Scalable and Scalably-Verifiable Sequential Synthesis

Alan Mishchenko **Michael Case** **Robert Brayton**

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

Stephen Jang

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

Abstract

This paper describes an efficient implementation of sequential synthesis 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 using an independent inductive prover similar to that used for synthesis, with guaranteed completeness. Experiments with this sequential synthesis show surprising effectiveness. When applied to a set of 20 industrial benchmarks ranging up to 26K registers and up to 53K 6-LUTs, average reductions in register and area are 12.9% and 13.1% respectively while delay is reduced by 1.4%. When applied to the largest academic benchmarks, an average reduction in both registers and area is more than 30%. The associated sequential verification is very scalable, with the runtime only about 50% longer than that of synthesis. 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 used.

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 [25] 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 [12].

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 [10][5][23][17] can be used to efficiently compute pairs of sequentially-equivalent nodes. This technique subsumes several other approaches that detect equivalent registers [15] [26].

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 to 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 VSS 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 and partitioned signal-correspondence,
- an efficient scalable implementation of *k*-step induction (primarily due to speculative reduction and partitioning),
- an efficient and scalable sequential synthesis flow which is surprisingly effective, resulting in significant reductions in registers and area over CS (demonstrated over a large set of large industrial designs and the largest available academic benchmarks), and
- a theoretical result (supported by experiments) that inductive sequential equivalence checking (ISEC) is scalable and complete after VSS.

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

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 [16][14][19][20]. 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) [4] 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 [6]. 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, they hold in the k+1st frame.

Our implementation of signal-correspondence follows previous work in [10][5][23][17]. 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+1st frame of the inductive case. We use a technique known as *speculative reduction,* pioneered in [23]. A similar approach was proposed and used in [17].

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( S, 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.

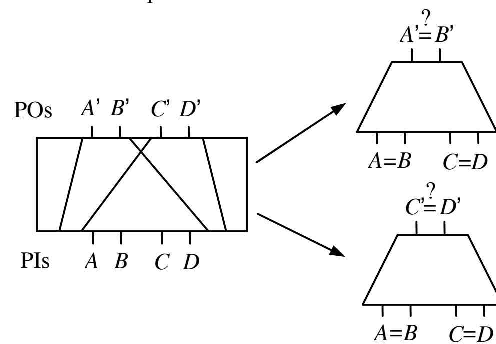

Figure 3.3.2. Illustration of output partitioning for induction.

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.

3.4 Partitioned signal-correspondence

Partitioning for register-correspondence is efficient and does not degrade the quality of results. In other words, a register equivalence found by register-correspondence without partitioning can also be found using the partitioned approach.

We investigated a similar lossless partitioning approach for signal-correspondence. In this case, two time-frames of the designs are to be partitioned so that the resulting partitions contain all relevant constraints. Because the total number of constraints is typically in the tens of thousands and the number of constraints affecting one property (a candidate equivalence) is typically in the hundreds, the partitions for signal-correspondence are harder to compute. Moreover, experiments have shown that this approach tends to create partitions with thousands of registers. In some cases, the only way to ensure losslessness, is to perform signal-correspondence on the whole design.

When it was found that the lossless partitioning is not feasible for signal-correspondence, we developed a fast lossy approach. The idea is to divide registers into groups while minimizing dependencies between the groups. Each register group together with logic feeding into it becomes a partition. If a partition depends on external registers, they are replaced by free variables.

Now when signal-correspondence is applied to the resulting partitions, it computes a conservative approximation. We experimented with several ways of grouping registers and found a simple approach that works. The registers are divided into groups, each containing a given number of them, by considering registers in the order in which they are specified in the design. This approach is used in the experimental results reported below.

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 right of procedure calls in pseudo-code of 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.*

Figure 4.2 illustrates the theorem. Synthesis based on *k*-step induction (shown on the left) transforms circuit *N* into circuit *N'*. Verification based on *k*-step induction (shown on the right) is applied to miter *M*, derived from *N* and *N'*. When all pairs of outputs of *N* and *N'* are proved equivalent, the output of the miter is proved identical to the constant-0 Boolean function.

```
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.

Proof: Consider circuit *N* and nodes *a* and *b* proved equivalent using *k*-step induction. When circuit *N'* is derived from *N* by merging equivalent nodes, node *a'* in *N'* is created to represent both nodes *a* and *b* in *N*. By construction node *a'* in *N'* is sequentially equivalent to nodes *a* and *b* in *N*. Since the logic was not further restructured after synthesis, miter *M* contains the nodes of both circuits *N* and *N'*. Since nodes *a, b,* and *a'* are sequentially equivalent, they fall into the same equivalence class of *M*. This is illustrated in Figure 4.2.

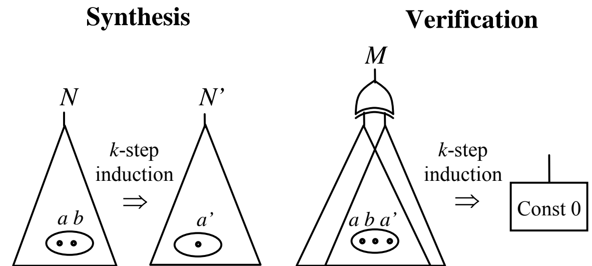

Figure 4.2. Illustration for the proof of the theorem.

A *feedback edge set S* is a set of nodes in a sequential circuit *C* such that all sequential loops of *C* are broken by a node in *S*. According to [13], for the equivalence of two sequential circuits, *C* and *C',* to be inductive, it is sufficient to find two feedback edge sets, *S* and *S',* in *C* and *C',* respectively, such that for every node in *S,* there exists an equivalent node in *S',* and vice versa. In

this case, the set of equivalence classes of nodes in the sequential miter constructed for *C* and *C'* contains classes that combine, for each node of *S,* a node from *S'*, and vice versa.

Now recall the discussion illustrated in Figure 4.2. It was shown that, among the equivalence classes of miter *M,* there are classes containing nodes *a* and *b* from *N,* and node *a'* from *N'*. For each node *a'* of *N'*, there exists a sequentially-equivalent node *a* in *N*. Similarly, for each node *a* or *b* in *N*, there exists a sequentially-equivalent node *a'* in *N'*. Since the set of all nodes of a circuit is a trivial feedback edge set, both *N* and *N'* have feedback edge sets satisfying the above condition. According to [13], equivalence of *N* and *N'* is inductive.

Moreover, sequential synthesis used *k*-step induction to prove equivalences in circuit *N*, which resulted in circuit *N'*. When *k*-step induction is applied to the miter composed of *N* and *N'*, the circuit structure of *M* has the same inductive properties as that of *N*, while *N'* is structurally a subset of *N*. Therefore, equivalence classes of *M* can be proved by induction with depth *k*. Q.E.D.

In the above theorem it is important that there is no further logic restructuring after VSS. Otherwise, equivalent points common to both networks may be lost, resulting in the loss of the ability to prove equivalence inductively [13]. It is also important that when ISEC is applied to the results of VSS, retiming is disabled during verification (use command *dsec –r*).

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 [13]. 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 [9]. 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 experiments were performed on 20 industrial sequential designs. For multi-clock designs, only the logic belonging to the most active clock domain was extracted and used. The registers of the remaining clock domains were treated as additional PIs and POs. The registers with asynchronous sets/resets were also treated as additional PIs and POs. The synchronous logic for set/reset and clock enable signals was converted to be part of the design logic. The original initial states of the registers were preserved. If a register has a don't-care initial state, it was replaced by the zero initial state for the sake of synthesis. The results of sequential synthesis after this transformation are conservative.

The profile of all benchmarks is shown in the first left part of Table 2. The columns "PI", "PO", and "Reg" show the number of primary inputs, primary outputs, and registers in the designs after preprocessing described above.

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 [22]
- *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 [7]; *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 [21], area recovery, and support of mapping with structural choices [7] (the mapper computes and stores at most eight 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 an ISEC command (Section 4)

Four experimental runs are reported in Table 1:

- The baseline (columns "if") corresponds to a typical run of high-effort technology-independent CS and technology mapping with structural choices for FPGA architectures with 6-LUTs. Script *baseline* is defined as five iterations of commands (*choice; if –C* 12 *–F* 2), followed by choosing the best result produced at the end of any iteration.
- The structural register sweep (columns "scl") stands for merging registers with identical combinational drivers and sweeping stuck-at-constant registers detected by ternary simulation [6], followed by *baseline*, i.e., five iterations of mapping with structural choices.
- The register-correspondence (columns "lcorr") consists of register sweep and partitioned register-correspondence followed by *baseline*.
- Signal-correspondence (columns "ssw") consists of register sweep, partitioned register-correspondence, and signal-correspondence followed by *baseline*. In signal-correspondence, the simple induction is used (k = 1).

The row "Geomean" in Table 1 list the geometric averages of the parameters after the corresponding runs. The rows "Ratio" shows the ratios of the averages across sections of the table.

Synthesis results

The quality of results reported in Table 1 is summarized in Figure 5.1. In this figure, the blue, dark red, and rose (gray, black, and white) bars show the reductions after structural register sweep, register-correspondence, and signal-correspondence, respectively. A substantial reduction over the baseline (about 13%) is achieved on the industrial designs in both registers and area, while the delay is reduced by about 1%.

An additional experiment was performed on 25 academic benchmarks. The breakdown of results for these benchmarks (registers, area, delay, and runtime) can be found online [27]. Another experiment was run on 9 industrial benchmarks from

IBM. On this set, the reduction in registers and LUTs after signal-correspondence is 14% and 9%, respectively.

The savings in registers and area may have several sources:

- Tools deriving logic netlists from HDLs may create identical or equivalent registers.
- Designers writing RTL may attempt logic duplication early in the design flow.
- Some circuits may be over-designed (available functionality is unreachable when the design starts from its initial state).

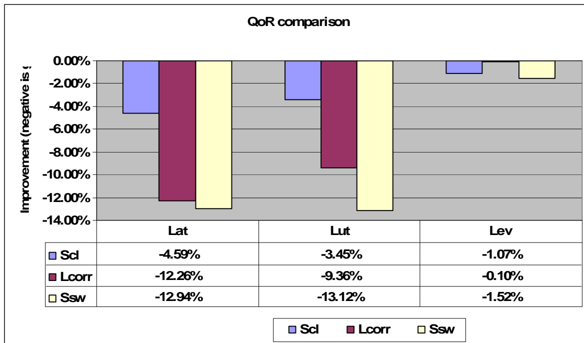

Figure 5.1. Quality of results comparison of different methods..

Partitioned signal-correspondence

Table 3 shows the results of signal-correspondence with design partitioning. The purpose of this experiment is to show the runtime/performance trade-off when the lossy partitioning is used, as described in Section 3.4. In this experiment, 8 out of 20 designs with more than 9000 registers are run with two flows:

The flow without partitioning:

- ○ *scl; lcorr*
- ○ *ssw*
- ○ 5x (*choice; if –C* 12 *–F* 2)
- ○ pick the best among the results produced by 5 runs.

The flow with partitioning:

- ○ *scl; lcorr*
- ○ *ssw –P* 9000 // partition size is 9000
- ○ 5x (*choice;if –C* 12 *–F* 2)
- ○ pick the best among the results produced by 5 runs.

After partitioning, the runtime of signal-correspondence (Columns "Time1") is reduced by 45.9% (1705 seconds vs. 922 seconds) and the runtime of the entire flow (columns "Time2") is reduced by 40.9% (1898 seconds vs. 1122 seconds). The register count is 0.3% worse, the LUT count is 0.4% worse, while the level count does not change.

Verification results

The geometric averages of synthesis and verification runtimes (column "dsec" in Table 2) for the industrial benchmarks are close to 150 sec and 220 sec, respectively.

We also 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 in several minutes, much faster than using the existing SEC algorithms, but on some ISEC gave up after attempting k-step induction up to a predefined limit of $k = 32$. The detailed results are not reported in this paper because the type of synthesis is unknown and therefore these results are not relevant to VSS. However, the results do suggest that ISEC is applicable to general-case 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 the same as in the baseline flow reported in columns "if" of Table 1. The geometric mean runtime of ISEC in these two experiments increased by 2.5x and 14x, respectively. In the second experiments that included retiming, timeouts at 5000 seconds were observed in 6 out of 25 academic benchmarks used.

The breakdown of runtimes for the academic benchmarks in the two additional experiments can be found online [27].

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,
- 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 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, CS should be run after VSS 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 both CS and retiming were used. One possibility to guarantee verifiability would be save the initial circuit $C0$, apply CS (say, script *baseline*) and store the result as circuit $C1$. Then, run VSS and store the result as circuit $C2$. Finally, confine all synthesis after this to CS, resulting in circuit $C3$. Then, a scalable SEC verification would involve the sequence $CEC(C0, C1)$, $ISEC(C1,C2)$, $CEC(C2, C3)$. 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, lazy simulation, etc).
- Developing additional sequential engines (interpolation [18], phase abstraction [6], localization, target enlargement, 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 [22].

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, IBM, Intel, Intrinsity, Magma, Synopsys, Synplicity, Tabula, 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] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs". *Proc. TACAS '99,* pp. 193-207.

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

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

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

[8] 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.

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

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

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

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

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

[15] B. Lin and A. R. Newton, "Exact removal of redundant state registers using Binary Decision Diagrams," *Proc. IFIP TC 10/WG 10.5 Intl Conf. VLSI '91*, Edinburgh, Scotland, pp. 277-286.

[16] 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.

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

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

[19] 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.

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

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

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

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

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

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

[26] E. M. Sentovich, H. Toma, and G. Berry, "Efficient latch optimization using exclusive sets", *Proc. DAC'97*, pp. 8-11.

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

Table 1. Register count, area, and logic level after VSS for industrial benchmarks.

Example	Register count				6-LUT				Logic level			
	if	scl	lcorr	ssw	if	scl	lcorr	ssw	if	scl	lcorr	ssw
Ex 1	16864	16864	13984	13984	24832	24832	21952	21952	3	3	3	3
Ex 2	26246	12771	12487	11976	38594	22699	22376	21611	8	7	8	8
Ex 3	892	790	773	749	1832	1707	1651	1156	5	5	5	4
Ex 4	5886	5886	5847	5779	11749	11749	11557	11170	12	12	13	12
Ex 5	12730	12728	12032	12013	18416	18350	17638	17570	6	6	6	6
Ex 6	8081	7922	5140	5140	11387	11220	7477	7483	9	9	9	9
Ex 7	265	265	265	265	2759	2759	2736	2587	4	4	4	4
Ex 8	3003	2984	2460	2460	10767	10753	10197	10140	11	10	11	12
Ex 9	5215	5215	4839	4839	10345	10345	9314	9132	7	7	7	7
Ex 10	41657	41566	38663	38553	34226	34116	31238	31098	8	8	8	8
Ex 11	11441	11404	10443	10383	24954	24964	23932	23831	16	16	15	16
Ex 12	2784	2774	2642	2608	5900	5862	5516	5233	12	12	12	13
Ex 13	2786	2780	2768	2766	8406	8353	8325	7628	7	7	8	7
Ex 14	722	722	721	721	4312	4356	4329	3939	5	5	5	5
Ex 15	20223	20186	17950	17634	52954	52595	47706	46603	11	10	10	10
Ex 16	2154	2154	2107	2087	4312	4299	4182	4073	7	6	6	6

Ex 17	13263	13185	12790	12790	23097	22694	21992	21916	5	6	5	5
Ex 18	7019	7019	6864	6857	5973	5969	5830	5799	7	7	7	7
Ex 19	2540	2540	1935	1935	6260	6260	5765	5786	7	7	7	7
Ex 20	23517	22316	22276	21865	34012	32410	32386	31188	12	13	13	12
Geomean	**5500**	**5248**	**4826**	**4788**	**11497**	**11100**	**10421**	**9989**	**7.47**	**7.39**	**7.46**	**7.355**
Ratio	**1**	**0.954**	**0.877**	**0.871**	**1**	**0.965**	**0.906**	**0.869**	**1**	**0.989**	**0.999**	**0.985**
Ratio		**1**	**0.920**	**0.912**		**1**	**0.939**	**0.900**		**1**	**1.010**	**0.996**
Ratio			**1**	**0.992**			**1**	**0.959**			**1**	**0.986**

Table 2. Runtime of VSS for industrial benchmarks.

Example	**Profile**			**Runtime Distribution**				
	PI	**PO**	**Reg**	**scl**	**lcorr**	**ssw**	**dsec**	**mapping**
Ex 1	135	128	16864	1.47	43.50	931.42	2062.74	102.12
Ex 2	8927	10761	26246	3.00	44.19	1018.85	1343.87	67.01
Ex 3	2042	485	892	0.17	0.27	0.94	1.31	4.23
Ex 4	665	780	5886	0.73	10.48	515.06	612.06	110.72
Ex 5	966	1432	12730	1.55	218.10	1005.17	1909.29	85.20
Ex 6	4143	6625	8081	1.14	1.79	16.79	10.83	57.16
Ex 7	68	64	265	0.14	0.04	5.66	10.31	12.29
Ex 8	6481	5864	3003	0.60	13.85	38.58	43.53	42.89
Ex 9	53	49	5215	0.59	6.52	310.02	376.20	90.40
Ex 10	2905	3608	41657	8.91	2517.73	4972.89	10210.98	159.79
Ex 11	13204	10913	11441	1.76	58.83	270.19	418.84	129.33
Ex 12	164	104	2784	0.35	3.31	34.64	52.70	56.07
Ex 13	25	329	2786	0.40	3.76	61.85	132.50	98.30
Ex 14	67	43	722	0.20	0.11	11.86	20.50	23.18
Ex 15	4822	4101	20223	4.90	918.94	3582.97	6176.54	415.22
Ex 16	113	224	2154	0.22	4.19	32.05	60.59	19.00
Ex 17	1903	2077	13263	1.37	127.37	1216.43	2529.66	216.86
Ex 18	1976	1880	7019	0.69	4.01	59.12	134.31	33.45
Ex 19	141	165	2540	0.34	1.94	28.49	34.17	47.59
Ex 20	3783	4188	23517	3.75	471.20	3255.21	4520.64	165.88
Geomean				**0.84**	**11.81**	**143.51**	**223.10**	**62.72**

Table 3. Comparison of monolithic and partitioned VSS.

	Profile			**scl + lcorr + ssw + if**					**scl + lcorr + ssw -P 9000 + if**				
	PI	**PO**	**Reg**	**Time1, s**	**Time2, s**	**Reg**	**LUT**	**Level**	**Time1, s**	**Time2, s**	**Reg**	**LUT**	**Level**
Ex 1	135	128	16864	976.39	1078.51	13984	21952	3	606.91	706.83	13984	21952	3
Ex 2	8927	10761	26246	1066.04	1133.05	11976	21611	8	503.51	580.19	11976	21683	8
Ex 5	966	1432	12730	1224.82	1310.02	12013	17570	6	919.78	1023.01	12017	17609	6
Ex 10	2905	3608	41657	7499.53	7659.32	38553	31098	8	3290.8	3519.19	38553	31146	8
Ex 11	13204	10913	11441	330.78	460.11	10383	23831	16	175.47	316.49	10383	23859	16
Ex 15	4822	4101	20223	4506.81	4922.03	17634	46603	10	2767.12	3224.00	17689	46746	10
Ex 17	1903	2077	13263	1345.17	1562.03	12790	21916	5	902.86	1126.70	12790	22010	5
Ex 20	3783	4188	23517	3730.16	3896.04	21865	31188	12	1292.08	1484.73	22242	31810	12
Geomean				**1705.30**	**1898.81**	**15844**	**25807**	**7.59**	**922.38**	**1122.48**	**15885**	**25922**	**7.59**
Ratio				**1**	**1**	**1**	**1**	**1**	**0.541**	**0.591**	**1.003**	**1.004**	**1.000**