Verification after Synthesis

Alan Mishchenko   Robert Brayton

Department of EECS
University of California, Berkeley
Berkeley, CA 94720

{alanmi, brayton}@eecs.berkeley.edu

Abstract

The disconnect between sequential synthesis and sequential verification has two consequences: (1) *strong sequential optimizations are not used during synthesis because they are hard to verify, and* (2) *verification, if performed in isolation from synthesis, borders on becoming intractable. This paper develops a scalable methodology for checking sequential equivalence of the original network and the network derived by integrated sequential optimization* [15]. *The method uses an "optimization history" describing the sequence of logic transformations carried out during synthesis. A format for representing optimization history is proposed and motivated. A preliminary implementation of the proposed methodology is described and experimentally compared with an efficient general-purpose equivalence checker that does not rely on information from synthesis.*

1 Introduction

With the rapid increase of the number and sizes of designs synthesized by the present-day CAD tools, the role of formal verification become more important. However, the high complexity of verification problems leads to prohibitive runtime of the tools. Although significant progress has been made in verification based on BDDs, SAT, and AIGs (And-Inverter Graphs), these results do not readily transfer to large sequential circuits. The capacity and scalability of state-of-the-art methods in sequential verification fall far short of industrial requirements.

The complexity of sequential verification stems from the intractability of recovering intermediate logic structures seen during the optimization process. A recent study [10] has shown that, counter to common belief, verifying the results of a set of combinational optimizations followed by retiming and another set of combinational optimizations is already as complex as the most general sequential verification problem, i.e. PSPACE complete.

Given this difficulty, sequential verification algorithms have fallen into the following categories:

- complex but unscalable *complete* methods [9]
- simulation and probabilistic *incomplete* methods [6]
- methods for specialized circuits (e.g. pipelines)
- methods for some types of synthesis (e.g. retiming) [18]
- verification using information from synthesis.

In this paper, we advocate the last category because scalable, complete, and general methods are necessary to advance both synthesis and verification. We propose a methodology for scalable sequential equivalence checking, capable of efficiently verifying very large designs using a compact *optimization history* supplied by a synthesis system.

An *optimization history* is a succinct record of the following types of logic transformations: (1) *Combinational restructuring* extracts a logic cone or a window and replaces it by a functionally equivalent one (up to complementation of outputs). (2) *Retiming* changes the positions of one or more latches without changing the logic structure. (3) *General sequential resynthesis* replaces a window of combinational logic and latches by a sequentially equivalent window. The third transformation subsumes the first two and is only needed to record the changes if complete sequential flexibility is used [14]. Re-encoding and resynthesis using subsets of unreachable states falls into the third category.

In a larger setting, our method covers verification after aggressive sequential optimization, which involves gate-level resynthesis and retiming. If a "golden model" of the design is given at the system- or RT-level, the user should use another verification tool to verify the initial gate-level specification against the "golden model". Our tool should be used to verify the result of final sequential optimization against the initial gate-level specification.

One argument against the proposed approach is that the intermediate equivalence checking problems may be too large to be verified easily, unless some special constraints are imposed on synthesis, for example, toggle equivalence of logic blocks [7]. A response to this is that scalable verification is an inevitable consequence of scalable synthesis. In a robust logic synthesis flow, individual synthesis operations are made local and resource-aware. This is achieved by performing each operation on a node using a transitive fanin cut or a containing window, whose size is determined by resource limits. Consequently, if the network changes as a result of a synthesis operation, the scope of change does not include the whole network but only the logic between the node and the leaves of the cut, or the logic inside a window containing the node. This observation motivated our work, which does not assume any artificial constraints on logic synthesis.

The rest of the paper is organized as follows. Section 2 discusses background on AIGs and equivalence checking.

Section 3 reviews "integrated" sequential optimization performing fast delay-optimal synthesis/mapping/retiming and motivating the need for scalable verification. Section 3 introduces the principles of recording optimization history. Section 4 presents some implementation details of the specialized AIG manager used to record optimization traces. Section 5 summarizes the experiments. Section 6 concludes the paper and outlines future work.

2 Background

Definition. A *Boolean function* is a mapping from n-dimensional ($n \geq 0$) Boolean space into a 1-dimensional one: $\{0,1\}^n \rightarrow \{0,1\}$.

Definition. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs).

Definition. The output of a node may be an input to other nodes called its *fanouts*. The inputs of a node are called its *fanins*. If there is a path from node *A* to *B*, then *A* is in the *transitive fanin* of *B* and *B* in the *transitive fanout* of *A*. The transitive fanin of *B*, *TFI*(*B*), includes node *B* and the nodes in its transitive fanin, including the PIs. The *transitive fanout* of *B*, *TFO*(*B*), includes node *B* and the nodes in its transitive fanout, including the POs. An *edge* in a Boolean network is a connection between two nodes, which are in the fanin/fanout relationship. The fanin/fanout of an edge is the fanin/fanout node of the connected pair of nodes.

Definition. *AND-INV graph* (AIG) is a Boolean network composed of two-input AND-gates and inverters.

Definition. The *size* of an AIG is the number of its AND nodes. The number of logic levels is the number of AND-gates in a longest path from any primary input to any primary output.

Inverters are ignored when counting nodes and logic levels. In the software implementation, inverters are represented as flipped node pointers, similar to the complemented edges in a BDD.

Definition. *Structural hashing* (*strashing*) of an AIG is a transformation intended to quickly reduce the AIG size by partially canonicizing the AIG structure. When a new AND node is added, a hash-table is checked for a node with the same two fanins. Although the resulting AIG is not functionally canonical, it does not contain isomorphic subgraphs.

Structural hashing was introduced originally for netlists of arbitrary gates in early IBM CAD tools [5] and was used in combinational equivalence checking [11].

Definition. A *cut C* of node *n* is a set of nodes of the network, called *leaves*, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called *root* of cut *C*. A *trivial cut* of the node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. All *K*-feasible cuts are computed using network flow [4] or cut enumeration [19]. Single good cuts are computed using heuristic methods [17].

Definition. The *local* function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, which form a cut of *n*. The *global function* of an AIG node is its function in terms of the PIs of the network.

Definition. *Exhaustive simulation* is a practical way of checking equivalence of two logic cones whose size does not exceed 16 inputs. The simulation is performed bitwise through the cone of 2^k different input patterns, where *k* is the number of cut leaves. Another way of looking at exhaustive simulation is that it computes the truth-table of the root of the cut in terms of the elementary truth-tables set at the leaves.

Definition. *Combinational equivalence* of two designs is the equivalence of the global functions of the POs and flip-flop inputs.

Definition. *Sequential equivalence* of two designs with initial states is understood as follows. The equivalent designs start in their respective initial states and, in all time frames, for identical input sequences, produce idential output sequences [9].

The concepts "verification" and "equivalence checking" are used interchangeably in this paper.

3 Integrated sequential optimization

Integrated sequential optimization is a synthesis method that combines logic synthesis, technology mapping, and retiming into a single integrated flow [15].

To illustrate the use of integration, consider logic synthesis and technology mapping. If these steps are not integrated, the network is first optimized by technology-independent logic synthesis. The resulting network is then given to delay-optimal technology mapping resulting in the best delay for the *given* logic structure. However, the decisions made during tech-independent synthesis are independent from technology mapping and so logic structures leading to a good mapping are often lost. Contrary to this, an integrated approach explores several search spaces at once. In the above case of logic synthesis and technology mapping, it finds the best mapping over all available logic structures [12][3].

The integrated optimization can be summarized as follows:

- It is applicable to both standard cell and FPGA designs.
- It guarantees strong optimality in terms of the minimum clock period, computed using a load-independent delay model, for all logic structures derived by a synthesis flow, for all technology mappings, and for all retimings.
- The global minimization of the clock period is achieved by a series of fast local transformations,

allowing the approach to scale to designs with millions of gates.

- An efficient implementation is based on sequential AIGs [1]. Experiments performed in [15] using industrial circuits show an average reduction in the clock period of 25%, compared to traditional mapping without retiming, and by 20%, compared to traditional mapping followed by retiming applied as a post-processing step.
- The elementary steps of the integrated optimization are easy to record and reproduce. The recorded history of optimization applied can be used to develop a scalable approach to verification, as shown in the present paper.

For a detailed description of the algorithms used in the integrated optimization and its experimental evaluation, refer to [15].

4 Optimization history

An *optimization history* is a record of the following three types of logic transformations: (a) combinational restructuring; (b) retiming; and (c) general sequential resynthesis. In this paper, we consider the first two types, since they cover the full scope of the current integrated sequential optimization system [15].

The optimization history is recorded in a specialized AIG manager described in the following section. To detect the relevant transformation steps that need to be verified, we take the final synthesized and mapped netlist and determine how library gates or LUTs map into the AIG nodes of the subject graph. In addition, we determine transformations used to create each AIG node. Unique IDs of the AIG nodes in the manager are used to retrieve the transformations applied, which, in turn, contain references to the previous AIG nodes. This process of backward unrolling leads to the original network through a set of connecting links.

In essence, the verification system is a proof checker that takes the optimization history and replays it while proving all intermediate steps. If the proof steps link the initial and final network, then by transitivity, global sequential equivalence holds.

This approach is similar to other proof methods. One example is verifying unsatisfiability proofs returned by a SAT solver [20]. An unsatisfiability proof is recorded as a sequence of resolutions. It is useful to observe that, although every learned clause is annotated with its resolution proof in the SAT solver, not all resolutions are needed for the final proof. Similarly, the final proof of sequential equivalence can involve only a subset of optimization steps performed during the optimization flow.

Another similarity is with efficient methods of combinational equivalence checking [11][16]. The identification of intermediate equivalences in the compared circuits helps enormously in proving equivalence of their outputs. A similar approach for sequential circuits [6] proves sequential equivalence of intermediate nodes using induction. The success of these methods stemmed from the fact that synthesis tools typically leave some parts of the original circuit intact, which leads to the existence of intermediate equivalent points.

We observe that the transformation history recorded in the AIG manager effectively contains *complete* information about the intermediate equivalences, not just between nodes in the original and final networks. Sequential verification using this additional information is much easier than verification using only *partial* information that combinational or sequential equivalence checkers can derive when only the initial and the final networks are given.

5 Implementation details

This section describes the implementation details of the *history AIG manager* used to store the optimization history.

Each two-input AND node in the history manager has a unique integer ID, composed of the ID of the network it came from and its integer ID within that network. The network IDs form a chronological sequence and can be considered as time-stamps of the nodes.

The history manager is initialized by adding the original network, which is considered the “golden model”. This network gets network ID 1. Next, AIG-based logic synthesis is performed, e.g [17]. During synthesis, new networks are created, one after another, and assigned consecutive network ID numbers. Whenever an AIG node is created in a network, a corresponding node is added to the history manager. Thus, an AIG structure in the history manager is isomorphic to an AIG structure of the series of the optimized networks.

An additional API of the history manager allows for cross-linking the parallel AIG structures in the manager. When a new node is created in the current network, which is known to be functionally equivalent to a node in the previous network, the corresponding history nodes are “linked”. Such linkages are similar to *choices* used in lossless synthesis [3], except that they are not considered proved but represent proof obligations to be verified in the same order during the verification phase. The corresponding PO pairs are always linked when optimization of an intermediate network is finished. This way, the equivalence of POs will be proved in due time, after the intermediate equivalences are proved.

Note that, when AIG nodes derived from different networks are linked in the history manager, there is no need to store their phase relationship because phases of all nodes in the history manager can be derived by simulating all nodes with a reference pattern (e.g. the pattern composed of all zeros). For each pair of linked nodes, this simulation will indicate the phase relationship between the nodes; if two nodes have identical simulation bits, they have the same phase; otherwise, opposite phases.

When the final network is to be verified against the original, the history manager considers adjacent pairs of intermediate networks (according to their network IDs). For each such pair of intermediate networks, the manager proves all the intermediate associated points cross-linked during the synthesis process.

For these intermediate proofs, it is enough to consider only relatively small logic cones corresponding to the cuts or windows that were used during synthesis. In fact, this is the principle reason for the scalability of this method of verification. If an equivalence proof at one of the intermediate points fails, the verification process is aborted. If all intermediate links verify successfully, including the pairs of corresponding POs, the history manager reports that the networks are equivalent.

Retiming is the only sequential optimization step implemented so far in the integrated flow [15]. A retiming of the network is represented in the history manager as a sequence of elementary retiming moves, each of which transports one latch backwards over an AIG node or two latches forward. The elementary retiming moves are scheduled as events in the same sequence as the node linking steps. Each such move receives a unique number composed of the network ID and the internal retiming move ID, which can be traced down to a pair of AIG nodes linked before the move and a pair to be linked after the move. During verification this retiming move is performed after verifying the former link and before verifying the latter.

6 Experimental results

The presented algorithms are implemented in a public-domain logic synthesis and verification system ABC [2].

The current implementation has the following limitations:

- Only combinational verification using optimization history has been implemented so far. The integrated sequential optimization [15] uses retiming, so currently we can’t verify the full process. We plan to add this capability soon.
- Only the cut-based AIG rewriting commands (e.g. *strash, balance, rewrite, refactor*) are supported in the optimization history. Window-based synthesis operations, such as redundancy removal [17] and optimization using don’t-cares [13], remain to be implemented in our synthesis tool.
- Intermediate combinational cones used for rewriting are limited to 16 inputs. This size is sufficient for the current logic synthesis flow. Cones are proved by exhaustive simulation. Larger cones, if they arise in the future, can be handled by a general equivalence checker [16].

Experiments were performed using several large IWLS 2005 benchmarks [8]. Runtimes are measured on a 1.6GHz IBM ThinkPad with 1Gb RAM.

Table 1 reports the sizes of the AIGs used and memory requirements for both synthesis and verification. The first column lists the IWLS benchmark names. Column “original” shows the number of AIG nodes after structural hashing. Column “resyn2” shows the number of AIG nodes after script “resyn2”, which performs 10 AIG rewriting passes to optimize area (the number of AIG nodes) under delay constraints. Delay is measured as the number of levels of the AIG. The delay constraint is set to be the minimum number of levels derived by algebraic tree-balancing of the AIG structure.

Column “total” shows the number of AIG nodes in the optimization history. Column “equiv” shows the number of internal nodes that, according to the optimization history, have the identical function to some other node in a previous network. This is also the number of non-trivial intermediate equivalence checking problems solved during verification.

The last two columns in Table 1, “synch” and “verif”, compare the memory requirements, in megabytes, for the synthesis (the peak memory) and verification (the total memory for storing the optimization history for the initial and 10 intermediate networks generated by running script “resyn2”). The verification memory is estimated as 20 bytes/node for all nodes stored in the optimization history (the number of all nodes is listed in column “total”). With such relatively small amount of memory needed to store the optimization history for 10 intermediate networks, the total memory requirements of scalable verification are less than the peak memory used by synthesis. However, processing large designs with many intermediate optimization steps may motivate storing large optimization histories on disk rather than in RAM.

Table 1. AIG size and memory requirements.

Benchmark	Number of AIG nodes				Memory, Mb	
name	original	resyn2	total	equiv	synth	verif
aes_core	21213	19880	222426	22459	16.77	4.24
des_perf	78299	69289	807241	68903	25.03	15.40
ethernet	19729	13020	162893	12920	17.70	3.11
pci_bridge32	22784	17842	205213	13348	14.78	3.91
usb_funct	15873	13214	151095	11194	13.22	2.88
vga_lcd	126711	90904	1074894	88724	32.19	20.50
wb_conmax	47853	40406	468268	36419	22.90	8.93
Average ratio	**1.00**	**0.81**	**9.41**	**0.76**	**1.00**	**0.36**

Table 2. Runtime comparison.

Benchmark	Synthesis		Verification	
name	strashing	resyn2	general	scalable
aes_core	0.24	4.99	6.92	1.80
des_perf	1.04	31.20	39.58	4.60
ethernet	0.19	2.06	3.65	0.70
pci_bridge32	0.26	2.64	5.27	1.10
usb_funct	0.17	2.28	1.87	0.50
vga_lcd	1.86	41.24	148.55	7.80
wb_conmax	0.40	10.13	13.18	1.60
Average ratio	**0.06**	**1.00**	**1.74**	**0.26**

Table 2 compares the runtimes of using a general combinational verification [16] (column “general”) versus those of the proposed scalable verification (column “scalable”). For reference, these runtimes are compared with the runtimes of synthesis (column “resyn2”) and structural hashing (column “strashing”).

The conclusion from Table 2 is that the scalable verification is faster than a state-of-the-art robust, resource-aware implementation of general-purpose combinational equivalence checking [16]. The gap between these two runtimes will be more substantial after sequential synthesis because general-purpose sequential equivalence checking methods are much less powerful than combinational ones.

The runtime of the proposed verification is also less than that of synthesis using a fast resource-aware script “resyn2” because, although it follows the same steps as synthesis, it does not involve search. The runtime of the proposed verification is composed of checking correctness of the recorded steps, which in the current implementation is done efficiently using exhaustive simulation.

7 Conclusions and future work

This paper exploits a synergy of synthesis and verification to develop a scalable approach to sequential equivalence checking using information from a synthesis flow. We believe that sequential verification will benefit from this approach because intractable verification problems arising after sequential synthesis are partitioned into a sequence of simple intermediate steps, which can be solved easily using currently available methods. We also believe that synthesis will be improved in the long run because the availability of scalable verification will ease the adoption of stronger sequential synthesis methods.

The proposed approach to verification can be used to produce “certificates of correctness” for the synthesized designs. These certificates can be verified by a third-party certificate-checker to make sure that the design is indeed correct. Fast and simple certificate checkers of this type can replace general-purpose sequential equivalence checkers in many cases when verification is performed after a recorded logic synthesis flow.

This paper has merely outlined a general approach to verification after synthesis and discussed an initial limited implementation. Much future work remains. This includes extending the scope of optimizations, for which the optimization history can be recorded, and including sequential optimizations as well as combinational synthesis with observability don’t-cares (such as redundancy removal). Our long-term goal is to co-develop two interacting public-domain systems: for integrated sequential synthesis and for scalable verification after synthesis.

Acknowledgement

This research was supported in part by SRC contract 1361.001, NSF contract CCR-0312676, and by the California Micro program with our industrial sponsors, Intel, Magma, Synplicity, and Altera.

References

[1] J. Baumgartner and A. Kuehlmann, “Min-area retiming on flexible circuit structures”, *Proc. ICCAD’01*, pp. 176-182.

[2] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*, *Release 60306*. http://www.eecs.berkeley.edu/~alanmi/abc/

[3] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD’05*. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf.

[4] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”, *IEEE Trans. CAD*, Vol. 13(1), January 1994, pp. 1-12.

[5] A. Darringer, W. H. Joyner, Jr., C. L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM J. of Research and Development*, Vol. 25(4), 1981, pp 272-280.

[6] C. A. J. van Eijk, “Sequential equivalence checking based on structural similarities,” *IEEE TCAD*, vol. 19(7), pp. 814–819, 2000.

[7] E. Goldberg, “On equivalence checking and logic synthesis of circuits with a common specification”. *Proc. GLSVLSI ‘05*, Chicago, pp.102-107. http://eigold.tripod.com/papers/glsvlsi-2005.pdf

[8] *IWLS ‘05 Benchmarks*. http://iwls.org/iwls2005/benchmarks.html

[9] J.-H. R. Jiang and R. K. Brayton. “On the verification of sequential equivalence”, *IEEE Trans. CAD*, vol. 22(6), June 2003, pp. 686-697. http://www.eecs.berkeley.edu/~brayton/publications/2003/tcad03_sec.pdf

[10] J.-H. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective", *Proc. IWLS '05*, pp. 8-15. http://www.eecs.berkeley.edu/~brayton/publications/2005/iwls05_r&r.pdf

[11] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification”, *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[12] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, “Logic decomposition during technology mapping,” *IEEE TCAD*, 1997,.

[13] A. Mishchenko and R. K. Brayton. “SAT-based complete don’t-care computation for network optimization”. *Proc. DATE '05,* pp. 418-423. http://www.eecs.berkeley.edu/~alanmi/publications/2005/date05_satdc.pdf

[14] A. Mishchenko, R. Brayton, J.-H. R. Jiang, T. Villa, and N. Yevtushenko, “Efficient solution of language equations using partitioned representations”, *Proc. DATE '05*, pp. 412-417. http://www.eecs.berkeley.edu/~alanmi/publications/2005/date05_lang.pdf

[15] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *Submitted to DAC '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_int.pdf.

[16] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Eén, “Improvements to combinational equivalence checking”, *Submitted to IWLS ’06*. ttp://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_cec.pdf

[17] A. Mishchenko and R. Brayton, “Scalable logic synthesis using a simple circuit structure”, *Submitted to IWLS ’06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_fls.pdf

[18] M. Mneimneh and K. Sakallah, “REVERSE: Efficient sequential verification for retiming”, *Proc. IWLS ’03*, pp. 133-139.

[19] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs”, *Proc. FPGA ’98*, pp. 35-42.

[20] L. Zhang and S. Malik, “Validating SAT solvers using an independent resolution-based checker”, *Proc. DATE ’03*, pp. 880-885.