2012 IEEE 27-th Convention of Electrical and Electronics Engineers in Israel

Machine-Learning-Based Circuit Synthesis

Lior Rokach
Ben Gurion University
Ben Gurion, Israel

Alexander Feldman
University College Cork
Cork, Ireland

Meir Kalech
Ben Gurion University
Ben Gurion, Israel

Gregory Provan
University College Cork
Cork, Ireland

***Abstract*—Multi-level logic synthesis is a problem of immense practical significance, and is a key to developing circuits that optimize a number of parameters, such as depth, energy dissipation, reliability, etc. The problem can be defined as the task of taking a collection of components from which one wants to synthesize a circuit that optimizes a particular objective function. This problem is computationally hard, and there are very few automated approaches for its solution. To solve this problem we propose an algorithm, called Circuit-Decomposition Engine (CDE), that is based on learning decision trees, and uses a greedy approach for function learning. We empirically demonstrate that CDE, when given a library of different component types, can learn the function of Disjunctive Normal Form (DNF) Boolean representations and synthesize circuit structure using the input library. We compare the structure of the synthesized circuits with that of well-known circuits using a range of circuit similarity metrics.**

I. Introduction

Logic (or Boolean Function) Synthesis is a well-known problem, and is a key to developing circuits that optimize a number of parameters, such as depth, energy dissipation, reliability, etc. The problem can be defined as the task of taking a collection of components from which one wants to synthesize a circuit that optimizes a particular objective function. This problem has been addressed since Roth [1].

Circuit synthesis is related to, but strictly more general than, Boolean minimization, on which there has been significant work (e.g., using the Quine-McCluskey method [2]). Rather than being given a function to optimize, we must jointly create the function and optimize it; in addition, we may want to address many other tasks in the synthesis process; such tasks include (1) optimize properties beyond just the number of gates that Boolean minimization addresses (e.g., circuit area, depth), (2) add components not present in the given function, and (3) design nested hierarchical structures in the device.

We aim to automate the process of generating circuits from component libraries. We propose a machine learning approach. Prior work has used genetic algorithms, which do not converge well [3], [4]. We adopt a decision tree approach, and in particular, an iterative greedy algorithm that adds the most efficient component in terms of model size. Our approach is not restricted by a pre-defined library of component types but uses a library that can dynamically grow, and thus keeps the model size small.

Our approach has several important applications.

- In *reverse engineering*, engineers can shorten the process of revere-engineering. For instance, automating this process could significantly reduce the time duration of unveiling key systems; e.g., it could emulate the reverse engineering of the ISCAS-85 benchmarks [5].
- In *model-based synthesis*, automating the process of Boolean function synthesis is needed for model-based systems. The existence of a model is a basic requirement for model-based systems. Unfortunately, in many cases such a model does not exist. We believe that automating the process will facilitate the design of model-based systems and will able to use techniques from model-based diagnosis [6], model-based prognostics [7] and model-based problem solving [8].
- In *model-based diagnosis*, this approach can take a system function and optimize its diagnostics properties, e.g., diagnosability, fault tolerance, failure probability, etc.

Our contributions are as follows. We propose a novel machine learning approach for Boolean function decomposition for the case of multi-level logic synthesis. We propose reverse engineering of Boolean formulas rather than addressing designing problems. We cope with multiple output functions rather than a single output. We implement a method that uses a library of different component types which can be dynamically increased with new types of components. Finally, our algorithm is empirically evaluated through various of circuits.

II. Related Work

This section compares our work to prior research in a range of different areas, including Boolean optimization, function learning, and synthesis.

The task of composing a model from components to achieve a goal function is known in the electrical and computer engineering literature as logic synthesis. Logic synthesis is a process for converting a high-level specification of circuit behavior, typically register transfer level (RTL), into a *design implementation*, which can be represented in terms of logic gates.

In general, there are two kinds of logic synthesis: two-level and multi-level. In two-level logic synthesis the goal is to represent a Boolean function by at most two gate levels between a primary input and a primary output. This can be achieved by representing the function as a DNF (in terms of the engineering literature: sum of products). Known methods for this task are Quine-McCluskey [2] to compute the exact prime implicants of the goal formula and heuristic

methods like ESPRESSO [9] which compute near-minimal prime implicants.

In multi-level logic synthesis there is no restriction on the number of gates between a primary input and a primary output. Actually, most circuits in real life are multilevel. Multiple levels of gates increase the complexity of logic synthesis dramatically, so exact solutions are not practical. There are many methods to reduce a logic formula to multi-level logic. Some of the methods use only primitive gates as AND, OR and NOT, like algebraic logic optimizations and Boolean logic optimizations [10].

Finally, Feldman et al. [11] present a new related problem to ours. They have implemented a General Redesign Engine (GRE), which uses model-based reasoning techniques and Boolean functional synthesis from component libraries, to automate redesign for combinational circuits. For the logic synthesis they consider fault tolerance optimization which reduces the probability of catastrophic failures. Feldman et al. do not implement a machine learning approach but a brute-force approach.

III. Concepts and Definitions

We start by presenting a set of definitions that are designed to facilitate the exposition of algorithms for automated reasoning.

Figure 1 shows an implementation of a full-adder, represented by the function $F(i_1, i_2, c_i) = (q \Leftrightarrow i_1 \wedge i_2) \wedge (p \Leftrightarrow i_1 \oplus i_2) \wedge (\Sigma \Leftrightarrow p \oplus c_i) \wedge (c_o \Leftrightarrow q \vee (p \wedge c_i))$.

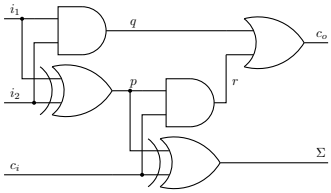

Fig. 1. This full-adder is used as a running example.

Definition 1 (Component): A component COMP, $\langle F$, IN, OUT$\rangle$, is specified using a Boolean function F over a set of variables Z, and input/output variables, IN, OUT $\in Z$.
Boolean functions that model components are often represented graphically, by using the same symbols as in a standard computer arithmetic schoolbook [12]. Figure 2 shows a component that implements a three-input AND gate by using two two-input ones. The Boolean function that is shown in Fig. 2 is $F(i_1, i_2, i_3) = (o \Leftrightarrow z \wedge i_3) \wedge (z \Leftrightarrow i_1 \wedge i_2)$ where IN $= \{i_1, i_2, i_3\}$, OUT $= \{o\}$, and z is an internal variable. We may omit specifying which variables are input and output, when that is clear from the context or from the common use (of an AND gate, for example).

Definition 2 (Component Library): A component library $\mathcal{L}$ is defined as a set of components.
Figure 3 shows a component library consisting of a half-adder, a two-input OR gate and a two-input NAND gate. In our

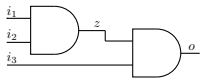

Fig. 2. A component that implements a three-input AND function by using two two-input AND gates

problem formulation, there are no restriction on the contents of the component library, i.e., it is a set of arbitrary multi-output Boolean functions. It is not necessary for a component library to contain a functionally complete subset of components (the two-input NAND gate in the component library shown in Fig. 3, for example, can be used to express any Boolean function, but that is not a requirement in our framework).

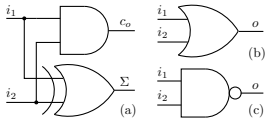

Fig. 3. A component library consisting of (a) a half-adder (HA), (b) a two-input OR gate (2-OR), and (c) a two-input NAND gate (2-NAND)

Definition 3 (System Description): A system description SD, $\langle \mathcal{L}, G \rangle$ is defined as a vertex-labeled and edge-labeled direct acyclic graph $G = \langle V, E \rangle$ such that $V = \{\text{PI} \cup \text{PO} \cup V'\}$ and if $v \in V'$, then $v \in \mathcal{L}$.
System description graphs contain a set of primary input vertices (PI), a set of primary output vertices (PO) and a vertex for each component. The graph edges are labeled with the names of the Boolean function variable names.

Figure 4 shows a system description of the full-adder circuit shown in Fig. 1, built from components drawn from the Fig. 3 library.

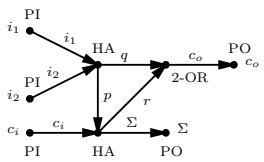

Fig. 4. System description of the full-adder circuit shown in Fig. 1

A system description SD is equivalent to exactly one Boolean function as shown in the following definition.

Definition 4 (Composition): Given a system description SD $= \langle \mathcal{L}, G \rangle$, $G = \langle V, E \rangle$, a composition $C(\text{SD})$ of SD is a Boolean function $(f_1 \circ \cdots \circ f_n)(x_1, \ldots, x_m)$ such that $n = |V| - |\text{PI}| - |\text{PO}|$ and for each $f_i \in \{f_1, \ldots, f_n\}$, there is an isomorphic function $f_i' \in \mathcal{L}$. The primary inputs and primary outputs of $f_1, \ldots, f_n$ are the respective edge labels in G and the internal variables in $f_1, \ldots, f_n$ are unique.
In the above definition the variables $\{x_1, \ldots, x_m\}$ are all internal variables, i.e., $\{x_1, \ldots, x_m\} = V \setminus \{\text{PI} \cup \text{PO}\}$.

The composition of the system description in Fig. 4, for example, is a Boolean function that is composed of two half-

adders, and a two-input OR gate. The i_1 and i_2 inputs of the half-adder in the component library shown in Fig. 3 are renamed to p and c_i for one of the instances.

Definition 5 (System Decomposition): Given a component library $\mathcal{L}$ and a Boolean function S, a system decomposition S^{-1} of S is a system description $\mathrm{SD} = \langle \mathcal{L}, G \rangle$ such that $S \equiv C(\mathrm{SD})$.

By equivalent function we mean that, since S and $C(\mathrm{SD})$ have the same primary inputs and primary outputs, a valuation $\phi(S) = 1$ iff $\phi(C(\mathrm{SD})) = 1$. Note that the problem of computing if two Boolean functions are equivalent is computationally very hard.

Computing decompositions of a given Boolean function is the main problem discussed in this paper. Certain decomposition are preferable, i.e., these minimizing some optimality criterion such as number of elementary functions (number of internal nodes in the resulting system description), a cost function, etc. In this paper, the optimality criterion minimizes the number of nodes in G.

IV. Circuit Decomposition Algorithm

Algorithm 1 shows the main system decomposition method of this paper. The basic idea of Alg. 1 is to greedily "carve-out" component instances, starting from some subset of the primary inputs and moving toward the primary output. Alg. 1 works on single-output Boolean functions only. The input function should be given in a Disjunctive Normal Form (DNF). The core of Alg. 1 is constructing multiple decision trees, one for each component instantiation candidate added to a reduced representation of the target Boolean function. A component instantiation is selected if it minimizes the depth of the decision tree.

Algorithm 1: Circuit Decomposition Engine (CDE)

```
   Input: S, a Boolean function in DNF
   Input: L, a component library
   Result: a system description
1  ⟨T, IN, OUT⟩ ← MakeTable(S);
2  repeat
3      foreach ⟨F, C_IN, C_OUT⟩ ∈ L do
4          foreach X ∈ SubsetsOfSize(IN, |C_IN|) do
5              Z ← F(X);
6              T' ← AddInternal(T, Z);
7              CT ← TreeInducer(T');
8              f* ← Evaluate(CT);
9              if f* < f then
10                 ⟨f*, Z*, CT*⟩ ← ⟨f, Z, CT⟩;
11         ⟨T, IN, OUT⟩ ← UpdateTable(T, Z*);
12 until Depth(CT*) > 2;
13 return MakeSystemDescription(CT*)
```

Table I shows the output of MakeTable (line 1) for the full-adder function shown in Fig. 1. Each column in T (in the running example T is initially constructed from Table I) is an attribute and this table is a partial specification of the system description and a full representation of the target Boolean function. Each attribute (column) represents a primary input, a primary output, or an internal variable. Note that each internal variable is also the output of a component and the name of this component can be specified in the name of the internal variable.

TABLE I
Truth table of the full-adder shown in Fig. 1

IN			OUT	
c_i	i_1	i_2	c_o	Σ
False	False	False	False	False
True	False	False	False	True
False	True	False	False	True
True	True	False	True	False
False	False	True	False	True
True	False	True	True	False
False	True	True	True	False
True	True	True	True	True

The main idea of Alg. 1 is to maintain a front of unused input or internal variables and to try all possible components from the component library. This front is initially constructed from all primary inputs contained in IN and later maintained in the same set of variables. Line 3 of Alg. 1 tries to use each component from the component library. Let the component chosen in line 3 has $k = |C_{\mathrm{IN}}|$ inputs. These k inputs are attempted to be connected to any k-subset of the variables in the set IN. These subsets are generated by the SubsetsOfSize auxiliary subroutine invoked in line 4.

Consider decomposing the function of the running example whose truth table is given in Table I. CDE first draws an inverter from the component library (the order is arbitrary). It will then try to use each of the IN variables of the full-adder as an input to this inverter. Line 5 of Alg. 1 computes the values at the output of the inverter. Line 6 of Alg. 1 adds the output of the inverter to the T truth table, storing the result in the temporary T' truth table as the choice of the inverter is not final. The first T' table for our running example is shown in Table II.

TABLE II
Truth table T' after connecting an inverter to c_i

c_i	$\neg c_i$	i_1	i_2	c_o	Σ
False	True	False	False	False	False
True	False	False	False	False	True
False	True	True	False	False	True
True	False	True	False	True	False
False	True	False	True	False	True
True	False	False	True	True	False
False	True	True	True	True	False
True	False	True	True	True	True

Each time a component is drawn from the component library and connected to unconnected input/internal variables, a decision tree is induced by the TREEINDUCER subroutine. A component is preferred if it leads to a binary decision tree with a smaller number of leaf nodes. Continuing our running example, the decision tree induces from the truth table T' shown in Table II is shown in Fig. 5.

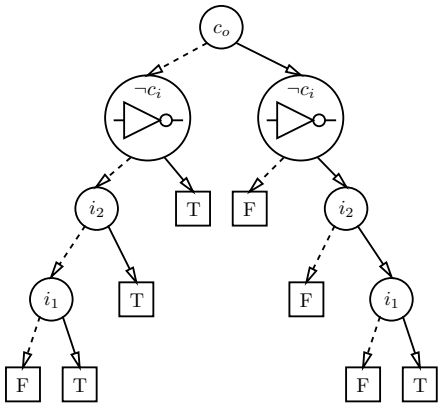

Fig. 5. Binary decision diagram induced from Table II

The tree shown in Fig. 5 has eight leaf-nodes and this is the value returned by the EVALUATE function in Alg. 1. After computing the quality of the tree shown in Fig. 5, CDE, tries all other possible components. For example, after a few attempts, CDE tries connecting a XOR gate to the primary inputs c_i and i_1. The resulting truth table is shown in Table III.

TABLE III
TRUTH TABLE T' AFTER CONNECTING AN XOR GATE TO c_i AND i_1

c_i	i_1	$c_i \oplus i_1$	i_2	c_o	Σ
False	**False**	**False**	**False**	**False**	**False**
True	**False**	**True**	**False**	**False**	**True**
False	**True**	**True**	**False**	**False**	**True**
True	**True**	**False**	**False**	**True**	**False**
False	**False**	**False**	**True**	**False**	**True**
True	**False**	**True**	**True**	**True**	**False**
False	**True**	**True**	**True**	**True**	**False**
True	**True**	**False**	**True**	**True**	**True**

Clearly, the quality of the second tree, shown in Fig. 6, and having 6 leaf-nodes is better than the first one (with 8 nodes), hence the XOR gate is preferred. The process continues until the resulting decision tree has only a root and leaf nodes, i.e., it is a stump tree. The resulting functional decomposition for our running example is shown in 7. The difference, from the original design comes from the fact that we run CDE separately for each primary output and then we combine the resulting Boolean functions. Despite that the design is very similar to the original and exhaustive checking verifies that the implemented Boolean function is equivalent to that of the original full-adder.

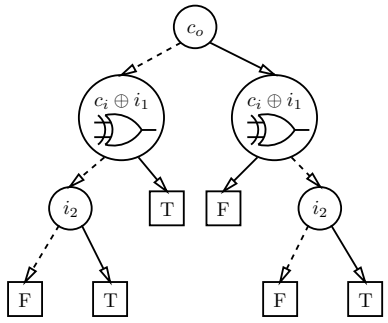

Fig. 6. Binary decision diagram induced from Table III

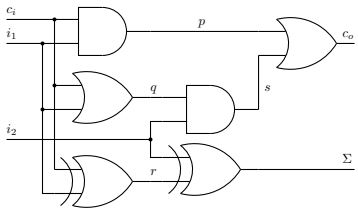

Fig. 7. Decomposition of the full-adder shown in Fig. 1

We next extend the results from running CDE on the full-adder to a benchmark of Boolean functions.

V. EXPERIMENTAL RESULTS

We have implemented CDE in Python using the Orange data mining and machine learning software suite [13] for inducing the binary decision trees. The implementation is straightforward and is a couple of hundred lines of code. We have run all our experiments on a recent Linux platform based on a 2.8 GHz Intel i7 CPU and equipped with 4 GB of RAM.

We evaluate the performance of CDE on a benchmark of combinational circuits that we introduce in this paper. The benchmark contains fifteen Boolean functions and is summarized in Table IV.

CDE computed decompositions for 13 out of 15 benchmark instances. The algorithm could not compute decompositions for MUL3 and 74181 within the preallocated time quota of 15 min. In all successful cases the returned Boolean functions were logically equivalent to the target function.

CDE produces interesting results in generating functions that do not only result in all metrics equal to 1 but also being equivalent (having equivalent system descriptions). This is the case for the instances HA, SUB1, PAR4 and PAR6.

The main results of CDE are summarized in Table V. The second and third column of Table V show the number of nodes and edges, respectively, of the system description returned by Alg. 1. The ratio of these sizes to the original graph sizes shown in Table **??** are given in the forth and fifth columns of Table V. We can see that these values are often close to 1 which means that the graphs are of similar size. The rightmost column of Table V shows the time in seconds it takes for CDE to decompose the target Boolean function.

TABLE IV
CIRCUIT DECOMPOSITION BENCHMARK

name	description	$\|V\|$	$\|E\|$	$\|\text{PI}\|$	$\|\text{PO}\|$
HA	half-adder	6	4	2	2
FA1	1-bit adder	10	8	3	2
FA2	2-bit adder	15	9	5	3
FA4	4-bit adder	23	15	9	5
SUB1	1-bit subtractor	12	10	3	2
MUX4	4-bit multiplexer	16	15	6	1
DEMUX4	2-to-4 demultiplexer	15	11	3	4
MUL2	2-bit multiplier	16	12	4	4
MUL3	3-bit multiplier	32	27	6	6
PAR4	4-bit parity checker	8	7	4	1
PAR6	6-bit parity checker	12	11	6	1
74182	4-bit CLA	33	28	9	5
74L85	4-bit comparator	47	44	11	3
74283	4-bit adder	50	45	9	5
74181	4-bit ALU	87	79	14	8

TABLE V
DECOMPOSED BOOLEAN FUNCTIONS

name	$\|V'\|$	$\|E'\|$	$\|V\|/\|V'\|$	$\|E\|/\|E'\|$	time [s]
HA	6	4	1	1	0.59
FA1	11	9	0.91	0	0.64
FA2	23	20	0.65	0	3.17
FA4	49	36	0.47	0	119.09
SUB1	12	10	1	1	0.66
MUX4	19	18	0.84	0	11.91
DEMUX4	17	13	0.88	0	1.23
MUL2	19	15	0.84	0	1.32
MUL3	-	-	-	-	-
PAR4	8	7	1	1	0.35
PAR6	12	11	1	1	0.92
74182	52	47	0.63	0	36.36
74L85	90	87	0.52	0	532.20
74283	108	103	0.46	0	135.17
74181	-	-	-	-	-

VI. CONCLUSIONS

In this paper we have formulated the problem of Boolean logic synthesis (or circuit decomposition) from generic component libraries. This problem is computationally very hard and, depending on the functional completeness of the component library, may have no solution. We have designed and implemented the CDE algorithm that is based on machine learning, i.e., it greedily "carves-out" component instances from the target function (i.e., the function to be synthesized) until some termination criterion is met. Our design is based on the idea to use a generic version of the Shannon decomposition (which is related to building decision trees) as a heuristic in solving the more difficult generalization of decomposing functions in terms of *arbitrary* component (function) libraries. The last feature sets apart our work from Boolean function minimization such as minimal covers, optimal decision diagrams, etc.

To verify the validity of our method we propose a benchmark of combinational circuits. In addition, we have identified a set of basic graph similarity metrics which we use for validating our algorithm. In some cases, CDE reverses a Boolean function that has been constructed manually. In the rest of the cases CDE computes function decompositions that have number and types of component similar to the original, manually created Boolean functions. Our approach is clearly many orders of magnitude faster than the trivial brute-force approach that terminates only for Boolean functions of a very few variables.

This work is introductory in a sense that, to the best of our knowledge, there is no in-depth *algorithmic* analysis of the problem of logic synthesis. As a future work we plan (1) to improve the CDE algorithm, (2) to formulate more problems related to logic synthesis, (3) to identify and implement more metrics for evaluating the performance of algorithms. To improve the CDE algorithm we intend to implement guided back-jumping, exploration of hierarchy and active learning of decomposed sub-functions.

Given the simplicity of our approach, it shows great promise given that there are many optimizations that can be introduced. Such optimizations include introducing better objective functions, applying heuristics to the simple greedy method, and learning sub-function component models that can be quickly substituted during the decomposition process.

REFERENCES

[1] J. Roth, "Algebraic topological methods for the synthesis of switching systems I," *Trans. Amer. Math. Soc*, vol. 88, no. 2, pp. 301–326, 1958.

[2] E. J. McCluskey, "Minimization of Boolean functions," *The Bell System Technical Journal*, vol. 35, no. 5, pp. 1417–1444, 1956.

[3] A. H. Aguirre, B. P. Buckles, and C. A. Coello, "A genetic programming approach to logic function synthesis by means of multiplexers," in *Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, ser. EH'99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 46–.

[4] A. H. Aguirre, E. C. G. Equihua, and C. A. C. Coello, "Synthesis of Boolean functions using information theory," in *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, ser. ICES'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 218–227.

[5] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test*, vol. 16, no. 3, pp. 72–80, 1999.

[6] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing diagnoses and systems," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 197–222, 1992.

[7] J. Luo, K. R. Pattipati, L. Qiao, and S. Chigusa, "Model-based prognostic techniques applied to a suspension system," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 38, no. 5, pp. 1156–1168, 2008.

[8] P. Struss, "Model-based problem solving," in *Handbook of Knowledge Representation*, 2008, pp. 395–465.

[9] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA, USA: Kluwer Academic Publishers, 1984.

[10] M. Fujita, Y. Matsunaga, and M. Ciesielski, *Multi-level logic optimization*. Norwell, MA, USA: Kluwer Academic Publishers, 2002, pp. 29–63.

[11] A. Feldman, G. Provan, J. de Kleer, L. Kuhn, and A. van Gemund, "Automated redesign with the General Redesign Engine," in *Proceedings of the Eighth Symposium on Abstraction, Reformulation, and Approximation (SARA'09), Lake Arrowhead, California, US*, July 2009.

[12] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford University Press, Inc., 2009.

[13] T. Curk, J. Demšar, Q. Xu, G. Leban, U. Petrovič, I. Bratko, G. Shaulsky, and B. Zupan, "Microarray data mining with visual programming," *Bioinformatics*, vol. 21, pp. 396–398, February 2005.