Incremental Sequential Equivalence Checking and Subgraph Isomorphism

Sayak Ray **Alan Mishchenko** **Robert Brayton**

Department of EECS, University of California, Berkeley

{sayak, alanmi, brayton}@eecs.berkeley.edu

Abstract

A method for finding large isomorphic subgraphs in two similar circuits is proposed, and its application to sequential equivalence checking (SEC) is discussed.

SEC ensures correctness of two designs. Among other things, efficient SEC is important for wider adoption of innovative sequential synthesis (SS) methods, which offer substantial reductions in delay, area, power and flip-flop counts, compared to the traditional combinational methods. In practice, some forms of SS, such as clock-gating, modify only a small portion of the design, but because of cyclic dependencies in sequential logic, current SEC solutions have to be applied to the entire designs. This leads to the inability to prove equivalence for important problems.

A promising solution to SEC for such situations is to identify large isomorphic subgraphs of the two circuits. Then SEC can be proved by compositional verification. Preliminary experiments show this method can be used effectively for some difficult industrial SEC problems.

The method for finding large isomorphic subgraphs is of interest in general and can be used, for example, in incremental physical design. The method is fast and effective.

1 Introduction

In many cases, a circuit is modified in only a few places leaving the rest of the circuit untouched. In these cases, large isomorphic subgraphs exist in the before and after circuits. If this isomorphism can be reliably and efficiently found, it can be used to enhance the solution of several problems. One is sequential equivalence checking (SEC), where the isomorphism can be used to create a set of highly-likely candidate equivalent nodes which can be proved more easily.

SEC aims to prove functional equivalence between two sequential circuits. It requires that the two designs produce identical sequences at the primary outputs (POs) for the same primary input (PI) sequence, starting from the initial states. In general, SEC is PSPACE-complete [13] but a recent evaluation of a general-case SEC engine [25] shows that about 15% of industrial designs cannot be verified after a typical sequential synthesis (SS) flow involving sequential and combinational transforms, even when the SEC engine is given generous resource limits. Improving the speed and capacity of SEC can benefit SS by expanding its scope and applicability, resulting in savings in delay, area, power, and flip-flop counts, compared to the traditional combinational methods.

In many practical scenarios, SS involves a small portion of the design, or is applied incrementally. For example, in the case of clock-gating [11] for power reduction, gating signals are selected and a few logic nodes are added to drive the clock-gates. In such cases, practical experience suggests that this creates very difficult sequential equivalence problems. A similar situation occurs when a designer manually modifies the design. Even though the scope of manual changes is typically small, it is important to formally verify them.

This motivates the problem of how to efficiently detect and exploit existing structural similarity between two designs. When the second design is created by SS, one way is to do this is to record a synthesis history. This can be useful to validate the sequence of changes by reproducing them in a different tool, or to create an inductive invariant to be verified by an independent inductive prover, as shown in [26]. In general, recording synthesis history is no acceptable in many situations.

The present paper develops a method for finding large isomorphic subgraphs in two circuits. It takes two designs whose primary inputs and outputs can be matched by name. However, no assumptions are made about the number and functionality of the registers. The approach detects similarities by using "extended" simulation and local structural signatures to identify possible matching nodes in the two circuits. This is iterated using until no more candidate matches can be found. Finally, the matching is checked for being an isomorphism and trimmed if the isomorphism check fails.

For application to SEC, the regions of difference in the two circuits can be sequentially verified against each other. If successful, SEC for the whole design succeeds by compositional arguments, since the environments of the difference regions are structural and functionally equivalent.

The contributions of this paper are a) a method for finding large isomorphic subgraphs in two similar circuits, and b) a procedure and supporting theorems which use this for SEC.

The paper is organized as follows. Section 2 describes some background. Section 3 describes previous related work. Section 4 presents the method for finding large isomorphic subgraphs. Section 5 presents a method for applying this to the SEC problem and possibly relaxing the isomorphism when the different subgraphs are not sequentially equivalent. Section 6 reports some preliminary experimental results. Section 7 concludes the paper and outlines possible future work.

2 Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network, circuit, and design are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. A network is assumed to be sequential unless it is stated otherwise. The terms memory element, flop-flop, and register are used interchangeably.

A node *n* has zero or more *fanins,* i.e. nodes that are driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs in combinational optimization and mapping, sometimes called PPIs and PPOs for pseudo inputs and outputs.

A combinational *And-Inverter Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. *Sequential AIGs* add registers to the logic structure of combinational AIGs. The registers are technology-independent D-flops with one input and one output that are assumed to belong to the same clock domain. We assume that the registers of two designs to be verified have a fixed binary initial state. If a register has an unknown or a don't-care initial state, it can be transformed to have 0-initial state by adding a new PI and a MUX controlled by a reset register that produces 0 in the first frame and 1 afterwards.

3 Previous Related Work in SEC

Research in SS and SEC started by defining notions of sequential equivalence [30][27]. Several methods have been proposed for SEC without initial state information [32][15]. However, as shown in [2], in an industrial setting it is often sufficient to consider designs whose initial states are known.

The prevalent SEC methods in the early days were based on BDD-based reachability [7][33]. Later, *k*-step induction [10] was shown to be more scalable than BDD-based reachability because it does not require computing the exact set of reachable states. A drawback of induction is that it is not guaranteed to prove or disprove the equivalence.

A number of improvements have been proposed to enhance induction as a proof method and enable SEC for realistic industrial designs. The main contribution is the use of SAT pioneered in [5] and its extension to *k*-step induction with dynamic addition of uniqueness constraints presented in [6] and [9]. Another improvement, which boosted the scalability of SAT-based sequential synthesis and SEC is speculative reduction [28][29].

In spite of the recent successes of SAT-based SEC engines [19][2][25], the methods are still limited due to the inherent complexity of SEC [13], non-inductiveness of SEC in many practical instances, and prohibitive runtime for large designs. This motivates research on alternative proof techniques, such as interpolation [20] and abstraction [21].

Another direction of research is efficient SEC methods for specialized types of SS. For example, [14] proposed a scalable method for SEC after retiming and combinational synthesis; [25] advocates the use of detection and merging of sequentially-equivalent signals as a form of SS, which preserves state-encoding and leads to substantial reductions while being scalable and scalable-verifiable under certain conditions; [26] advocates the use of a synthesis history to guarantee inductiveness. A drawback of this is that implementation of SS has to be modified to enable recording history. In contrast, the present work rediscovers the history (in some sense) while comparing the original and final designs. Structural differences are detected and, if these differences are small, the SEC problem is reduced to a smaller problem of proving SEC on the differences.

4 Finding Large Isomorphic Subgraphs

4.1 Overview

We consider the following problem: Given two designs in the form of sequential AIGs, their structures are compared and areas of structural isomorphism are identified. The designs are assumed to have PIs and POs, which can be matched exactly by name, but the designs may have different numbers of registers and no name matching is assumed for them. A final check is made that the matching computed is an isomorphism between the two subgraphs identified, and if not, the matching is trimmed until it is.

More specifically, suppose that two sequential AIGs, N_1 and N_2, are given where the PIs and POs are matched by name. The objective is to find two isomorphic subgraphs, C_1 and C_2, one each from N_1 and N_2. The subgraphs need not be connected, however, the isomorphism must include the matching of the PIs and POs of N_1 and N_2. Ideally, the isomorphism should include a maximum number of nodes, but that is not a requirement.

We employ two complementary methods for finding large isomorphic subgraphs, one based on extended simulation, and the other based on structural signatures. In the experiments discussed in Section 6, we use simulation first to identify candidate matches and structural similarities to resolve ambiguities.

The computation starts at the PIs and POs, which are already matched by name. For each node, we compute a sequential simulation signature and identify pairs of nodes, one in N_1 and one in N_2, which have the same simulation signature, and each of which is unique in their respective AIGs. These are then matched, and forward and backward propagations are done for each matched node, e.g. if for a given matched pair, their fanins (or fanouts) have different signatures and can be paired uniquely, the fanins (or fanouts) are matched. This backward (forward) propagation may end for two reasons. First, the propagation may stop because there is a fanin (fanout) signature in one circuit which has no corresponding fanin (fanout) signature in the other AIG. This is a case where we know that the isomorphism does not extend. Second, two fanins (fanouts) of a node may have the same simulation signature and it is not clear how to do the matching. At this point we use structural signatures to disambiguate the result if possible. If this succeeds, the propagation can continue.

To propagate sequential simulation "deeper" into the network, we use *extended simulation*, by inserting a PI at some points where two nodes are already matched. This can introduce some errors which can be filtered out using structural information.

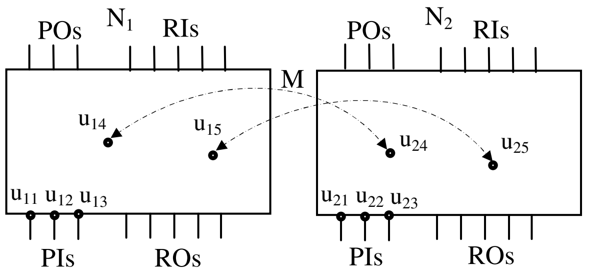
4.2 Extended Sequential Simulation

Traditional sequential simulation is typically performed by propagating thousand patterns for hundreds of cycles. In each cycle, the nodes are visited in a topological order and all patterns are evaluated in a bit-parallel fashion. Primary inputs are assigned random values in each clock-cycle, while the register outputs are set to the initial state in the first cycle and thereafter are equal to the register inputs in the previous cycle. The result is, for each node, a simulation signature, which contains its value for each simulated pattern in each cycle. If a node has a unique simulation signature (that is, there is no other node in its circuit having the same signature), the node is *uniquely identified*.

Experiments show that the percentage of nodes uniquely identified by this traditional sequential simulation in large industrial designs is typically less than 10%. This phenomenon does not depend on how long simulation continues and is due to poor controllability of nodes located far from the PIs.

We extend traditional sequential simulation in order to uniquely identify nearly all nodes in each copy of the design and match them across the two designs. The notion of extended sequential simulation is illustrated in Figure 1. It is iterative, where at first unique identification is performed using traditional sequential simulation; only the PIs are assumed to have random values. In subsequent iterations, the set of nodes treated as PIs is extended to include new PIs replacing some nodes that have been uniquely identified and matched already. Thus for each pair of matched nodes chosen, a new PI is introduced, which directly drives the fanouts of the pair.

This process continues until extended simulation saturates and cannot uniquely distinguish additional nodes. This can happen when all nodes are matched, or when one of the two designs under verification contains structurally identical components, perhaps due to duplication of logic. In such cases, several workarounds can be devised to prevent aliasing inside each design, and thus increase the number of pairs of nodes uniquely identified. Note that extended simulation can introduce errors because although the paired nodes are simulated to be equal, their value is dictated by a free input, which may not be a value that can be related with the other inputs.



The u-nodes include PIs and other uniquely identified nodes. All are treated as primary inputs in a new round of random simulation. Simulation uses the same random pattern in both N_1 and N_2.

Figure 1. Extended sequential simulation illustrated.

4.3 Structural Similarities

Detecting the regions of structural isomorphism is precisely the problem of maximum subgraph isomorphism which is NP-hard [32]. Although there are various special cases of this problem, which can be solved in polynomial time, their non-linear complexity makes them impractical for large. Some methods find isomorphic subgraphs by computing vertex invariants – these methods are very efficient, but do not guarantee a correct solution in general.

A vertex invariant method computes an integer *i(v)* for each vertex *v* of the graph and declares two vertices *u* and *v* to correspond under isomorphism if *i(u) = i(v)*. In our case, a simulation signature of each AIG node can serve as a vertex invariant. Another vertex invariant is its structural signature: for each node *n* in the AIG, we define sig(*n*) as a tuple $<m_1, m_2, m_3, m_4, m_5, m_6>$ where

1. m_1 = number of fanouts of *n*,
2. m_2 = number of complemented fanouts of *n,*
3. m_3 = number of fanouts of fanin-0 of node *n,*
4. m_4 = number of complimented fanouts of fanin-0 of node *n,*
5. m_5 = number of fanouts of fanin-1 of node *n*,
6. m_6 = number of complemented fanouts of fanin-1 of node *n*.

For a given AIG, we compute sig(*n*) for all internal nodes (i.e. nodes that are not primary inputs, primary outputs, latch inputs or latch outputs). For a node $n_1 \in N_1$ and $n_2 \in N_2$, if sig(n_1) = sig(n_2), then it is likely that n_1 and n_2 are the corresponding nodes. Other kinds of structural signatures may be defined depending on specific requirements.

Like other vertex invariants, two arbitrary nodes can have the same signature, so we cannot rely on these alone to determine isomorphic subgraphs. Instead, we can use a structural signature synergistically with a simulation signature by using one for local disambiguation when the other is used as the global vertex invariant. A good technique should at least be able to identify a 100% node correspondence. In experiments with Altera benchmarks, indeed we detected 100% matches. One advantage of our proposed structural signature is that it can be computed by traversing an AIG once in any order, spending constant time per node and then can be stored in a hash table for further use, consuming time and memory that is linear in the size of the AIG.

4.4 Combining Simulation and Structure

Although we found that the extended simulation can identify a majority of node correspondences, it fails to detect node correspondence in cases where a circuit has two copies of the same sub-circuit. A simulation signature is ignorant of structure and can miss obvious node correspondences easily disambiguated with a structural signature. We have adopted the following approach.

When we detect a possible match (but not uniquely identified) based on the simulation signature, we check the structural signatures of the candidate nodes for matching also. We divide the structural signature into two parts, a fanin part and a fanout part. Two nodes having the same simulation signature in a single circuit are now treated as different if their fanin structural signatures or fanout structural signatures differ. This increases the number of uniquely identified nodes during each iteration and hence improves the quality of the matching. This division into fanin and fanout parts is particularly important for the nodes on the boundary of the region of change.

4.5 Final Trimming of the Matching

After we have found a matching, we have a partial function *F* which maps nodes of N_1 into nodes N_2 as follows: $F(n_1) = n'$ where $n \in N_1$ and $n' \in N_2$ means our algorithm matches *n* to n'; $F(n) =$ NULL means we have not found any match for *n*. Similarly, we have another partial function *G* which maps nodes of N_2 into nodes of N_1 in an analogous way. Now, for $S_1 \subseteq N_1$ and $S_2 \subseteq N_2$ where $|S_1| = |S_2|$, *F* and *G* will be isomorphic mappings if the following properties hold:

1. for all nodes $n \in S_1$, $F(n) \neq$ NULL,
2. for all nodes $m \in S_2$, $G(m) \neq$ NULL,
3. $F(n) = m$ implies *F*(fanin0(*n*)) = fanin0(*m*) and *F*(fanin1(*n*)) = fanin1(*m*),
4. *G*(*m*) = *n* implies *G*(fanin0(*m*)) = fanin0(*n*) and *G*(fanin1(*m*)) = fanin1(*n*)
5. *F*(*n*) = *m* iff G(*m*) = *n*, i.e. *G* is an inverse mapping of *F* .

Note that this final isomorphism test is independent of how we found the mappings.

Our approximate matching algorithm does not necessarily give an isomorphic matching on the whole graph of N_1 and N_2. We need to find a pair of subsets S_1 and S_2 by removing some nodes from N_1 and N_2 (for which *F* and *G* are defined, respectively) which will satisfy the above conditions. This is achieved by iteratively removing nodes which do not meet the above conditions. Iteration stops when we reach a fixed point. At that point, all nodes in the resulting sets meet the above conditions and hence form a pair of isomorphic subgraphs.

In our experiments, we found that this fixed point can be reached typically within three iterations and that only a few nodes are dropped during this trimming. The sizes of the resulting isomorphic subgraphs depend on the size and quality of the initial candidate matching.

5 Application to SEC

5.1 Overview

The identified isomorphic subgraphs are sequential circuits with matched PIs and POs and by Theorem 1 below, they are sequentially equivalent. Given the isomorphism, the remaining sets of nodes form two other sequential circuits, D_1 and D_2. The POs of D_1 are PIs C_1 or POs of N_1, and similarly for D_2. Also the PIs of D_1 are POs C_1 or PIs of N_1, and similarly for D_2.

At this point we use Theorem 2 to prove that $N_1 \equiv N_2$ by proving that $D_1 \equiv D_2$. However, it may be that the isomorphism identified is too large and that D_1 is not equivalent to D_2. We then refine the process by enlarging D_1 and D_2 to include more matched nodes of C_1 and C_2. The need for this is discussed below.

5.2 Theorems and Definitions

In Section 4, we discussed a method for finding large isomorphic subgraphs in two sequential AIGs. We now make this notion of isomorphic AIGs more precise and give theorems on how this can be applied to SEC.

Definition: Two AIGs are structurally isomorphic if there exists a 1-1 mapping M from one graph to the other, such that if two nodes are matched, then

1. they must be of the same type i.e. register, AND node, inverter node, PI, or PO, **and**
2. their inputs are matched, **and**
3. two matched registers have the same initial value.

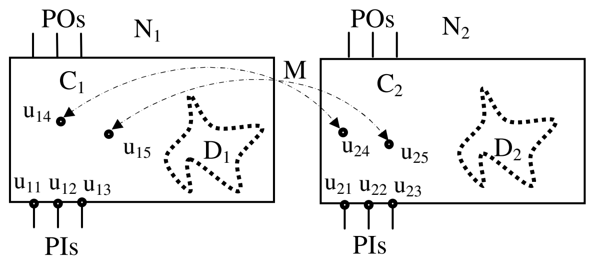
Theorem 1: Two isomorphic AIGS are sequentially equivalent.

Proof: As we simulate the two AIGs from their initial states, the circuit can be viewed as a combinational one with a topological order. Starting at the PIs and register outputs, the simulated values are the same in both graphs at these nodes. Inductively assume all nodes of topological order k have equal values. For the two nodes at order k+1, they have match matched input with order less that k+1, and hence have equal values. Since the two nodes have the same type, their outputs values are equal given equal inputs. Thus all matched nodes simulate to the same values and since this holds for the PIs, the two graphs are sequentially equivalent. **QED**.

Theorem 2: Assume that subgraphs, C_1 and C_2 of N_1 and N_2 respectively, are isomorphic AIGs whose PIs can be uniquely matched by name. Let D_1 and D_2 be the remaining sets of nodes where their PIs and POs, which are internal to N_1 and N_2, are matched by corresponding POs and PIs in C_1 and C_2. Then $(D_1 \equiv D_2) \Rightarrow (N_1 \equiv N_2)$.

Proof: Since $C_1 \equiv C_2$, then any unconstrained sequence of their PIs from the initial state produces the same sequence at their POs. Similarly, for $D_1 \equiv D_2$. Let PI_X denote the PIs of AIG X. Since $PI_{N_1} \subseteq PI_{C_1} \cup PI_{D_1}$, $PI_{N_2} \subseteq PI_{C_2} \cup PI_{D_2}$, $PO_{N_1} \subseteq PO_{C_1} \cup PO_{D_1}$, $PO_{N_2} \subseteq PO_{C_2} \cup PO_{D_2}$, and $N_1 = D_1 \text{ e } C_1$, $N_1 = D_1 \text{ e } C_1$, then any sequence of PIs to N_1 and N_2 produce the same sequences of POs of N_1 and N_2. **QED.**

Theorem 2 is useful to develop a compositional approach to SEC as described in the next section.



After isomorphic matching on N_1 and N_2, common parts, C_1 and C_2, and differing parts, D_1 and D_2, are isolated as separate sequential AIGs. Theorem 1 states: $C_1 \equiv C_2$. Theorem 2 states: $(D_1 \equiv D_2) \Rightarrow (N_1 \equiv N_2)$.

Figure 2. Theorem 2 illustrated.

5.3 Compositional SEC and Refinement

The compositional approach is based on Theorem 2 and illustrated in Figure 2. It starts by identifying large isomorphic subgraphs in designs, N_1 and N_2. Each of these is partitioned into $D_k \cup C_k$, and SEC is reduced to proving $D_1 \equiv D_2$.

A problem occurs if this does not hold, because of the following. Suppose a small set of nodes S is resynthesized using SDCs and ODCs, extracted from a window W, where $S \subset W$. Suppose the set S is changed structurally by synthesis, but nodes in $W\backslash S$ are not changed structurally. However, they may be changed functionally because S may be changed functionally by the use of don't cares. We are only guaranteed by synthesis that the outputs of W are unchanged functionally.

Thus, if the isomorphism has included nodes in $W\backslash S$, $D_1 \equiv D_2$ may not hold. What needs to be done is to eliminate some nodes from the isomorphism to the point that it does not include any nodes in $TFO(S) \cap W$. Such nodes can be identified (even though we do not know what are S and W) by sequential simulation. Any pair of isomorphically matched nodes that do not have equal sequential simulation signatures must be deleted from C_1 and C_2 and added to D_1 and D_2. Note that the method described in Section 4 may allow this in the isomorphism it finds, because it uses extended simulation.

It is acceptable to be imprecise about how to change C_1 and C_2. For example, one can just make a guess and decrease C_1 and C_2 appropriately. If $D_1 \equiv D_2$ can be proved, then $N_1 \equiv N_2$ is proved. A heuristic would be to move one layer of nodes at the PIs of C_1 and C_2 to D_1 and D_2, try again to prove $D_1 \equiv D_2$, and continue until it is proved or the D_k get too large.

If designs N_1 and N_2 have substantial structural similarity, it is likely that C_1 and C_2 are relatively large, while D_1 and D_2 are relatively small. In this case, running SEC on D_1 and D_2 is substantially easier than running SEC on N_1 and N_2.

It is possible that N_1 and N_2 are not equivalent. In this case, a counterexample disproving D_1 and D_2 might be extended to disprove N_1 and N_2. Currently, we do not have a method for this.

The refinement approach, in which the boundary between C_1 and D_1 is gradually shifted to include more logic of D_1 into C_1 (and similarly for C_2 and D_2), in the limit may just lead to proving the equivalence of N_1 and N_2, when C_1 and C_2 become empty.

6 Experimental Results

The structural and simulation signature computations were implemented in ABC [4]. The experiments were run on a Windows laptop with Intel Core2 Duo CPU and 2 GB RAM.

Experimental results reported have the goal of finding structural isomorphism in large designs. To generate interesting examples, we applied synthesis (a mixture of rewriting, retiming, signal correspondence) to several already synthesized benchmarks. These examples do not necessarily have a small set of nodes that have been changed and currently we do not have a way of measuring exactly which nodes have been changed. However, since the examples were already heavily synthesized, the additional synthesis probably did not make significant changes. In a few cases, we created examples by manually inserting functional errors.

The results of our experiments are given in Table 1. Each row represents an experiment on a pair of circuits *C*, *C*' where *C*' is synthesized from *C*. The numbers (*x,y*)(*x',y'*) in the second column have the following meaning: circuit *C* has *x* flip-flops and *y* AND gates, circuit *C*' has *x*' flip-flops and *y*' AND gates. The columns 'initial FF match' and 'initial AND match' show the number of flip-flops and AND nodes that are matched by our algorithm based on extended simulation and structural signatures. The column 'isomorphic AND match' shows the number of AND nodes that conform to isomorphism after the trimming operation.

Note that in the majority of cases, only a relatively few nodes are eliminated in the isomorphic test and trimming step. For example, in the first example (29min) there were 958 ANDs matched initially and 941 remained after the isomorphism test and trimming. Thus, the initial matching found by our algorithm is usually a very good approximation of an isomorphic subset of nodes and suggests that extended simulation introduced few errors. The last column represents the runtime of each experiment.

The entries marked with '?' are because of the following. After trimming is done, we take the set of matched nodes and instantiate additional PIs and POs that connect to the regions of change. This is done for both circuits, and the isomorphism check is done again. In the ? cases, the check fails, possibly because the fanouts to the regions of change differ in the two circuits.

Our preliminary implementation works reasonably well but it has not been fine-tuned in terms of quality of results and runtime. However, the results are encouraging, demonstrating that large isomorphic subgraphs can be found in two somewhat similar circuits. On the other hand, we don't know how close we are to finding the largest isomorphism that exists. In some cases, the synthesis done to produce the second circuit was quite light, but we do not know how many nodes were actually changed. In other cases, a heavier synthesis was done, e.g. retiming and signal correspondence. One can judge this roughly by the extent of mismatch between the numbers of the FFs and ANDS in the two circuits as given in Column 2 of the table. Such examples show that the method is effective in finding isomorphisms even when the two circuits have significant regions of difference.

7 Conclusions and Future Work

This paper describes two complementary contributions:

- An algorithm for finding large isomorphic subgraphs of two AIGs to be compared, which is fast and effective.
- A way of using the detected isomorphism in SEC.

Experiments so far have shown the proposed algorithm can identify large isomorphic subgraphs on very large circuits. Future work will include its use sequential equivalence checking.

Acknowledgements

This work is supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668 entitled "Sequentially Transparent Synthesis", and the California MICRO Program with industrial sponsors Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

References

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

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations". *Proc. ICCD'06*, pp. 259-266.

[3] J. Baumgartner, H. Mony and A. Aziz, "Optimal constraint-preserving netlist simplification", *Proc. FMCAD'08*, pp. 18-26.

[4] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Release 70930. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs". *Proc. TACAS '99,* pp. 193-207.

[6] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal". *Proc. FMCAD'00.* LNCS, Vol. 1954, pp. 372-389.

[7] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits", *Proc. ICCAD'90*, pp. 126-129.

[8] N. Een and N. Sörensson, "An extensible SAT-solver". *SAT '03.* http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat

[9] N. Een and N. Sörensson, "Temporal induction by incremental SAT solving", *Proc. BMC'03*. Electronic Notes in Theoretical Computer Science, Vol. 89(4), Elsevier, 2003, pp. 543-560.

[10] C. A. J. van Eijk, "Sequential equivalence checking based on structural similarities", *IEEE Trans. CAD*, vol. 19(7), July 2000, pp. 814-819.

[11] A. P. Hurst, "Automatic synthesis of clock gating logic with controlled netlist perturbation", *Proc. DAC'08*, pp. 654-657.

[12] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[13] J.-H. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective". *IEEE TCAD*, Vol. 25 (12), Dec. 2006, pp. 2674-2686.

[14] J.-H. Jiang and W.-L. Hung, "Inductive equivalence checking under retiming and resynthesis", *Proc. ICCAD'07*, pp. 326-333.

[15] D. Kaiss, M. Skaba, Z. Hanna, and Z. Khasidashvili, "Industrial strength SAT-based alignability algorithm for hardware equivalence verification", *Proc. FMCAD'07*, pp. 20-26.

[16] 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.

[17] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking". *Proc. ICCAD '04,* pp. 50-57.

[18] F. Lu, L. Wang, K. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," *Proc. DAC `03*, pp. 668-673.

[19] F. Lu and T. Cheng, "IChecker: An efficient checker for inductive invariants". *Proc. HLDVT '06*, pp. 176-180.

[20] K. L. McMillan, "Interpolation and SAT-Based model checking". *Proc. CAV'03*, pp. 1-13.

[21] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples", *Proc. TACAS'03*, LNCS-2619, 2003, pp. 2-17.

[22] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report*, EECS Dept., U. C. Berkeley, March 2005.

[23] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843

[24] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[25] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iccad08_seq.pdf

[26] A. Mishchenko and R. K. Brayton, "Recording synthesis history for sequential verification", *Proc. FMCAD'08*, pp. 27-34. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fmcad08_haig.pdf

[27] M. N. Mneimneh and K. A. Sakallah. "Principles of sequential-equivalence verification." *IEEE D&T Comp*. Vol. 22(3), pp. 248-257, 2005.

[28] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it". *Proc. DAC'05*.

[29] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification", *Proc. DATE'09*. http://www.eecs.berkeley.edu/~alanmi/publications/2009/date09_srm.pdf

[30] C. Pixley, "A theory and implementation of sequential hardware equivalence", *IEEE Trans. CAD*, vol. 11(12), Dec 1992, pp. 1469-1478.

[31] E. Sentovich et al, "SIS: A system for sequential circuit synthesis". *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, UC Berkeley, 1992.

[32] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton, "Theory of safe replacements for sequential circuits", *IEEE Trans. CAD*, vol. 20(2), February 2001, pp. 249-265.

[33] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDDs", *Proc. ICCAD'90*, pp. 130-133.

[34] Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

Table 1. Experimental results.

pair name	initial	initial FF	initial AND	isomorphic AND	total runtime
	((f, a)(f, a))	match	match	match	(sec)
29min, 29min2	(66, 1061)(61, 967)	61	958	941	0.22
29min2, 29min3	(61, 967)(59, 931)	57	900	854	0.19
amba9min, amba9min2	(52, 29641)(52, 29113)	47	27790	27540	108.3
brpmin, brpmin2	(44, 301)(44, 300)	34	204	189 ?	0.05
guid2min, guid2min2	(86, 1291)(86, 1286)	85	790	725 ?	0.3
guid2min2, guid2min3	(86, 1286)(86, 1333)	85	690	610 ?	0.27
ns8min, ns8min2	(98, 907)(98, 898)	97	887	884 ?	0.22
p00min, p00min2	(88, 933)(88, 919)	82	851	838	0.25
p01min, p01min2	(100, 1310)(98, 1026)	29	464	455	0.17
soap2min, soap2min2	(100, 733)(100, 732)	100	730	729	0.16
atavhd, atavhd2	(594, 4285)(562, 3795)	558	3136	3093	2.52
atv, atv2	(530, 3245)(522, 3245)	514	3223	3219	2.14
corr, corr2	(219, 1366)(219, 1366)	219	1221	1218	0.56
dctmin, dctmin2	(171, 801)(171, 786)	131	336	335	0.33
hdlc, hdlc2	(408, 1573)(363, 1573)	213	949	947	0.47
mips, mips2	(1256, 14480)(1257, 14445)	1213	8809	8217 ?	26.83
nut, nut2	(37, 76)(31, 71)	18	38	36	0.02
pci, pci2	(1354, 8862)(828, 5896)	733	4941	4784	7.09
vga, vga2	(1068, 6426)(1068, 6425)	1045	6320	6318	7.91