Scalable and Scalably-Verifiable Sequential Synthesis

Alan Mishchenko **Michael Case** **Robert Brayton** **Stephen Jang**

Department of EECS
University of California, Berkeley
{alanmi, casem, brayton}@eecs.berkeley.edu

Xilinx Inc.
San Jose, CA
sjang@xilinx.com

Abstract

*This paper describes an efficient implementation of an effective sequential synthesis operation that uses induction to detect and merge sequentially-equivalent nodes. State-encoding, scan chains, and test vectors are essentially preserved. Moreover, the sequential synthesis results are **guaranteed** to be sequentially verifiable against the original circuits. Verification can use an independent inductive prover similar to that used for synthesis, with guaranteed completeness and with experimental runtimes close to that of synthesis. Experiments with this form of sequential synthesis show surprising effectiveness; when applied to the largest academic benchmarks, an average reduction in both registers and area of more than 30% is obtained; on a set of 50 industrial benchmarks ranging in size from 100 to 20K registers an average reduction of 32% in registers and 15% in area is obtained while preserving delay. The geometric means for the runtimes for synthesis and sequential verification on this set were 7.8 sec and 4.4 sec, respectively. The implementation is publicly available in the synthesis and verification system ABC.*

1 Introduction

Given a circuit with registers initialized to a given state, *the set of reachable states* includes any state that can be reached from the initial state. *Sequential equivalence* requires that the two circuits produce identical sequences at the primary outputs (POs) for the same primary input (PI) sequence, starting at the initial states. A sufficient condition for this is that they are combinationally equivalent on the reachable states.

Combinational synthesis (CS) involves changing the combinational logic of the circuit with no knowledge of its reachable states. As a result, the Boolean functions of the POs and register inputs are preserved for **any** state of the registers. CS methods allow some flexibility in modifying the circuit structure and can be easily verified using state-of-the-art combinational equivalence checkers (CEC). However, they have limited optimization power, e.g. reachable state information is not exploited.

In contrast, *traditional sequential synthesis* (TSS) can modify the circuit where behavior is preserved on the reachable states but arbitrary changes are allowed on the unreachable states. Thus after TSS, the POs and register inputs can differ as combinational functions expressed in terms of the register outputs and PIs, but the resulting circuit is sequentially-equivalent to the original one. Since many practical circuits have reachable state sets that are a small fraction of all states, TSS can result in significant logic restructuring, compared to CS. Thus, when the CS methods reach their limits, TSS becomes the next thing to try. This is happening now because design teams that traditionally used only CS are turning to sequential synthesis for additional delay minimization and power reduction.

Scalable[1] sequential synthesis as used in SIS [23] is predominantly structural. It performs *register sweep*, which merges stuck-at-constant registers, and *register retiming*, which moves registers over combinational nodes while preserving the sequential behavior of the POs. In addition to its limited nature, a significant drawback is that it changes state encoding and hence may invalidate initialization sequences and functional test-vectors developed for the original design. Also, it was shown that if retiming is interleaved with CS, then proving sequential equivalence is, in general, PSPACE-complete [11].

Thus, sequential synthesis based on register sweeping and retiming, although scalable, has limited optimization power, invalidates state-encoding, initialization-sequences and test-benches, and may be hard to verify.

This paper is concerned with a type of scalable sequential synthesis based on identifying pairs of *sequentially-equivalent nodes* (i.e., signals having the same or opposite values in all reachable states). Such equivalent nodes can be merged without changing the sequential behavior of the circuit, leading to a substantial reduction of the circuit, e.g. some pieces of logic can be discarded because they no longer affect the POs. *k*-step induction [9][4][21][15] can be used to efficiently compute pairs of sequentially-equivalent nodes.

We call this kind of synthesis, verifiable sequential synthesis (VSS) and show that it mitigates most of the drawbacks listed above for TSS. Although VSS is quite restrictive compared TSS, it is scalable and powerful, as can be seen from the experimental results. Unlike retiming, VSS requires only minor systematic changes to the state-encoding, initialization-sequences, and test-benches, only involving dropping the state-bits of the removed registers. Also, it is significant that, unlike verifying after TSS, the results of the proposed synthesis are straight-forward to verify. We prove in Section 4 and confirm experimentally in Section 5 that, the runtime of verification after VSS can be faster than VSS itself.

The contributions of this paper are,

- a new efficient method for partitioned register correspondence,
- an efficient scalable implementation of *k*-step induction (primarily due to speculative reduction and partitioning during register correspondence),
- an efficient and scalable sequential synthesis flow which is surprisingly effective, resulting in significant reductions in registers and area over CS (demonstrated over the largest academic benchmarks and a large set of large industrial designs), and

[1] SIS offers other sequential operations which are not scalable, such as extracting the reachable and using its complement as don't cares.

- a theoretical result (supported by experiments) that sequential equivalence checking (SEC), based on induction (ISEC), after VSS is scalable and complete.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 describes the algorithms of VSS. Section 4 discusses verification after VSS. Section 5 reports experimental results. Section 6 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 and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. The terms memory elements, flop-flops, and registers are used interchangeably in this paper.

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. It is assumed that each node has a unique integer called its *node ID.*

A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network, reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node *n* is a subset of the fanin cone, such that every path from a node in the MFFC to the POs passes through *n*. Informally, the MFFC of a node contains all the logic used exclusively by the node. If a node is removed, its MFFC can also be removed.

Merging node *n* onto node *m* is a structural transformation of a network that (1) transfers the fanouts of *n* to *m* and (2) removes *n* and its MFFC. Merging is often applied to a *set* of nodes that are proved to be equivalent. In this case, one node is denoted as the *representative* of an equivalence class, and all other nodes of the class are merged onto the representative. The representative can be any node of the class such that its fanin cone does not contain any other node of the same class. In this work, the representative is a node of the class that appears first in given topological order.

SAT sweeping is a technique for detecting and merging equivalent nodes in a combinational network [14][13][17][18]. SAT sweeping is based on a combination of simulation and Boolean satisfiability. Random simulation is used to divide the nodes into candidate equivalence classes. Next, each pair of nodes in each class is considered in a topological order. A SAT solver is invoked to prove or disprove their equivalence. If the equivalence is disproved, a counter-example is used to simulate the circuit, which may result in disproving other candidate equivalences. SAT sweeping is used as a robust combinational equivalence checking technique and as a building block in VSS.

Bounded model checking (BMC) uses Boolean satisfiability to prove a property true for all states reachable from the initial state in a fixed number of transitions (*BMC depth*). In the context of equivalence checking, BMC checks pair-wise equivalence of the outputs of two circuits under verification. A *timeframe* (or *frame*) of a sequential circuit is one copy of combinational logic used in the circuit. When the circuit is *unrolled* for *k* frames, its combinational logic is duplicated *k* times and the registers between the frames are removed. BMC is typically implemented by applying SAT sweeping to the unrolled frames of the circuit.

3 Sequential Synthesis

This section gives an overview of the steps used in VSS.

3.1 Register sweep

The idea of register sweep is to look for registers that are stuck-at constant, depending on the initial state. Initial x-values are allowed. Structural register sweep iterates the procedure in Figure 3.1 as long as there is a reduction in the number of registers.

This procedure starts with the assumption that all registers have the 0 initial state. If a registers has a 1 initial state, it is transformed by adding a pair of inverters at the output of the register and retiming the register forward over the first inverter. If a register has a don't-care initial state, it is transformed by adding a new PI and a MUX controlled by a special register that produces 0 in the first frame and 1 afterwards.

Detection of stuck-at-constant registers using ternary simulation is based on the algorithm given in [5]. This assigns the initial values to the registers and simulates the circuit using x-valued primary inputs. The ternary states reached at the registers are collected. Simulation stops when a new ternary state is equal to a previously seen ternary state. At this point, if some register has the same constant value in every reachable ternary state, this register is declared stuck-at-constant.

```
aig runStructuralRegisterSweep( aig N )
{
    // start the set of equivalent register pairs
    set of node subsets Classes = ∅;

    // detect registers with combinationally-equivalent inputs
    for each register r1 in aig N
        if (there is register r2 in aig N with the same driver as r1)
            Classes = Classes ∪ {r1, r2};

    // detect registers that are stuck-at-constant
    analyzeRegistersUsingTernarySimulation( N );
    for each register r1 in aig N
        if ( register r1 is stuck-at-constant c )
            Classes = Classes ∪ {c, r1};

    // use the equivalences to reconstruct the aig
    aig N1 = mergeEquivalences( N, Classes );
    return N1;
}
```

Figure 3.1. Structural register sweep.

3.2 Signal-correspondence

Signal-correspondence is a computation of a set of classes of sequentially-equivalent nodes using induction. The classes are *k*-step-inductive in the following sense:

- **Base Case -** they hold for all inputs in the first *k* frames starting from the initial state, and
- **Inductive Case -** if they are assumed to be true in the first *k* frames starting from *any* state, then they hold in the $k+1^{st}$ frame.

Our implementation of signal-correspondence follows previous work in [9][4][21][15]. The pseudo-code is given in Figure 3.2.

It was found that the scalability of signal-correspondence hinges on the way the candidate equivalences are assumed before they are proved in the $k+1^{st}$ frame of the inductive case. We use a technique known as *speculative reduction,* pioneered in [21]. A similar approach was proposed and used in [15].

Speculative reduction merges any node of an equivalence class in each of the first *k* time frames onto its representative. After merging, the non-representative node is *not* removed, because a

constraint is added to assert that this node and its representative are equal. Merging facilitates logic reduction in the fanout cone of the node. For example, an AND gate with inputs *a* and *b* can be removed if *b* has been merged onto *a*. Propagating these changes can make downstream merges trivial and many corresponding constraints redundant. Experiments confirm a dramatic decrease in the number of constraints added to the SAT solver. The gain in runtime due to speculative reduction can be several orders of magnitude for large designs.

```
aig runSignalCorrespondence( aig N, int k )
{
    // detect candidate equivalences using random simulation
    set of node subsets Classes = randomSimulation( N );
    // refine equivalences by BMC from the initial state for depth k-1
    refineClassesUsingBMC( N, k-1, Classes );
    // perform iterative refinement of candidate equivalence classes
    do {
        // do speculative reduction of k-1 uninitialized frames
        network NR = speculativeReduction( N, k-1, Classes );
        // derive SAT solver containing CNF of the reduced frames
        solver S = transformAIGintoCNF( NR );
        // check candidate equivalences in k-th frame
        performSatSweepingWithConstraints( St, Classes );
    }
    while ( Classes are refined during SAT sweeping );
    // merge computed equivalences
    aig N1 = mergeEquivalences( N, Classes );
    return N1;
}
```

Figure 3.2. Signal-correspodence using k-step induction.

3.3 Partitioned register-correspondence

Register-correspondence is a special case of signal-correspondence, when candidate equivalences are limited to register outputs. There are two key insights here. One is that for the $k = 1$ case, it is enough to consider one time-frame of the circuit, while signal-correspondence requires at least two time-frames. This is because the inductive case can look at the register outputs in the first time frame.

```
set of node subsets partitionOutputs( aig N, parameters Pars )
{
    // for each PO, compute its structural support in terms of PIs
    set of node subsets Supps = findStructuralSupps( N );
    // start the output partitions
    set of node subsets Partitions = ∅;
    // add each PO to one of the partitions
    for each PO n of aig N in a degreasing order of support sizes {
        node subset p = findMinCostPartition(n, Partitions, Pars);
        if ( p != NONE )
            p = p ∪ {n};
        else
            Partitions = Partitions ∪ {{n}};
    }
    // merge small partitions
    compactPartitions( Partitions, Pars );
    return Partitions;
}
```

Figure 3.3.1. Output-partitioning for induction.

The second insight is that using only one timeframe allows partitioning to be effective. Verification of the equivalences can be done by partitioning the registers and proving each partition separately. Each partition must include its transitive fanin, which for one frame can be a small fraction of the circuit, but for two or more time frames can become nearly the whole circuit. A side benefit is that this is obviously parallelizable.

It is typical that some of the speculated equivalences do not hold, so refinement is needed. Since partitioning is fast, repartitioning can be done during each refinement iteration. This leads to improved performance, compared to using an initial partition across multiple refinements.

The idea of partitioning was motivated by observing that most of the runtime of signal correspondence for large designs was spent in Boolean constraint propagation during the satisfiable SAT runs plus simulation of the resulting counter-examples. Partitioning allows the inductive problem to be solved independently by several instances of a SAT solver, without impacting the completeness of equivalences proved.

Our partitioning algorithm is shown in Figure 3.3.1. The partitions found by the algorithm are used in the procedure **performSatSweepingWithConstraints** of Figure 3.2, which does register-correspondence by limiting candidates to register outputs.

The partitioning algorithm in Figure 3.3.1 takes a combinational AIG and a set of parameters, e.g. bounds on the partition sizes. The structural supports for all outputs (these are the register inputs for the single time frame) are computed in one sweep over the network. These are sorted by support size. The outputs are added to the partitions in decreasing order of their support sizes. The procedure that determines the partition to which an output is added, considers the cost of adding the output to each of the partitions. The cost used is a linear combination of attraction (proportional to the number of common variables) and repulsion (proportional to the number of new variables introduced in the partition if the given output is added). The coefficients of the linear combination were found experimentally. In addition to the cost considerations, a partition is not allowed to grow above a given limit. If the best cost of adding an output exceeds another limit, NONE is returned. In this case, the calling procedure starts a new partition. In the end, some partitions are merged if their sizes are below a given minimum.

The following observations are used when creating a partition:

- All nodes in a candidate class are added to one partition (allows for proving the largest set of register equivalences).
- Constant candidates can be added to any partition.
- Candidates are merged at the PIs and proved at the POs.
- After solving all partitions, the classes are refined if counterexamples are found.

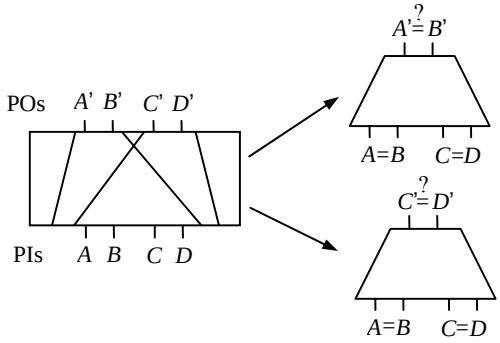

Figure 3.3.2. Illustration of output partitioning for induction.

An example in Figure 3.3.2 illustrates partitioning for induction with two candidate equivalence classes of registers, $\{A, B\}$ and $\{C, D\}$, which are added to different partitions. Note that the equivalences $A = B$ and $C = D$ are assumed in both partitions.

4 Sequential Equivalence Checking (SEC) and Scalability

SEC starts with construction of the sequential miter between the two circuits to be compared. The pseudo-code for a form of SEC, based on induction, is shown in Figure 4.1. This is denoted ISEC in this paper. The sequence of transformations applied by the integrated ISEC command in ABC (command "dsec") illustrates how synthesis is used in verification. The procedure stops as soon as the sequential miter is reduced to a constant zero, or a counter-example is found. Termination conditions are checked after any command; e.g., after register sweeping or after signal-correspondence (not shown in the pseudo-code). Names of the corresponding stand-alone commands in ABC are given to the left of each procedure call in Figure 4.1. Note that a major part of ISEC is the synthesis operations from Section 3.

The main verification result of this paper is related to the scalability of sequential verification after VSS.

Theorem. Let *N* be a sequential circuit with a given initial state. Suppose some signals in *N* are proved sequentially-equivalent using VSS with *k*-step induction and merged by replacing each signal by the representative of its equivalence class. Assume that the logic is not further restructured and denote the resulting circuit by *N*'. Let *M* be the miter of *N* and *N*'. Then the equivalences of the corresponding POs of *N* and *N*' can be proved by *k*-step induction, where *k* is the same as used in VSS.

```
sequentialEquivalenceChecker( network N1, network N2 )
{
    // convert networks to AIG; pair PIs/POs by name;
    // transform registers to have constant-0 initial state
    aig M = createSequentialMiter( N1, N2 );// command "miter -c"

    // remove logic that does not fanout into POs
    runSequentialSweep( M ); // command "scl"

    // remove stuck-at and combinationally-equivalent registers
    runStructuralRegisterSweep( M ); // command "scl -l"

    // move all registers forward and compute new initial state;
    // this command can be disabled by switch "–r", e.g. "dsec –r"
    runForwardRetiming( M ); // command "retime –M 1"

    // merge sequential equivalent registers
    // (this completely solves SEC if only retiming was performed)
    runPartitionedRegisterCorrespondence( M ); // com. "lcorr"

    // merge comb. equivalence before trying signal-correspondence
    runCombinationalSatSweeping( M ); // command "fraig"

    for ( k = 1; k ≤ 64; k = k * 2 ) {
      // merge sequential equivalences by k-step induction
      runSignalCorrespondence( M, k ); // command "ssw –K"

      // minimize and restructure speculated combinational logic
      runAIGrewriting( M ); // command "drw"

      // move registers forward after logic restructuring
      runForwardRetiming( M ); // command "retime –M 1"

      // target satisfiable SAT instances
      runSequentialAIGsimulation( M );  // command "sim"
    }

    // if miter is still unsolved, save it for future research
    dumpSequentialMiter( M ); // command "write_aiger"
}
```

Figure 4.1. ISEC - Integrated inductive SEC in ABC.

In the above theorem it is important that there is no further restructuring of the circuit after VSS. Otherwise, equivalent points common to both networks may be lost, resulting in the loss of the ability to prove equivalence inductively [12]. It is also important that when ISEC is applied to the results of VSS, retiming should be disabled (command "dsec –r") in the verification algorithm.

However, for ISEC, not specific to VSS, it was found experimentally that applying the most-forward retiming to the sequential miter before *k*-step induction often leads to substantial speed-ups - but only if some form of retiming was applied during synthesis! In the VSS experiments (Section 5), retiming is not used and it was found helpful to skip retiming when verifying the results of VSS. The intuition is that as a result of merging nodes in VSS, structural changes in the circuit allow registers to travel to different locations after the most-forward retiming. When this happens, alignment of the register locations across the original and the final circuit may be lost. On the other hand, if retiming is used as part of synthesis, then applying the most-forward retiming during verification facilitates alignment of the registers in both copies of the design. This alignment is perfect if retiming was the only transformation during synthesis. In this case, register-correspondence using simple induction is enough to solve the verification problem [12]. But even if some logic restructuring was done before or after retiming, the most-forward retiming during verification tends to increase the number of matches of register locations. This tends to speed up verification and help solve difficult instances by making them inductive.

5 Experimental results

The synthesis and verification algorithms are implemented in ABC [3] as commands *scl, lcorr, ssw* and *dsec*. The SAT solver used is a modified version of MiniSat-C_v1.14.1 [8]. The experiments targeting FPGA mapping into 6-input LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The resulting networks are verified by ISEC.

The academic benchmarks used for the experiments are 25 of the largest benchmarks from the ITC'97, ISCAS'89, and IWLS'05 suites [10]. The industrial benchmarks are 50 industrial circuits ranging in size from 100 to 20K registers. For industrial benchmarks with multiple clock domains, optimization was applied only to the domain with the largest number of registers. Registers of other clock domains were represented as additional pairs of PIs and POs, disallowing sequential flexibility in the related logic cones. An additional experiment was run using several IBM benchmarks.

The following ABC commands were included in the scripts used to collect the experimental results:

- *resyn* is a CS script that performs 5 iterations of AIG rewriting [20].
- *resyn2* is a CS script that performs 10 iterations of AIG rewriting that are more diverse than those of *resyn.*
- *choice* is a CS script that allows for accumulation of structural choices [6]; *choice* runs *resyn* followed by *resyn2* and collects three snapshots of the network: the original, the final, and the intermediate one saved after *resyn.*
- *if* is a structural FPGA mapper with priority cuts [19], fine-tuned area recovery, and the capacity to map over a subject graph with structural choices [6] (the mapper computes and stores at most 8 6-input priority cuts at each node; it performs five iterations of area recovery, three with area flow and two with exact local area)
- *scl* is a structural register sweep (Section 3.1).
- *lcorr* is a partitioned register-correspondence computation using simple induction (*k* = 1) (Section 3.3).

- *ssw* is a signal-correspondence computation using *k*-step induction (Section 3.2)
- *dsec* is a ISEC command (Section 4)

Three experimental runs were performed:

- **Baseline Run** corresponds to a typical run of high-effort baseline combinational synthesis, consisting of technology-independent CS and technology mapping with structural choices for FPGA architectures with 6-LUTs. Script *baseline* is defined as follows: (*choice; if; choice; if; choice; if*).
- **Reg Corr Run** corresponds to register sweep and partitioned register-correspondence followed by the baseline synthesis and mapping (*scl; lcorr; baseline*).
- **Sig Corr Run** corresponds to register sweep, partitioned register-correspondence, signal-correspondence, followed by the baseline synthesis and mapping (*scl; lcorr; ssw; baseline*).

Tables 1-2 show the summary of results for the set of 25 academic benchmarks. Tables 3-4 show the summary of results for the set of 50 industrial benchmarks.

In Tables 1 and 3, sections "Baseline Run", "Reg Corr Run", and "Sig Corr Run" show geometric averages of the parameters after the corresponding runs. Columns "Ratio" show the ratios of the columns to the left against column "Baseline". The rows show the number of registers, area (the number of 6-LUTs), and delay (the depth of 6-LUT network).

In Tables 2 and 4, columns correspond to transformations. The row "Geomean" shows the average runtimes of each transformation. Row "Ratio" shows the ratios of the runtimes to the total runtime of synthesis and verification. The runtimes of the synthesis operations are comparable to the ISEC runtimes.

The breakdown of results for the academic benchmarks (registers, area, delay, and runtime) can be found online [24].

Synthesis results

A substantial reduction (about 33%) over the baseline case is achieved on academic benchmarks (Table 1) in both registers and area while the delay is reduced by 14%. Such large reductions on these benchmarks may be the result of unrealistic initial states that were possibly reset to the all-0 state after the original initial state was lost. In this case, the proposed sequential synthesis removes the logic that is not exercised on the reachable states.

The reduction achieved for the industrial benchmarks (Table 3) is 32% in registers and 15% in area while preserving the delay. The geometric averages of synthesis runtimes and verification runtimes on this set are close to 7.8 sec and 4.4 sec, respectively.

An additional experiment was run on 9 industrial benchmarks from IBM. On this set, the reduction in registers and LUTs is 14% and 9%, respectively.

The large savings in registers and area for the industrial benchmarks may have three sources:

- Inefficient HDL compilers, which create equivalent or redundant registers.
- Presence of sequential logic that is not exercised when the design starts from the given initial state.
- Intentional early duplication of sequential logic to facilitate backend synthesis.

Verification results

The verification runtimes and runtime ratios are shown in Tables 2 and 4 in columns "ISEC".

It is interesting to note that, for academic benchmarks, the verification runtimes are larger than that of sequential synthesis, possibly because larger synthesis reductions lead to larger structural gaps that need to be bridged by Boolean reasoning during the iterative refinement. For the industrial benchmarks, verification takes less time than synthesis. A possible explanation is that verification uses the partitioned implementation of register-correspondence which can quickly reduce the size of the sequential miter composed of two similar copies of the same design, leaving less work for the slower signal-correspondence. Meanwhile, synthesis applies signal-correspondence to a single copy of the design and gave a reduction of only by 30% on average after partitioned register-correspondence.

Finally, we tried applying ISEC to several sets of industrial problems, which had not been synthesized by VSS. These problems ranged in size from less than a hundred to several thousand registers. Most were solved successfully within several minutes, much faster than existing SEC algorithms, but on some, ISEC gave up after attempting *k*-step induction up to a predefined limit of $k = 32$. The details are not included because the type of synthesis is unknown and therefore these results are not relevant to VSS. However, the results do suggest that ISEC can be an advance for general SEC.

Scalability of verification

To confirm the importance of preserving logic structure after VSS before ISEC (stated in the theorem of Section 4), two additional experiments were performed on the academic benchmarks: (1) ISEC was applied to the results of VSS followed by CS, (2) ISEC was applied to the results of VSS followed by minimum-delay retiming and CS. In both cases, CS was as described in this Section as "Baseline". The geometric mean runtime of ISEC in these two experiments increased by 2.5x and 14x, respectively. Timeouts at 5000 seconds were observed in 6 out of 25 benchmarks in the experiment that included retiming.

The breakdown of runtimes for the academic benchmarks in the two additional experiments can be found online [24].

6 Conclusions and future work

This paper presents a sequential synthesis method and implementation (VSS) and a corresponding scalable approach to ISEC that can efficiently validate the VSS results. The algorithms have the following salient features:

- efficient implementation using partitioned inductive solving.
- substantial savings in registers and area without increasing the delay.
- minimum modification to state encoding, scan chains, and functional test vectors after synthesis.
- scalable sequential verification by an inductive prover comparable in the amount of runtime to that used for scalable synthesis.

The experimental results on both academic and industrial benchmarks confirm the practicality of the proposed synthesis and demonstrate affordable runtimes of unbounded SEC after VSS. Although we used the same code to synthesize and verify the results in our experiments, we note that the verification phase should use an independent inductive prover for insurance purposes. Of course, such a prover can use the ideas of this paper for efficient implementation.

In practice, once VSS has been done, CS should be run to take full advantage of the VSS results. However, in this case, we can't guarantee scalable verification and demonstrated this with some additional experiments when CS and retiming was used. One possibility to guarantee verifiability would be to run a CS (say, *baseline*) and store the result, say, circuit $C1$. Then run VSS and store the result, say circuit $C2$. Finally, confine all synthesis after

this to CS, resulting in circuit *F*. Then, a scalable SEC verification would involve the sequence *CEC*(initial, *C*1), *SEC*(*C*1,*C*2), *CEC*(*C*2, *F*). This approach can be implemented in commercial tools, resulting in a scalable sequential verification methodology.

Future work in this area will include:

- Tuning the inductive prover for scalability (for example, using unique-state constraints).
- Developing new sequential engines (interpolation [16], synthesizing equivalence for induction, etc).
- Extending the proposed form of sequential synthesis to include (a) on-the-fly retiming [1], (b) logic restructuring using unreachable states as external don't-cares, (c) iterative processing similar to that of combinational synthesis [20].

We also plan to make our implementation of VSS applicable to sequential circuits with multiple clock domains and different register types. For this, registers will be grouped into classes. Each class will include registers of the same type clocked by the same clock. Merging of registers will be allowed if they belong to the same class. Merging of combinational nodes will be allowed if they are in the cone of influence of registers of the same class.

Acknowledgements

This work was 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, Calypto, Intel, Magma, Synopsys, Synplicity, and Xilinx. The authors are indebted to Jin Zhang for her careful reading and useful suggestions in revising the manuscript.

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*.

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

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

[5] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification", *Proc. ICCAD'06*, pp. 1076-1082.

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05,* pp. 519-526.

[7] 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.

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

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

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

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

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

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

[14] 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.

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

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

[17] 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.

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

[19] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*.

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

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

[22] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

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

[24] http://www.eecs.berkeley.edu/~alanmi/publications/other/vss.pdf

Table 1. Geomeans of register count, area, and delay after VSS for 25 academic benchmarks.

	Baseline Run	Reg Cor Run	Ratio	Sig Corr Run	Ratio
Registers	809.9	610.9	**0.75**	544.3	**0.67**
6-LUTs	2141	1725	**0.80**	1405	**0.65**
Depth	6.8	6.33	**0.93**	5.83	**0.86**

Table 2. Runtime of VSS and ISEC for 25 academic benchmarks.

	Reg Corr	Sig Corr	ISEC	Syn & Map	Total
Geomean	0.27	2.12	3.51	6.45	16.44
Ratio	**0.05**	**0.22**	**0.60**	**0.12**	**1.00**

Table 3. Geomeans of register count, area, and delay after VSS for 50 industrial benchmarks.

	Baseline Run	Reg Corr Run	Ratio	Sig Corr Run	Ratio
Registers	1583	1134	**0.71**	1084	**0.68**
6-LUTs	7472	6751.5	**0.90**	6360	**0.85**
Depth	8.5	8.5	**1.00**	8.5	**1.00**

Table 4. Runtime of VSS and ISEC for 50 industrial benchmarks.

	Reg Corr	Sig Corr	ISEC	Syn & Map	Total
Geomean	0.55	7.28	4.43	23.98	48.51
Ratio	**0.02**	**0.47**	**0.26**	**0.25**	**1.00**