Standardizing Boolean Transforms

Alan Mishchenko, Yukio Miyasaka, John Wawrzynek, Robert Brayton
University of California, Berkeley
alanmi@berkeley.edu

Abstract— **This paper reviews several flavors of Boolean resubstitution and shows that many known implementations of logic rewriting, circuit restructuring, and Boolean decomposition can be expressed using resubstitution. It proposes a unified representation of individual instances of the resubstitution problem and several solvers with different runtime/quality tradeoffs. The solvers are implemented in ABC and ongoing experiments show that they cover a broad range of use-cases in logic synthesis and formal verification.**

I. Introduction

In the age when both runtime and quality of results produced by hardware design flows pose significant challenges, it is helpful to rethink Boolean transforms, classify and prioritize them, and come up with a standard way to solve them. In practice, it means developing solvers with different runtime/quality tradeoffs accepting problems in a unified representation format, similar to how a variety of SAT solvers can solve Boolean satisfiability problems specified using the CNF representation in DIMACS format [1]. This is particularly helpful in the age of multicores and GPUs, because having a standard representation and a uniform interface helps deploy diverse workloads in the parallel setting.

When talking about Boolean transforms in this paper, we consider individual instances of Boolean computations, such as decomposing a small Boolean function into a minimum set of two-input gates, restructuring a logic cone to have fewer nodes, or re-expressing a node in the lookup-table network to be independent of timing critical fanins.

Each of these instances is solved by one call to a Boolean transform solver (such as a decomposition package or a rewriting engine). Typically a series of similar transforms is applied to the design representation, resulting in a cumulative improvement of the target function. This is similar to how the same SAT solver can be used to solve a series of related SAT problems represented in CNF. In some cases, the problems are completely independent while in other cases they are related through shared logic or timing.

When characterizing a Boolean transform, we may ask the following questions:

- is it applicable to completely-specified functions, incompletely-specified functions (functions with don't-cares), and Boolean relations
- is it applicable to single-output or multiple-output Boolean functions/relations, resulting in logic cones with logic shared across different outputs
- how a function is represented (truth table, SOP, BDD, AIG, CNF, etc)
- can the transform take advantage of side-divisors, that is, available nodes potentially useful as building blocks for the target functions (in the simple case of decomposing a single-output function without surrounding logic, the only divisors are the primary inputs)
- what is the goal of the transform (minimizing area, delay, switching activity, etc)
- what is the runtime/memory/data-transfer budget
- is it possible to err on some inputs (as in machine learning or in approximate synthesis)

In this paper, we distinguish and characterize three types of Boolean resubstitution: exact synthesis, don't-care-based resubstitution, and circuit rewriting, which have different runtime/quality tradeoffs, discussed in detail in Section IV. We also propose resubstiution solvers to efficiently solve individual instances of resubstitution problems of the above three types, uniformly represented using the proposed format.

In the rest of the paper, we give some background (Section II), present the simulation-guided implementation of Boolean methods that largely motivated this work (Section III), describe the three flavors of resubstitution while emphasizing their tradeoffs and use-cases (Section IV), propose a unified input/output representation format (Section V), and describe the current implementation in ABC (Section VI), followed by conclusions and open problems (Section VII).

II. Background

III. Simulation-guided implementation of Boolean transforms

The efficiency of bit-parallel simulation (when a large number of simulation patterns is packed into 64-bit machine words and efficiently simulated through the circuit using bitwise operators) and the use of simulation signatures (when nodes' functions can be exactly or approximately represented using their simulation information) leads to a new and efficient way of implementing Boolean transforms, called simulation-guided paradigm [2].

We briefly discuss this paradigm here, because it motivates the proposed standardization of the Boolean transforms and is likely to benefit from it.

The reason why simulation-guided computations are important is because of their fast runtime and high scalability, compared to more traditional Boolean computations based on SOPs, BDDs, and SAT. For small functions (up to 16 inputs), simulation-based computations can be made accurate because the complete truth table of each node can be represented and manipulated using simulation signatures. For larger functions (and even for functions of 10-16 inputs, for which complete truth tables can be used) a speedup can be obtained by approximating the functionality of the nodes using approximate simulation signatures.

To this end, whenever a circuit size exceeds 10 inputs, the nodes' functions can be represented using simulation signatures created using, say, 256 randomly generated input patterns. This approximate representation can be used to compute functional properties and perform logic restructuring. In the end, efficient SAT-based equivalence checking can be performed [3]. If the equivalence check passes, a given instance of a Boolean transform is correct and the computation moves to the next node; if, however, the check fails, we collect a randomized set of counter-examples failing the check and append them to simulation signatures.

For convenience, we may choose to collect 64 different counter-examples because this is equivalent to appending one machine word to the simulation signatures of the nodes. Only this one additional machine word of simulation information need to be recomputed before the next attempt to perform the Boolean transform, while all the existing simulation information remains valid.

The advantage of the approximate representation using simulation signatures is that it is compact and thus one iteration of a Boolean transform can be orders of magnitude faster than when it is implemented using an accurate representation (such as a truth table, BDD, or SAT). However, in practice, several refinement iterations may be needed. Overall, the simulation-guided computations tend to run faster and be more scalable than their accurate versions. Symbolic sampling [4] is a similar approach that has been proposed for approximating nodes' functions using BDDs.

IV. Resubstitution solvers and their use-cases

A. *Exact synthesis*

Exact synthesis [5]–[9] is the least scalable method, which can handle the general case of multi-output Boolean relations and synthesize provably smallest circuits. The method can be implemented using SAT solving or functional enumeration. It has been used to synthesize minimum circuits for any 5-input function but it fails to finish on many 6-input functions. The fact that this method is applicable multi-output Boolean relations makes it the only truly multi-output synthesis method implemented in ABC where all other methods are single-output in the sense that they can modify the function of only one node while assuming that the function of other nodes remains unchanged.

B. *Don't-care based resubstitution*

Don't-care based resubstitution [10]–[12] works for single-output incompletely-specified functions with side-divisors representing nodes already existing in the circuit. It is typically performed in two phases: first, a minimal subset of side-divisors is computed resulting in the support of a resubstitution function; second, the functional representation of the target function is computed. This method is not the most fast and scalable, this is why its use is often limited to post-mapping resynthesis and high-effort area optimization.

C. *Logic rewriting*

Logic rewriting [13], [14] is the most scalable logic synthesis method implemented in ABC. It is applicable to four-input functions with side-divisors found in the surrounding logic. It has been extended to work with five-input cuts [15] and has been further generalized by orchestrating several transforms to be performed interchangeably [16]. The method makes local changes in the contest of structural hashing [17], which employs a global hash table storing all available nodes. Because it is very fast, it can be iterated, resulting in non-local changes. The high speed and scalability of this method makes it applicable to large designs.

V. Representation format

This section discusses the unified representation format used by all Boolean resubstitution engines presented in this paper. This is an input/output format because it describes the representation of both the input functions/relations and the output circuits produced by resubstitution.

After a careful consideration and experiments with different format, we adopted a variation of the Espresso PLA format [18] as a way to represent input data in this project.

The PLA format represents Boolean functions by listing the number of their inputs (`.i <num>`) and outputs (`.o <num>`), followed by a table of input/output combinations. At the end of the table, there is an optional final line (`.e`).

According to the PLA format description, "each position in the input plane corresponds to an input variable where a 0 implies the corresponding input literal appears complemented in the product term, a 1 implies the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term" [18].

For the output part, we adopt `type fdr` described as follows: "with `type fdr`, for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, a - means this product term belongs to the DC-set, and a implies this product term has no meaning for the value of this function" [18].

Incompletely-specified functions can be represented in this format by using symbol "-" in the output column or by not listing an input minterm or cube in the PLA description, in which case it is assumed to be a don't-care. Boolean relations can be represented by allowing the same input combination to appear several times with different output values.

Fig. 1 shows three examples of the input data representation in the PLA format: (a) is the Boolean function of a full-adder with three inputs, two outputs, and eight input combinations; (b) is the Boolean function of a full-adder modified to allow for any output value when the input is 111; (c) is the Boolean function of a full-adder modified to be a Boolean relation, which allows for the outputs to be both 00 and 11 when the input is 111.

```
.i 3
.o 2
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
.e
```
(a)

```
.i 3
.o 2
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 --
.e
```
(b)

```
.i 3
.o 2
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 00
111 11
.e
```
(c)

Fig. 1. Examples of the input data representation.

As stated above, side-divisors are functions that may already exist in the circuit and thus could be used to express the target function along with the primary inputs. If side-divisors are present, their number is added to that of the primary inputs in the `.i` line and their values are listed in the input/output table after input and before outputs.

The synthesized circuit can be represented as an AIG [19] composed of the constant node, inputs (I) (including primary inputs and side-divisors if any), internal nodes (A), and outputs (O). In this AIG, the object with ID 0 is reserved for the constant 0 function, followed by I inputs with IDs ranging from 1 to I, followed by A internal nodes with IDs ranging from $1 + I$ to $I + A$, followed by outputs with IDs ranging from $1 + I + A$ to $I + A + O$. The total number of objects in this AIG is $N = 1 + I + A + O$.

The synthesized circuit is represented as a list of pairs of integer numbers. The constant node and the inputs are not explicitly listed because it is assumed that their number is known. The internal nodes are represented as pairs of fanin literals, and the outputs are represented as pairs of duplicated literals of their drivers.

For example, function $F = ab + cd$ can be represented as an AIG with four objects: three internal nodes and one output. Since each of the four objects is a pair of integer numbers, the resulting representation is an array of eight numbers: 2 4 6 8 11 13 15 15. The first pair of numbers (2 and 4) represents node ab, because the IDs of inputs a and b are 1 and 2, respectively, and the positive literals of these inputs are $1 \times 2 + 0 = 2$ and $2 \times 2 + 0 = 4$. Similarly, the second pair (6 and 8) represents node cd. The third pair (11 and 13) represents negative literals of the first two internal nodes whose IDs are 5 and 6. Finally, the output is a duplicated negative literal (15 and 15) of the last internal nodes whose ID is 7 because $2 \times 7 + 1 = 15$.

If a synthesized circuit is associated with an input PLA file, there is no ambiguity regarding the number of inputs. However, if the synthesized circuit in the proposed format is found in a stand-alone file, the number of outputs (O) can be found by looking for pairs of identical numbers, while the number of internal nodes (A) is the number of remaining pairs, and the number of inputs (I) is the last node ID pointed to by an output minus the number of internal nodes. Alternatively, the output file may contain the first line listing the number of inputs (I), internal nodes (A) and outputs (O). For other needs, such as interfacing with the external tools, the synthesized circuit can be dumped in the standard AIGER format [19].

VI. Current implementation

All three engines are implemented in ABC and are available (or will be available shortly) as separate commands applicable to the specification in the proposed format.

A. Exact synthesis

Exact SAT-based synthesis is implemented as commands `twoexact` and `lutexact`, which take an input PLA file on the command line. Alternatively, if the input is a single-output completely-specified function, the function can be represented as a truth table on the command line. Currently the exact synthesis is not integrated into any optimization engine in ABC but there are external integrations [20].

B. Don't-care based resubstitution

Don't-care based resubstitution is implemented as a command `resub_core`, which takes an input PLA file on the command line. Currently this flavor of resubstitution is used in commands `resub` [21] and `mfs2` [22].

C. Logic rewriting

Circuit rewriting [13] was implemented in the first public release of ABC in 2005 and has become a de facto standard for fast logic synthesis. The original command `rewrite` was later superseded by `drw`. It was found experimentally that iterating AIG rewriting often results in smaller AIGs than employing other slow and complicated Boolean transforms. However, ultimate area reductions cannot be achieved by circuit rewriting alone and requires the use of high-effort Boolean methods, such as the AIG-based transduction [23] (command `&transtoch`).

VII. Conclusions and open problems

The goal of this tutorial is to give the reader an understanding of Boolean resubstitution, its three flavors offering different runtime/quality tradeoffs, its place in a logic synthesis flow, and how it is implemented in ABC using simulation signatures. The reader may also be able to run ABC commands performing these and other transforms and get started with writing their code to customize and generalize available implementations.

Several open research problems are listed below.

Problem 1: Adapting circuit rewriting methods to work with very large circuits, in particular, AIGs containing more than 10 million nodes. Although such massive AIGs can be currently handled in ABC, the runtime is often prohibitive. The challenge is to obtain a wide spectrum of runtime/quality tradeoffs, which allows some users to spend 1 minute to reduce the AIG size by, say, 10%, while other users may choose to spend 10 minutes and reduce the AIG size by 15%, while yet another user may choose to run the tool overnight and reduce the AIG size by 20%. The latter overnight option may already exist in ABC, but the former 1-minute option is currently missing. It is true that large AIGs can partitioned into smaller ones, but many non-scalabilities still remain even when handling 100K-node partitions, while using smaller partitions can limit optimization quality due to inability to optimize across the partition boundaries.

Problem 2: Developing AIG optimization methods to be applied before technology mapping, which are aware of the LUT size or the technology library used for the mapping. This is because the current AIG optimization methods are only aware of the AIG node count and level count, which often results in a substantial structural bias, which shows when a well-optimized AIG is mapped into LUTs or into a standard-cell library, resulting in larger area and/or delay than when the unoptimized AIG is mapped into the same library. Recent work has shown that standard-cell mapping can be improved if tech-independent AIG-based synthesis is customized to minimize factored-form literal counts [24].

Problem 3: Developing placement aware logic synthesis methods, which explore logic restructuring in the context of placement. This may require developing novel placement proxy cost functions, which allow for estimating the results of placement during synthesis and using the estimations in the inner loop of logic restructuring. Recent work shows the potential of this method, indicating that area can be reduced by as much as 25% if logic structures are chosen while evaluating the results using placement [25].

Acknowledgement

This research was supported in part by the SRC Contract 3173.001 “Standardizing Boolean transforms to improve quality and runtime of CAD tools”, the NSA grant “Novel methods for synthesis and verification in cryptanalytic applications”, and donations from AMD, Siemens, and Synopsys.

References

[1] *DIMACS CNF format*, `https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html`.

[2] S.-Y. Lee, H. Riener, A. Mishchenko, R. Brayton, and G. D. Micheli, “A simulation-guided paradigm for logic synthesis and verification,” *IEEE Trans. CAD*, vol. 41, no. 8, pp. 2573–2586, 2022.

[3] H.-T. Zhang, J.-H. R. Jiang, L. Amaru, A. Mishchenko, and R. Brayton, “Deep integration of circuit simulator and SAT solver,” in *Proc. DAC*, 2021.

[4] Y.-T. Lin, J.-H. R. Jiang, and V. N. Kravets, “Symbolic uniform sampling with XOR circuits,” in *Proc. ICCAD*, 2020.

[5] N. Een, *Practical SAT: A tutorial on applied satisfiability solving*, Invited presenation at FMCAD, 2007.

[6] D. E. Knuth, “Boolean evaluation,” in *The Art of Computer Programming*, vol. 4A, Boston, MA, USA: Pearson Education, Inc., 2015, ch. 7.1.2.

[7] W. Haaswijk, M. Soeken, A. Mishchenko, and G. D. Micheli, “SAT-based exact synthesis: Encodings, topology families, and parallelism,” *IEEE Trans. CAD*, vol. 39, no. 4, pp. 871–884, 2020.

[8] F.-X. Reichl, F. Slivovsky, and S. Szeider, “Circuit minimization with QBF-based exact synthesis,” in *Proc. AAAI Conf. on Artificial Intelligence*, vol. 37, 2023, pp. 4087–4094.

[9] S.-Y. Lee, J.-H. R. Jiang, A. Mishchenko, and R. Brayton, “Enumeration of minimum fanout-free circuit structures,” in *Proc. IWLS*, 2019.

[10] M. A. Perkowski, M. Marek-Sadowska, L. Józwiak, *et al.*, “Decomposition of multiple-valued relations,” in *Proc. ISMVL*, 1997.

[11] V. N. Kravets and K. A. Sakallah, “M32: A constructive multilevel logic synthesis system,” in *Proc. DAC*, 1998.

[12] A. Mishchenko and R. Brayton, “SAT-based complete don’t-care computation for network optimization,” in *Proc. DATE*, 2005.

[13] P. Bjesse and A. Boralv, “DAG-aware circuit compression for formal verification,” in *Proc. ICCAD*, 2004.

[14] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis,” in *Proc. DAC*, 2006.

[15] N. Li and E. Dubrova, “AIG rewriting using 5-input cuts,” in *Proc. ICCD*, 2011.

[16] Y. Li, M. Liu, M. Ren, A. Mishchenko, and C. Yu, “DAG-aware synthesis orchestration,” *IEEE Trans. CAD*, to appear, 2024.

[17] M. Ganay and A. Kuehlmann, “On-the-fly compression of logical circuits,” in *Proc. IWLS*, 2000.

[18] *Input file format for Espresso (PLA-file)*, `https://user.engineering.uiowa.edu/~switchin/OldSwitching/espresso.5.html`.

[19] *AIGER format*, `https://fmv.jku.at/aiger/index.html`.

[20] F. Reichl, F. Slivovsky, and S. Szeider, “Circuit minimization with exact synthesis: From QBF back to SAT,” in *Proc. IWLS*, 2023.

[21] A. Mishchenko and R. K. Brayton, “Scalable logic synthesis using a simple circuit structure,” in *Proc. IWLS*, 2006.

[22] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, “Scalable don’t care based logic optimization and resynthesis,” in *Proc. FPGA*, 2009.

[23] Y. Miyasaka, “Transduction method for AIG minimization,” in *Proc. ASP-DAC*, 2024.

[24] A. T. Calvino, A. Mishchenko, H. Schmit, E. Mahintorabi, and a. X. X. Giovanni De Micheli, “Improving standard-cell design flow using factored form optimization,” in *Proc. DAC*, 2023.

[25] R. Roy, J. Raiman, and S. Godil, *Designing arithmetic circuits with deep reinforcement learning*, `https://developer.nvidia.com/blog/designing-arithmetic-circuits-with-deep-reinforcement-learning`.