APPLICATION OF ESOP MINIMIZATION IN MACHINE LEARNING AND KNOWLEDGE DISCOVERY

Marek A. Perkowski
Dept. Electr. Engin.
Portland State University
P.O. Box 751
Portland, Oregon, 97207-0751
mperkows@ee.pdx.edu

Tim Ross, Dave Gadd, Jeffrey A. Goldman
Wright-Laboratory
WL/AARA-3
2010 5th Street Bld 23
Wright-Patterson AFB, 45433-7001
tross@mbvlab.wpafb.af.mil

Ning Song
Lattice Logic Corp.
1820 McCarthy Blvd.
Milpitas CA 95035
ning@lattice.com

Abstract— This paper presents a new application of an Exclusive-Sum-Of-Products (ESOP) minimizer EXORCISM-MV-2: to Machine Learning, and particularly, in Pattern Theory. An analysis of various logic synthesis programs has been conducted at Wright Laboratory for machine learning applications. Creating a robust and efficient Boolean minimizer for machine learning that would minimize a decomposed function cardinality (DFC) measure of functions would help to solve practical problems in application areas that are of interest to the Pattern Theory Group - especially those problems that require strongly unspecified multiple-valued-input functions with a large number of variables. For many functions, the complexity minimization of EXORCISM-MV-2 is better than that of Espresso. For small functions, they are worse than those of the Curtis-like Decomposer. However, EXORCISM is much faster, can run on problems with more variables, and significant DFC improvements have also been found. We analyze the cases when EXORCISM is worse than Espresso and propose new improvements for strongly unspecified functions.

I. Introduction.

Recently, there has been an increasing interest in applying methods developed in design automation to other fields (see DAC'94 and Euro-DAC'94 panel discussions). Amazingly, the techniques developed in the last 15 years by the design automation community have been so universal and powerful, that they are also increasingly used in areas outside circuit design. For instance, they are now used in automatic theorem proving, robotics, industrial operations scheduling, stock market prediction, genetics research, and many others. It is then quite probable, that the design automation methods will lead to breakthroughs in these other fields.

One of the sub-areas of design automation that can find numerous external applications is logic synthesis. Unfortunately, the potential of logic synthesis for external applications is still less appreciated (by both the CAD and Machine Learning communities, and the general research and industrial circles) than the potentials of placement, routing, scheduling, simulation, and database techniques of design automation.

Until very recently, the logic synthesis discipline formulated efficient methods and algorithms to minimize and optimize only the hardware of digital computers and other digital circuits. This goal was achieved very successfully, and the logic synthesis techniques developed in universities and industry in the last 15 years became one of the sources of the success of Design Automation in creation of such products as Intel's Pentium microprocessor. In the last few years, however, one can observe some increased trend to apply these methods also in image processing, machine learning, knowledge discovery, knowledge acquisition, data-base optimization, AI, image coding, automatic theorem proving and verification of software and hardware [4, 15, 22, 7, 8, 9].

This paper is related to the use of an ESOP minimizer in Machine Learning (ML) and Knowledge Discovery in Databases (KDD) applications. We will examine the performance of the EXORCISM circuit minimizer on small binary functions comparing the results with Espresso. References will be made to data on these same test functions with C4.5 and function decomposition. The comparison will serve to highlight the strengths and the shortcomings of all approaches. The goal of this paper is not to go too deeply in algorithms, but rather to demonstrate the applicability of logic synthesis, and specifically EXOR-based synthesis, to KDD and ML. All four algorithms share the common goal of a consistent, minimal complexity solution, albeit different measures of complexity. All four differ in their method to find it.

The paper is organized as follows. In section 2 we will present briefly the basic concepts developed by the Pattern Theory Group at Wright Lab (WL). Section 3 concentrates on DFC and its role. Section 4 will discuss specific requirements for logic minimizers in Machine Learning applications. Section 5 outlines the application of EXORCISM-MV-2 ESOP minimizer for machine learning. Section 6 presents numerical results on learning benchmarks. Section 7 briefly characterizes the machine learning benchmarks of WL. Section 8 presents the way to improve EXORCISM on very strongly unspecified functions. Finally, section 9 concludes the paper and outlines future areas of research.

II. The Basic Research Ideas of the PTG.

The Pattern Theory Group (PTG) in the Avionics Directorate at Wright Laboratory develops new system-level concepts for military applications, mostly based on image processing and machine learning technologies. Since 1988, the PTG developed a radically new approach to machine learning, where by "machine learning" we understand any method of teaching a computer to recognize any kind of patterns. Specifically, we are examining Supervised Classification Learning paradigms. The machine learning technologies most influential in military applications until now have been the neural nets and the bayesian methods. If successfully completed, the new approach of PTG will allow both automatic learning of any kind of images, and automatic creation of algorithms. Interestingly, in contrast to most of the well-known approaches, the approach of the PTG is based on logic synthesis methods: the so-called Curtis Decomposition of Boolean functions is applied here as the main component[18]. Many decomposition ideas have been implemented in the programming system FLASH (the Function Learning and Synthesis Hotbed) developed by this group[20]. This system is a Testbed for machine learning based on the logic synthesis approach. The group also compares FLASH to other logic optimizers and machine learning programs, (such as Espresso and C4.5, respectively) from the point of view of the robustness of learning[7, 8].

Simplifying, the main difference of the logic approach and the previous approaches to machine learning is that in these previous methods, the recognizing network had some assumed structure which was "tuned" by the learning process (for instance, by decreasing and increasing the numerical values of some coefficients). Thus, creating a new structure is accomplished by setting some coefficients to zero. All "new" structures are in a sense "hidden" in the assumed mother-structure. Also the type of an element, such as a formal neuron in Neural Nets, or a polynomial in Abductory Inference Mechanism (AIM[1]) is decided before the learning takes place.

In contrast, in the Curtis Decomposition approach, there is no a priori assumption of structure of the learning network, nor on the type of the elements. The elements are arbitrary discrete mappings (functions) in the form of a combinational network with universal gates. Both the structure and the elements are **calculated** in the learning process, and this process is entirely based on finding patterns in data.

The central concept of Pattern Theory is a "pattern." In Pattern Recognition and Knowledge Discovery the problems with nice representations based on "features" belong to a more general class of problems with restrictive "patterns." Pattern finding is, therefore, a generalization and formalization of feature extraction. The goal of the Pattern Theory is to support "automating" the pattern finding process, and to construct a representation of a function based on examples; therefore, this is a method for constructive induction. Furthermore, it constructs this representation by minimizing complexity as in Occam-based learning. The Ashenhurst Function Decomposition (AFD) method based on Curtis Decomposition implemented in FLASH is unusual in that it learns both the architecture of the combinational representation and the component functions **from** the examples.

Induction is often modeled as the process of extrapolating samples of a function. This extrapolation requires both the samples and the "inductive bias." The bias towards low complexity, as in Occam's Razor, is particularly important. There is a strong theoretical basis for Occam-based learning, see for example [2, 3]. Kolmogorov complexity was developed specifically for induction [14], however, it has been proven that its exact computation is not tractable. There have been some tractable measures of complexity used in actual implementations of Occam-based learning, such as the Abductory Inference Mechanism which uses polynomial networks and C4.5 [17] which uses decision trees. However, the measures of complexity used in these applications are relatively narrow, which implies some compromise in the robustness of the learning; for example, neither of these methods would find the parity function to have low complexity even though it is highly patterned and can be easily computed. The challenge is to develop robust *and* tractable measures of complexity.

Pattern Theory [18] treats robust minimal complexity determination as the problem of finding a pattern. Pattern theory uses Decomposed Function Cardinality (DFC), proposed by Y. S. Abu-Mostafa as a general mea-

[1] Trademark of AbTech Corp.

sure of complexity [1, p.128]. DFC is based on the cardinality of a function. After all, a function is a *set* of ordered pairs and, as with any set, has a definite property in its number of elements or cardinality. DFC is defined as the sum of the cardinalities of the components of a combinational representation of a function, when that sum has been minimized. DFC is especially robust in the sense that it reflects patterns of many different kinds. Its robustness is supported by its relationship to more conventional measures of complexity, including circuit size complexity, time complexity, decision tree or diagram size and Kolmogorov complexity. If a problem has low complexity by any of these measures then it will also have a low DFC [18, Chapter 4]. The PTG work has concentrated on functions with binary inputs, but the concept is easily extended to continuous and multiple-valued functions [21].

The decompositions are evaluated based on the sum of the function cardinality of the decomposed partial blocks. Of course, the real DFC of f is less than or equal to this sum in any particular (usually approximate) realization. The objective of the decomposition is to search through partitions of input variables to find one that produces the smallest total DFC. In other words, the "cost" of the feature is measured in terms of DFC and that cost is minimized. Therefore, features are constructed to minimize their computational complexity. The use of DFC as the measure of complexity allows for robustness in the type of feature that can be generated.

Let us observe that both in the Curtis decomposition and the ESOP minimization approach of EXORCISM, we look for certain patterns in data: in Curtis decomposition these patterns are in columns of the map corresponding to the cofactors of the bond set of variables [5]. The patterns for ESOP minimization are based on certain rules of Boolean Algebra[24].

Let us also observe that in both cases we minimize a certain cost of the circuit. Traditionally in ESOP minimization one calculates the number of terms and the number of literals. In our case, we calculated additionally the total DFC as a sum of function cardinality of all non-decomposable blocks (this is an upper bound approximation of the minimum DFC). For an arbitrary non-decomposable block in Curtis Decomposition, the DFC of the block is calculated as 2^k where k is the number of inputs to the block. In "gate-based" minimizers such as Espresso and EXORCISM it is then fair to assume that a DFC of a decomposable gate (such as AND, OR or EXOR) is equal to the total DFC of a circuit equivalent to this gate, that is constructed from two-input gates. The DFC of a four input AND gate, OR gate, or EXOR gate is then $2^2 + 2^2 + 2^2 = 12$, since such gates can be decomposed to balanced trees of three two-input gates.

One variant of the AFD algorithm looks at all partitions at a given level and 40 random partitions one level deeper. Essentially, this is a 2-ply look ahead search where we only explore 40 grand children. It chooses a particular partition over another based on the function's DFC. Exorcism finds an ESOP, Espresso finds a SOP, and C4.5 finds a low complexity decision tree. Minimizing complexity provides good generalization performance [8, 15, 18].

III. Summary of DFC Measurements and Applications

A number of experiments have been conducted to assess the generality of DFC across different problem domains (see [18, 8, 19]. The DFC of over 800 non-randomly generated functions was measured, including many *classes* of functions (numeric, symbolic, chaotic, string-based, graph-based, images and files). Roughly 98 percent of the non-randomly generated functions had low DFC (versus less than 1 percent for random functions). The 2 percent that did not decompose were the more complex of the non-randomly generated functions rather than some class of low complexity that AFD could not deal with. It is important to note that when AFD says the DFC is low, which it did some 800 times, it also provides an algorithm (or a description of the pattern or features found). AFD found **the** classical algorithms for a number of functions. Each of these algorithms are represented in a combinational form that includes intermediate states. These intermediate states are features in the sense of concentrating information.

There is also high correlation between DFC and a person's judgment of the strength of a pattern in an image, the degree of compression by a data compression program, and the Lyapunov exponents of logistic functions. This shows DFC's ability to reflect patterns within each domain, despite their different nature.

In learning experiments, AFD has been compared to back-propagation trained Neural Networks (NN), AIM, C4.5, SMOG [15] and standard logic minimization tools. These comparisons used a broad range of function types (from the 800 mentioned above). For each of the test functions, AFD performed near the best of any method, while the other methods generalized well on some functions but not on others.

In the context of ML, when one talks about "noise," it is assumed that he is referring to the situation where you have some training data classified correctly as a "1" or a "0" but then "noise" flips that bit to the incorrect entry. Another situation is that the value of some minterm is or becomes unknown - a "don't care". This would be referred to as "unknown" in the ML community. ML techniques can be used to restore noisy images [4]. Of course, we do not know which pixels (or feature values) are "noisy" - so we treat those that are most suspicious as don't cares.

The more robust generalization performance of AFD, including dealing with noise and unknown data, is a reflection of its robust measure of complexity, DFC. It is also an indication that the useful inductive bias of these various standard methods results from their parsimonious tendencies rather than their particular representation schemes (be they thresholded weighted sums, polynomials, decision trees or Boolean operators).

IV. Towards Improved Approaches to Logic Minimizers for Machine Learning

Although the AFD approach of FLASH gives very robust results, it is slow, which essentially restricts its practicality to 20 or even less, variables. It is, therefore, important, to be able to compare the FLASH decompositional logic approach to other logic approaches that use the same DFC measure, but introduce some restricted bias resulting from the assumed network's structure. Such approaches are then faster and can be used for larger variable functions. In this respect, the well-known circuit minimizer Espresso and the standard machine-learning program C4.5 were tested together with EXORCISM. These programs have the following structure/gate-type biases: Espresso assumes a two-level AND-OR network, EXORCISM assumes a two-level AND-EXOR network, C4.5 assumes an ordered tree. (The input variables can be multiple-valued).

The questions arise:

- How much of the network's simplicity is lost by assuming these structures?
- How much is gained in the speed of the program with respect to a bias-free decomposer? Is this speedup worth an increased DFC and thus a more limited extrapolation capability?
- Is the method with a biased structure still robust enough for practical applications?

Other important question that must be faced with while developing improved minimizers for machine learning applications is the following:

- What are the reasons that machine learning using logic synthesis is not exactly the same as circuit design using logic synthesis?

This question is very important practically. Improving the performance of the FLASH system orders of magnitude without sacrificing much of its robustness (DFC) is required for making it useful for such important military applications as High Resolution Radar, for example.

The data (switching functions) used in learning and algorithm design applications by the PT group are *arbitrary switching functions*. Thus, the standard and generally applicable minimization procedures of "logic synthesis' can be applied. An extremely important observation is that these functions have *quite different properties* than the data taken from industrial companies on which the programs are tested in the "logic synthesis" community (MCNC benchmarks). In theory, the algorithm should work well on any type of data. However, since all practical network minimization problems are NP-hard, all practical algorithms, by necessity, are heuristic in nature. Thus, they are very dependent on the type of data. Taking into account the data characteristics (such as closeness to unate Boolean functions) was, in principle, the main reason of the commercial success of two-level logic minimizers in circuit-design applications.

What is it that distinguishes the machine learning data from the circuit design data? Our preliminary answer is the following:

1. ML problems have an extremely high percent of don't cares (Don't cares are combination of argument value for which the function value is not specified.) The missing data can be represented as don't cares.
2. Arguments (variables) in ML problems are naturally multiple-valued, or continuous.
3. ML problems involve missing values in inputs (missing fields) and conflicting data for discrete as well as continuous fields.

V. ESOP Minimization for Machine Learning

In the past, EXORCISM-mv-2 was tested very successfully on industrial circuits that have up to 80% of the values as don't cares [24]. Comparisons with EXMIN [23] and MINT [13] demonstrates that EXORCISM either finds the best solution, or finds one that is nearly the best one of the three. EXMIN does not allow for don't cares so it is of little use to machine learning, and MINT is slower than EXORCISM. Its approach to handling don't cares is very similar to the older variant of EXORCISM. One can then assume that the following critical remarks about EXORCISM apply to all current ESOP minimizers.

Testing EXORCISM on "machine learning" applications, we found that its performance on learning benchmarks is still unsatisfactory on larger examples. Analyzing these cases, we came to the conclusion, that the inferior behavior of the approach is caused predominantly by the extremely high percent of don't cares that is typical for machine learning benchmarks, and can be as high as 99.99%. Espresso beats EXORCISM in cases where functions had a high percentage of don't cares.

However, EXORCISM is able to find a pattern of EXOR (parity) or similar functions, even when it is corrupted by "unknowns". This is a difficult problem in machine learning. To explain this case on an example, let us assume that we recognize the even/odd parity function. For a completely specified function and a relatively small (less than 16) number of variables, the AFD minimizer finds the exact minimal result (EXOR of input variables) quite fast. When we add some "unknowingness" to this function by replacing some ones and zeros with "don't cares", we should still be able to find the EXOR of inputs solution, since the underlying principle function did not change, only its pattern has been corrupted, "hidden" by the unknown values. This seemed to work, but when the percentage of don't cares increases and the number of variables increases, the average error increases and the method yields poor results. First it ceases to recognize the "EXOR of variables" pattern, and second, on 96-variable functions, it finds no EXOR's at all and looses track of any patterns (so would a human on this case). As seen in tests, EXORCISM deals better with these kinds of problems, but is still unable to solve the 96-variable problem.

Another positive property of EXORCISM is simultaneous classification of patterns to more than two categories (you want not only to distinguish a "friend from foe" airplane, but you want to learn its orientation, speed, etc.). In terms of logic synthesis, this property corresponds to concurrent minimization of switching functions with many outputs. Currently FLASH operates on single-output functions, but EXORCISM works with multi-output functions. There are many decomposers that decompose multi-output functions, but all of them have been designed for circuit design. One needs a minimizer for **strongly unspecified, multi-valued input, multi-output functions.** EXORCISM (as well as Espresso) both satisfy this requirement, but they both can be improved for strongly unspecified functions.

What is also missing in both "industrial circuit" and "machine learning" decomposing systems, is the **decomposition of multiple-valued input, multiple-valued-output functions.**

Why is this important? In theory, which is also the approach of the PT group, any multiple-valued variable can be encoded by a vector of binary variables. What happens, however, in learning situations is, that the learning system inferences rules that depend on the encoding of multiple-valued variables with binary variables. To give an example, if the system would infer from a large set of data that people who live close to power lines develop cancer, we would perhaps treat such "invention" with due care. If the system would, however, infer that people whose third bit of encoded distance from the line is 1 develop cancer, we would treat such inference as a "coding-related" artifact. Therefore, the best approach to the learning system would be not to use coding at all, but perform the inference on the variables that are natural for any given feature; e.g. either binary (man, woman), or multiple-valued (distance in yards). EXORCISM has this property.

VI. Numerical Results

This section compares the results of Espresso and Exorcism and gives a partial description of functions from the "Learning Benchmark." In the first table, we show the On-Set, Don't-Care Set, the number of terms, the number of literals, the calculated DFC, the CPU time, and the average number of errors for a learning experiment for every function with Espresso. The second table is the same categories for EXORCISM.

The *average error* on the individual functions were calculated as follows. First, each method was given a random set of data to train on ranging from 25 to 250 out of a total of 256 possible cares. Once the method was trained, the entire 256 cases were tested and the number of differences were recorded as errors. This procedure was repeated 10 times for a given sample training size in intervals of 25. Thus, the total number of runs for each function was 100 of varying sample size. None of the learning was incremental. All of the runs were independent. For each function, the average number of errors for the entire run was recorded in the table.

Espresso generalizes as well as C4.5 - a main stream ML method[7]. It, however, does have a weakness when the function to be learned is most naturally represented with EXORs. This points out to the complementary nature of these two programs in the search of low DFC solutions. Lower combinational size complexity (DFC, gate count, Decision Tree node count, Decision Diagram size, etc) provides better generalization [15, 6, 19, 22]. There is greater than 0.9 correlation between complexity reduction and generalization performance for both SMOG (RODD size) and FLASH (DFC) [22]. In the two comprehensive tables, we provide some results for functions that do not have a low complexity SOP representation but do have a low complexity ESOP for total functions, such as parity and palindrome.

VII. Characterization of Benchmark Functions

We thought it may be interesting to describe some examples of benchmarks that we used in addition to MCNC benchmarks[2]. Some of the benchmark names from the tables are separated into several groups listed and briefly explained below.

LEARNING8_SET. This set of functions is intended to be representative of a wide variety of functions for testing machine learning systems.

RANDOM. There are 3 randomly generated functions, generated with FLASH with seeds 1,2, and 3: *rnd1, rnd2,* and *rnd3.*

RANDOM MINORITY ELEMENTS. There are 5 functions generated with a fixed number of minority elements placed at random. The seed for all was 1: *rnd_m1, rnd_m5, rnd_m10, rnd_m25, rnd_m50.*

BOOLEAN EXPRESSIONS. These are functions intended to represent database concepts for knowledge discovery[7]. *kdd1*: (x1 x3) + x2'; *kdd2*: (x1 x2' x3)(x4 + x6'); *kdd3*: NOT (x1 OR x2) + (x1' x4 x6); *kdd4*: x4';

[2] More "machine learning benchmarks" are available from U.C. Irvine. (http://www.ics.uci.edu/AI/ML/MLDBRepository.html)

Function	ON	DC	terms	literals	DFC	time	average error
add0	120	0	15	64	252	0.15	19.86
add2	128	0	16	68	268	0.23	28.6
add4	128	0	2	4	12	0.1	0.8
ch15f0	88	0	12	60	236	0.12	30.02
ch176f0	64	0	2	6	20	0.1	1.4
ch177f0	128	0	2	4	12	0.11	0
ch22f0	48	0	6	30	116	0.13	16.01
ch30f0	64	0	7	32	124	0.11	17.06
ch47f0	52	0	9	48	188	0.11	24.54
ch52f4	50	0	18	108	428	0.16	26.87
ch70f3	24	0	5	28	108	0.08	12.36
ch74f1	39	0	10	58	228	0.11	21.71
ch83f2	38	0	17	115	456	0.12	30.41
ch8f0	224	0	7	16	60	0.28	12.08
contains 4_ones	70	0	70	560	2236	0.22	63.94
greater_than	120	0	15	64	252	0.15	20.26
interval1	58	0	16	96	380	0.24	29.12
interval2	128	0	22	110	436	0.46	35.8
kdd1	160	0	2	3	8	0.12	0.96
kdd10	120	0	8	28	108	0.26	17.16
kdd2	24	0	2	8	28	0.08	12.86
kdd3	80	0	2	5	16	0.1	3.52
kdd4	128	0	1	1	0	0.11	0
kdd5	106	0	4	13	48	0.12	8.44
kdd6	240	0	4	4	12	0.19	2.64
kdd7	175	0	4	8	28	0.15	5.69
kdd8	64	0	2	6	20	0.09	5.64
kdd9	64	0	8	36	140	0.16	16.54
majority gate	93	0	56	280	1116	0.4	31.68
modulus2	43	15	10	45	160	0.12	14.73
mux8	128	0	4	12	44	0.1	8.49
pal	16	0	16	128	508	0.11	32.2
pal_dbl output	160	0	29	151	600	0.31	45.91
pal_output	118	0	46	291	1156	0.48	60.43
parity	128	0	128	1024	4092	0.42	82.6
remainder2	88	15	23	137	528	0.18	29.57
rnd_m1	1	0	1	8	28	0.06	104.51
rnd_m10	10	0	9	71	280	0.09	25.8
rnd_m25	25	0	20	154	612	0.12	37.27
rnd_m5	5	0	5	40	156	0.09	36.62
rnd_m50	50	0	34	250	996	0.16	52.12
rnd1	122	0	50	324	1320	0.41	62.91
rnd2	124	0	47	292	1164	0.39	61.35
rnd3	134	0	49	306	1240	0.5	59.56
substr1	142	0	6	18	68	0.13	12.8
substr2	79	0	5	20	76	0.1	14.93
subtraction1	104	0	34	200	796	0.28	47.26
subtraction3	128	0	2	4	12	0.11	0.8

TABLE I
Espresso on Machine Learning Benchmarks.

Function	ON	DC	terms	literals	DFC	time	average error
add0	120	0	15	68	268	0.61	58.91
add2	128	0	5	10	36	0.18	59.35
add4	128	0	2	2	4	0.14	59.38
ch15f0	88	0	9	37	156	0.24	40.07
ch176f0	64	0	2	4	12	0.08	28.58
ch177f0	128	0	2	2	4	0.14	59.34
ch22f0	48	0	6	30	116	0.1	22.25
ch30f0	64	0	7	32	124	0.12	29.39
ch47f0	52	0	7	34	132	0.13	23.48
ch52f4	50	0	15	93	392	0.84	22.04
ch70f3	24	0	5	28	108	0.09	9.82
ch74f1	39	0	10	58	248	0.2	18.72
ch83f2	38	0	13	77	304	0.32	16.65
ch8f0	224	0	7	28	124	0.36	103.88
contains 4_ones	70	0	40	224	968	8.23	32.65
greater_than	120	0	15	72	284	0.61	57.83
interval1	58	0	16	98	388	1.44	26.9
interval2	128	0	19	88	308	1.62	59.16
kdd1	160	0	2	4	12	0.18	73.8
kdd10	120	0	4	10	40	0.16	55.28
kdd2	24	0	2	8	28	0.07	18.05
kdd3	80	0	2	6	20	0.09	33.48
kdd4	128	0	1	1	0	0.14	58.9
kdd5	106	0	6	27	108	0.16	50.25
kdd6	240	0	2	4	12	0.31	111.3
kdd7	175	0	15	64	268	0.54	82.68
kdd8	64	0	2	4	12	0.08	32.25
kdd9	64	0	4	12	44	0.11	28.09
majority gate	93	0	34	189	732	5.73	45.52
modulus2	43	15	9	52	140	0.13	18.96
mux8	128	0	4	12	44	0.15	59.92
pal	16	0	16	72	268	0.43	16.82
pal_dbl output	160	0	22	92	412	2.14	74.45
pal_output	118	0	37	206	820	16.32	55.1
parity	128	0	8	8	36	1.38	11.1
remainder2	88	15	19	108	416	3.86	36.89
rnd_m1	1	0	1	8	28	0.06	102
rnd_m10	10	0	9	62	268	0.14	11.68
rnd_m25	25	0	18	122	448	0.53	13.52
rnd_m5	5	0	5	38	148	0.07	26.75
rnd_m50	50	0	27	169	680	3.12	22.97
rnd1	122	0	39	212	812	7.3	56.77
rnd2	124	0	34	179	684	9.82	57.37
rnd3	134	0	39	213	812	8.63	63.05
substr1	142	0	14	69	272	0.4	66.63
substr2	79	0	6	28	108	0.18	37.6
subtraction1	104	0	20	98	404	2.52	47.32
subtraction3	128	0	2	2	4	0.14	59.38

TABLE II
EXORCISM on Machine Learning Benchmarks.

kdd5: (x1 x2 x4')+(x3 x5' x7 x8)+(x1 x2 x5 x6 x8)+(x3' x5'); *kdd6*: x2 + x4 + x6 + x8; *kdd7*: (x1 x2) + (x3 x4) + (x5 x6) + (x7 x8); *kdd8*: (x1 x2') XOR (x1 x5); *kdd9*: (x2 XOR x4)(x1' XOR (x5 x7 x8)); *kdd10*: (x1 $\rightarrow$ x4) XOR (NOT (x7 x8))(x2 + x3).

multiplexer, used in [12], this is a 2-address bit, 4-data bit multiplexer with two vacuous variables (x0 and x1) to make 8 inputs.

STRING FUNCTIONS. Palindrome acceptor, *pal*, palindrome output, *pal_output,* randomly generated 128 bits then mirror imaged them to create the outputs of an 8 variable function. Doubly palindromed output, *pal_dbl_output,* as above but generated 64 bits and flipped them twice. 2 interval acceptors from FLASH, *interval1* accepts strings with 3 or fewer intervals (i.e. substrings of all zeros or all ones). *interval2* accepts strings with 4 or fewer intervals 2 sub-string detectors. *substr1* accepts strings with the sub-string "101". *substr2* accepts strings with the sub-string "1100".

IMAGES. *chXfY* means character X from font Y of the Borland font set. All were generated with the Pascal program charfn.exe. *ch8f0* - kind of a flat plus sign, *ch15f0* - an Aztex looking design, *ch22f0* - horizontal bar, *ch30f0* - solid isosceles triangle, *ch47f0* - slash, *ch176f0* - every other column of a checker board, *ch177f0* - checker board, *ch74f1* - triplex J, *ch83f2* - small S (thin strokes), *ch70f3* - sans serif F, *ch52f4* - gothic 4.

SYMMETRIC FUNCTIONS. *parity. contains_4_ones*, (f(x)=1 iff the string x has 4 ones). *majority_gate*, f(x)=1 iff x has more 1's than 0's.

NUMERICAL FUNCTIONS. *addition, add0, add2, add4* - outputs bits of a 4 bit adder, 0 is the most significant bit. *greater_than:* f(x1,x2)=1 iff x1 > x2. *subtraction: subtraction1, subtraction3* - output bits 1 and 3 of the absolute value of a 4-bit difference. 0 is most significant bit. *modulus2,* output bit 2 of 4-bit modulus 0 is the most significant bit. *remainder2,* output bit 2 of 4-bit remainder 0 is the most significant bit.

VIII. Proposed Improvements to EXORCISM for Strongly Unspecified Functions.

In data with very many don't cares, EXORCISM is unable to find the best pattern. As an example, let us consider function f_1 defined as follows: ON = {0X10, 1X01}, OFF = {0X01, 1111, 1010}. When cube 0X10 is extended to 0X1X, and cube 1X01 is extended to 1X0X, these two extended cubes can be correctly exorlinked. However, when there are many don't cares surrounding an ON-cube, the program does not know which of them to select. (*Extension cube* of cube C_K is cube C_K with any subset of literals removed - including none and all). For instance, if cube 1X01 is extended to 1X0X and cube 0X10 is extended to 0XX0, the minimization is more difficult. If DC-cubes 0011 and 0111 were merged to DC-cube 0X11, then this DC-cube can be exorlinked with 0XX0 to cubes 0XXX and 0X01. Next 0X01 and 1X0X can be reshaped to 0X00 and XX0X. Finally it can be found using disjoint sharp operation that 0X00 is included in DC set, so the final result is XX0X and 0XXX.

We separate cubes used in synthesis into ON cubes, ON/DC cubes and OFF/DC-cubes. ON cube includes only true minterms. It is then a cube C_j such that C_j $\#$ ON $= \phi$. ON/DC cube includes true minterms and don't cares. It is then a cube C_l such that $C_l \cap$ OFF $=$ ϕ. OFF/DC cube includes false minterms, and possibly don't cares. # denotes sharp operation.

In this section we propose a new method of combining Espresso and EXORCISM into a single program, called EXORCISM_DC.

The algorithm given below, using Espresso, creates larger ON, DC and ON/DC cubes to be used by EXORCISM. It iterates for several starting points and finally, when no improvement is possible, it returns the best of all ESOP and SOP solutions. Thus, its result is never worse than that of either Espresso or EXORCISM.

EXORCISM_DC
Given: ON, OFF.
1. DC := Espresso[1 # (ON ∪ OFF].
2. SOP := Espresso(ON, OFF).
3. ON := Random_Disjoint(Espresso(ON)).
4. ESOP := Random_Disjoint(SOP).
   ON/DC := ON ∪ Random_Choice(DC,ON,ESOP).
   ESOP_new := Exorcism(ON/DC,DC).
   Remove from ESOP_new all OFF/DC cubes.
   if cost(ESOP_new) < cost(ESOP)
     then ESOP := ESOP_new.
5. [ON/DC, DC] := Random_Reshape[ON, DC, ESOP].
6. Iterate k_1 times steps 4 - 5.
7. If improvement in steps 4 - 6 then go to 4.
8. Iterate k_2 times steps 3 - 7.
9. If improvement in steps 3 - 8 then go to 3.
10. If cost(SOP) < cost(ESOP)
   then return ["SOP", SOP, cost(SOP)].
   else return ["ESOP", ESOP, cost(ESOP)].

Above, *Espresso* is the well-known U.C. Berkeley minimizer that can be called with several data formats, including completely specified format with set ON, and incompletely specified format with sets ON and OFF. *Exorcism* is our minimizer that can be called with any ESOP (including disjoint ON) as the first argument and any set DC as the second argument. *Random_Disjoint* is a program that takes a set of cubes and transforms it to random set of disjoint cubes. Every time this program is called it produces different set of disjoint cubes which is functionally equivalent to its argument. *Random_Choice* takes a random subset of cubes from DC that are expected to be exorlinked with cubes from ESOP. *Random_Reshape* reshapes each set ON and DC separately.

IX. Conclusions and Future Research.

This work proved that EXORCISM performs well as a machine learning program on functions with a small and a medium percentage of don't cares. The data support the conclusion that EXORCISM compares favorably to Espresso in general and fills in some of its weaknesses. While EXORCISM in its present form does not minimize partial functions well, we outlined some proposed changes and have further developed the algorithm in [16]. We can then conclude that EXORCISM has potential as a viable ML method that would especially complement existing ML techniques.

Our goal is the creation of a **practical** "machine learning" algorithm, which means at the minimum, 30 binary input variables but more likely, about 100 multiple-valued variables. As presented above, in machine learning, with the increase in the number of input variables there is only a small increase in the number of both positive and negative samples, but a dramatic increase in the number of don't cares. For instance, it is reasonable to expect that for a function of 100 variables there will not be more than 10,000 cares.

The program must be robust across various classes of data from the learning benchmarks. Combining SOP and ESOP minimizers, like Espresso and EXORCISM, into a single program will create a program that would be superior to both of them. We believe that this analysis pinpoints strengths and weaknesses of all analyzed approaches: AFD is clearly superior on small functions but it is not yet tractable on larger ones; Espresso has trouble with "counting" type of dependencies such as parity and arithmetic circuits but handles don't cares relatively well; EXORCISM is superior to Espresso and C4.5 on some functions, in spite of the uncorrected problem of very strongly unspecified functions. C4.5, the defacto standard machine learning tool, can handle more special cases of data, such as noise and missing input values.

The observation that functions in Machine Learning are very strongly unspecified and thus none of the known approaches work well, makes the requirements on the minimization programs in circuit design and machine learning very different, a point that has not yet been sufficiently observed and appreciated. **This fact calls for the development of totally new approaches to synthesis,** and is a very positive opportunity for people working in the area of "Reed-Muller logic." Instead of adapting the algorithms created for circuit design, new algorithms should be created **from scratch**, and from the very beginning they should take into account the problem specifics. Moreover, since these algorithms run only in software, EXOR gates are as good as any other and there is no problem of its realization or speed.

Missing values, and especially noise, are still not adequately part of the circuit design world but are a reality in KDD and ML. It will be necessary to find solutions to these issues if we are to use logic synthesis tools in these fields.

We believe that machine learning will become a new and fruitful area for logic synthesis research, and an application territory for logic minimizers. There exist big challenges, but also great wins for successful programs. The first research results that appreciate this synergy and try to link the two worlds of the "machine learning community" and the "design automation community" already start to appear: new decision-diagram approaches were presented in 1994 by Ron Kohavi [10, 11], and Arlindo Oliveira [15]. It is our hope that the participants of this Workshop will also partake in this challenge, develop new theories and software, and will test them on the machine learning benchmarks. It is quite possible, that problems with an unusually high percent of don't cares will also occur in circuit design, when more sophisticated VHDL compilers start to appear.

References

[1] Y.S. Abu-Mostafa, *"Complexity in Information Theory",* Springer-Verlag, New York, 150pp, ISBN 0-387-96600-5, 1988.

[2] A.R. Barron, and R.L. Barron, "Statistical Learning Networks: A Unifying View", *Symposium on the Interface: Statistics and Computing Science,* 1988.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth, "Occam's Razor", *Information Processing Letters,* Oct. 1987, pp. 377-380.

[4] M.A. Breen, T.D. Ross, M.J. Noviskey, and M.L. Axtell, "Pattern Theoretic Image Restoration," *Proc. SPIE'93 Nonlinear Image Processing,* Intern. Soc. for Optical Engineering, January 1993.

[5] H.A. Curtis, *"A New Approach to the Design of Switching Circuits",* Princeton, N.J., Van Nostrand, 1962.

[6] J. A. Goldman, "Machine Learning: A Comparative Study of Pattern Theory and C4.5," Wright Laboratory, USAF, Technical Report, WL-TR-94-1102, WL/AART, WPAFB, OH 45433-6543, August 1994.

[7] J. A. Goldman and M. L. Axtell, "On Using Logic Synthesis for Supervised Classification Learning," *7th IEEE International Conference on Tools with Artificial Intelligence,* IEEE, November 1995.

[8] J.A. Goldman, T.D. Ross, and D.A. Gadd, "Pattern Theoretic Learning", *AAAI Spring Symposium Series on Systematic Methods of Scientific Discovery,* AAAI, March 1995.

[9] S. J. Hong, "R-MINI: A Heuristic Algorithm for Generating Minimal Rules from Examples", *Pacific Rim International Conference on Artificial Intelligence,* PRICAI, 1994.

[10] R. Kohavi, and B. Frasca, "Useful Feature Subsets and Rough Set Reducts", *Third International Workshop on Rough Sets and Soft Computing,* 1994.

[11] R. Kohavi, "Bottom-up Induction of Oblivious Read-Once Decision Diagrams," In *European Conference on Machine Learning,* 1994.

[12] J. Koza, *"Genetic Programming,"* MIT Press, 1992.

[13] T. Kozlowski, E.L. Dagless, J.M. Saul, "An Enhanced Algorithm for the Minimization of Exclusive-Or Sum-Of-Products for Incompletely Specified Functions", private information, 1995.

[14] M. Li and P. M. B. Vitányi, "Inductive Reasoning and Kolmogorov Complexity", *Journal of Computer and System Sciences,* Vol. 44, pp. 343-384, 1992.

[15] A.L. de Oliveira, *"Inductive Learning by Selection of Minimal Complexity Representations,"* Ph.D. Thesis, University of California at Berkeley, Dec. 1994.

[16] M. A. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Report,* Department of Electrical Engineering, Portland State University, 1995.

[17] J. R. Quinlan, *"C4.5: Programs for Machine Learning",* Morgan Kaufmann, 1993, Palo Alto, Ca.

[18] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[19] T.D. Ross, M.L. Axtell, M.J. Noviskey, "Logic Minimization as a Robust Pattern Finder", *International Workshop on Logic Synthesis,* May 1993.

[20] T.D. Ross, M.J. Noviskey, M.L. Axtell, D.A. Gadd, "Flash user's guide," *Technical report, Wright Laboratory,* USAF, WL/AART, WPAFB, OH 45433-6543, December 1993.

[21] T.D. Ross, J.A. Goldman, D.A. Gadd, M.J. Noviskey, and M.L. Axtell, "On the Decomposition of Real-Valued Functions", *"Third International Workshop on Post-Binary ULSI Systems in affiliation with the Twenty-Fourth International Symposium on Multiple-Valued Logic",* 1994.

[22] T.D. Ross, "Variable Partition Search for Function Decomposition," *Technical report, Wright Laboratory,* USAF, WL/AARA-3, WPAFB, OH 45433-6543, November 1994.

[23] T. Sasao, "Exmin2: A Simplification Algorithm for Exclusive-Or Sum-of-Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions", *IEEE Trans. on CAD.,* Vol. 12, No. 5, pp. 621-632, 1993.

[24] N. Song, and M.A. Perkowski, "Minimization of Exclusive Sum of Products Expressions for Multi-Output Multiple-Valued Input Switching Functions," *accepted to IEEE Trans. on CAD.*