are promising.

Despite these problems and considering the classifiers learnt, see Table I, verification engineers can understand a lot about their verification environment and the DUV. They can use these rules to construct alternative test suites and to infer relationships between the bias and coverage variables. These relationships will either coincide or not with the idea engineers have about their task. It is an excellent way to test hypotheses/assumptions about the DUV, the TG, and the overall verification environment and to perform the necessary debugging.

For example, the first classifier on Table I indicates that when MAC opcodes are not used, regardless of using MMV, MSUM, and MUL opcodes and when OTHER opcodes are used in a test, it is expected that none of the chosen control signals will be toggled. This rule refers to 32 bias vectors that should be avoided in a test suite because they will simply not produce any coverage. The rule is given maximum reward, i.e., R=1000, because it is correct. Classifier 2, on the other hand, is wrong (R=0), but it can still be inferred that when MUL opcodes are used in a test, and *no other* opcodes are included, regardless of what the other bias bits say, it is expected that coverage will be achieved on at least one or more signals.

Furthermore, to achieve full coverage, only two tests should be constructed according to the biases in classifiers 3 and 5 or 4 and 5: e.g., bias vectors 0111000 and 1101100 will produce two tests each containing 100 instructions that will toggle all four control signals. In the same manner, an alternative but slightly bigger test suite would include three tests, i.e., those constructed using biases from classifiers 5, 6, and 8, or 5, 7, and 8.

Finally, because there are 4 different signals in the coverage model used, all possible coverage patterns are $2^4 = 16$. In the

experiments, only 9 of them were ever recorded. This means that some combinations of coverage between the 4 control signals are impossible given the $2^7 = 128$ possible biases. An example of how that was learnt and represented by XCS is shown by classifiers 9 and 10, which correspond to two of the seven coverage patterns that cannot be achieved. These rules indicate that signals 3 and 4 and signals 2 and 3 are not expected to be toggled together during a test. Thus an engineer can know which coverage vectors are possible and plan out a series of different test suites achieve maximum coverage.

TABLE II. COVERAGE OVERVIEW

Coverage Overview				
Signals:	**simd_stg2_ip[1]**	**simd_stg2_ip[0]**	**forward_1_to_4_ip**	**forward_2_to_4_ip**
Coverage Vectors:	**1**000 (8), **1**001 (9), **1**100 (12), **1**110 (14)	0**1**00 (4), 0**1**01 (5), 1**1**00 (12), 1**1**10 (14)	00**1**0 (2), 11**1**0 (14)	000**1** (1), 010**1** (5), 100**1** (9)

Table II summarizes the 4-bit coverage vectors associated with each signal in the coverage model. Each vector (action) has some associated aggregated directives (condition), so if the engineer wants to toggle any or all signals he/she needs to group the associated 7-bit directive strings together to construct the appropriate tests. This way an alternative of at least 4x4x2x3=96 different ways of attaining full coverage on the 4 control signals can be achieved.

B. On the Use of XCS on DV

The idea behind the use of XCS on DV is being able to learn the relationships between the test generator biases and the coverage model chosen, as simulations are still running, to balance the coverage on the tasks dictated by the model. When this is achieved, the chances of exercising each task in more than one way are maximized, which increases the chances of detecting a bug in the design.

Also, balancing coverage means that for the same amount of tests, i.e., simulation effort, an equal amount of coverage is achieved for each coverage task. Knowing the contribution of each test to coverage allows choice of an effective group of tests that avoids excessive coverage of some tasks at the expense of others.

XCS can learn rules that will neatly separate out the search space according to the effectiveness of bias vectors in terms of the coverage subspace they exercise. Using the resulting classifier (rule) population, more effective test suites can be planned that, given the effectiveness of the original bias vectors, can cover all tasks in a maximally efficient manner.

Additionally, XCS does not require verification engineers to know how it works internally or how to set its parameters to maximize performance. Engineers need only to set up the test generator and the coverage model for the simulations and choose the size of tests and the fixed seed used for their construction. The learning outcome is a group of rules that collectively express the relationship between the biases and their coverage.

Finally, the chip development lifecycle is an iterative process and often companies go through several updates before the final release of a design. In addition, the reuse of intellectual property (IP) allows the creation of a new design from a combination of parts from previous designs. Due to the above, design and verification teams try to capitalize on the tests already developed for such IPs during previous projects. With XCS, each IP can have its own already developed set of rules and then, when an update is required, these rules can be reused to seed the learning phase, thus making the whole process less time consuming. This is not the same as only keeping the tests from project to project as is currently the case in industrial practice. It is argued here that learning should occur at a higher abstraction level, which requires vastly less space for its storage and is flexible enough regardless of changes in test generators or the designs, given that the same interfaces are kept.

One of the limitations of the approach is that for each experiment, a fixed test generation seed is used. This makes the learning environment easier to learn because a single bias will always give the same coverage. If multiple seeds are used on a single experiment, as is the case in realistic pseudorandom test generation, then this will increase the noise in the environment and will reduce the ability of XCS to reach a complete solution. There are techniques available [26] to make XCS more robust against noise in the environment, but this has not been tried in this approach. Consequently, the effectiveness of XCS in the specific case of DV would need to be proven at a later stage.

Also, as mentioned earlier, the ternary representation limits the true potential and usefulness of XCS in coverage balancing because it can only construct rules to say whether a task is covered but not how many times that has happened. Accordingly, the bias vectors learnt are not a probability distribution over several random variables, but rather flags that indicate the use or exclusion of certain opcodes from the tests constructed. The operands and other details of the tests generated are left to the default behavior of the test generator.

Finally, the size of the problem is relatively small (7 bits for Condition and 4 bits for Action representation), and the upper bound on XCS effectiveness on the DV problem has not been determined yet. Nevertheless, this is not necessarily a negative point because, regardless of the size addressable by XCS, the verification engineer is in the position to choose how large the bias and coverage spaces will be, thus dividing the problem into sizeable chunks that can be dealt with by XCS more easily.

Despite the limitations, all the necessary mechanisms are fundamentally in place, and this work is a simple proof of concept that XCS can pose an effective solution for CDG at the abstraction level of biases to code or functional coverage models. More time is required, however, before the concept is mature enough for XCS to use a real number or interval representation [27] that will allow the learning probability distributions over bias variables that predict the number of times they cover a specific task.

From the work described so far, it is evident that this is not a comparison attempt between ML related CDG methodologies but rather the first step of using an alternative and promising technique on that field.

VIII. CONCLUSIONS

This paper proposes a novel technique to deal with the challenges related to CDG. The proposed technique can be used to greatly reduce the manual labor in the feedback path between coverage analysis and alteration of test generation biases. It does so by performing real-time learning during the biased pseudorandom test generation process in order to form a population of rules that map biases to achieved coverage. This population of rules can then be used to generate minimal test suites that achieve full or more balanced coverage on the chosen coverage model.

Learning Classifier Systems could be used in more than one way to help in the DV problem. For example, they can be used in multistep mode to learn how to reach all coverage tasks during on-the-fly test generation, i.e., as a test is constructed and simulated instruction by instruction. An alternative is this work, which uses XCS in single-step mode to learn the relationship between biases and coverage at a higher abstraction level. This can offer a great benefit to the verification engineer and, ultimately, the DV process due to its versatility, effectiveness, and usability.

Future work should test XCS on verification problems dealing with larger bias bit strings and coverage models. Once the upper bound for XCS effective operation is discovered, the exploitation of different, more useful, representations should be sought. The ultimate goal would be to learn the connection between probability distributions on bias variables to the number of coverage hits on each variable/signal in the coverage model used.

ACKNOWLEDGMENT

This work was supported by the Systems Centre and the EPSRC funded Industrial Doctorate Centre in Systems at the University of Bristol (Grant EP/G037353/1) and Broadcom Corporation. The authors would also like to thank all design and verification engineers at Broadcom's Bristol site that have helped with their guidance and useful feedback over the years and regarding this work.

REFERENCES

[1] T. Mitchell, *Machine Learning*, 1st ed. McGraw Hill Higher Education, 1997.

[2] C. Ioannides and K. Eder, *Coverage Directed Test Generation automated by Machine Learning – A Review*. University of Bristol, Rep. CSTR-11-001, 2010.

[3] J. H. Holland, "Adaptation. In R. Rosen & F. M. Snell (Eds.)," *Progress in theoretical biology IV*, pp. 263-293, 1976.

[4] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New Orleans, Louisiana, United States, 1999, pp. 175-180.

[5] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," 2001, vol. 1, pp. 442-448.

[6] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A functional validation technique: Biased-random simulation guided by observability-based coverage," 2001, pp. 82-88.

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

[8] T. Kovacs, "Evolving Optimal Populations with XCS Classifier Systems," MSc Thesis, Dept. of Computer Science, University of Birmingham, 1996.

[9] 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.

[10] 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, Jan. 2009.

[11] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th annual Design Automation Conference*, Anaheim, CA, USA, 2003, pp. 286-291.

[12] H. Azatchi, L. Fournier, E. Marcus, S. Ur, A. Ziv, and K. Zohar, "Advanced Analysis Techniques for Cross-Product Coverage," *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1367-1379, 2006.

[13] J. E. Smith, M. Bartley, and T. C. Fogarty, "Microprocessor design verification by two-phase evolution of variable length tests," in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, Indianapolis, IN, USA, 1997, pp. 453-458.

[14] A. Samarah, A. Habibi, S. Tahar, and N. Kharma, "Automated coverage directed test generation using a cell-based genetic algorithm," Piscataway, NJ, USA, 2006, pp. 19-26.

[15] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Automatic test program generation for pipelined processors," in *Proceedings of the 2003 ACM symposium on Applied computing*, Melbourne, Florida, 2003, pp. 736-740.

[16] P. Bernardi, K. Christou, M. Grosso, M. K. Michael, E. Sanchez, and M. S. Reorda, "Exploiting MOEA to automatically generate test programs for path-delay faults in microprocessors," Heidelberg, D-69121, Germany, 2008, vol. 4974, pp. 224-234.

[17] D. Baras, L. Fournier, and A. Ziv, "Automatic boosting of cross-product coverage using Bayesian networks," Berlin, Germany, 2008, pp. 53-67.

[18] I. Wagner, V. Bertacco, and T. Austin, "StressTest: an automatic approach to test generation via activity monitors," in *Proceedings of the 42nd annual Design Automation Conference*, Anaheim, California, USA, 2005, pp. 783-788.

[19] I. Wagner, V. Bertacco, and T. Austin, "Microprocessor verification via feedback-adjusted Markov models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1126-38, 2007.

[20] S. Wilson, *The FirePath Processor Architecture Guide*. Broadcom Corporation, 2008.

[21] S. W. Wilson, "Function Approximation with a Classifier System," in *Proc. 3rd Genetic Evol. Comput. Conf. (GECCO-2001)*, 2001, p. 974--981.

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

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

[24] C. Ioannides, G. Barrett, and K. Eder, "XCS Cannot Learn All Boolean Functions," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, 2011.

[25] C. Ioannides, G. Barrett, and K. Eder, "Improving XCS Performance on Overlapping Binary Problems," in *IEEE Congress on Evolutionary Computation*, New Orleans, Louisiana, USA, 2011.

[26] M. Troc and O. Unold, "Self-adaptation of learning rate in XCS working in noisy and dynamic environments," *Computers in Human Behavior*, vol. In Press, Corrected Proof, 2010.

[27] S. W. Wilson, "Get Real! XCS with Continuous-Valued Inputs," in *Learning Classifier Systems, From Foundations to Applications*, London, UK, 2000, p. 209–222.