Recording Synthesis History for Sequential Verification

Alan Mishchenko   **Robert Brayton**

Department of EECS, University of California, Berkeley

{alanmi, brayton}@eecs.berkeley.edu

Abstract

Performing synthesis and verification in isolation has two undesirable consequences: (1) *verification runs the risk of becoming intractable, and* (2) *strong sequential optimizations are not applied because they are hard to verify. This paper develops a methodology for sequential equivalence checking using feedback from synthesis. A format for recording synthesis information is proposed. An implementation is described and experimentally compared against an efficient general-purpose sequential equivalence checker that does not use synthesis information. Experimental results confirm expected substantial savings in runtime of equivalence checking for large designs.*

1 Introduction

In this paper, we propose a methodology shift to a synthesis transparent process, which records and uses the synthesis history in an efficient way. It can promote the use of sequential synthesis and enable a scalable verification of the result.

Sequential synthesis can result in considerable reductions in delay (e.g. see [19]) and area; however, it is mostly avoided for reasons of non-scalability of both synthesis and verification. To circumvent this, we believe that sequential synthesis and verification must go hand-in-hand to make sequential synthesis acceptable, and propose a way to make this happen.

General sequential equivalence checking (GSEC) of two FSMs is PSPACE complete; however, if synthesis is restricted by one set of combinational transformations followed by one retiming or vice versa, the problem is provably simpler. On the other hand, iterating retiming and combinational transformations makes the problem again PSPACE complete [12], even though this is a very restricted set of sequential transformations. Also, as in the case of combinational equivalence checking (CEC) [18], in practice the problem becomes simpler if there are structural similarities between the two circuits to be compared.

The current work has similarities with the following two approaches in the literature. Van Eijk [9] derived an inductive invariant, constructed by a fixed point process, consisting of a set of equivalences between signals in the two circuits. This invariant characterizes a superset of the reachable states of the product machine. Bjesse [4] and Case [7] extended this to an invariant composed of implications, which can give tighter approximations.

Such methods are dependent on the particular implementation structures of the two machines being compared because equivalences or implications can be stated only between existing signals. To overcome this limitation, Van Eijk proposed creating additional signals, without any fanout, which might be useful in establishing additional equivalences. His proposal involved adding nodes which could be obtained by retiming. These signals approximate the reachable state space, thereby simplifying SEC, but do not guarantee that the invariant derived is sufficient to prove sequential equivalence.

Mneimneh et. al. [22] looked at the problem of one retiming and one set of combinational logic transformations (in either order)

and proposed a retiming invariant composed of a conjunction of functional relations among latch values derived from atomic retiming moves.

We address the problem when one machine is derived from the other by a sequence of more general transforms, which may include retiming, combinational synthesis, merging sequentially equivalent nodes, and performing window-based sequential synthesis with don't-cares. We propose to record the synthesis history which will provide the extra signals to aid verification. In contrast to van Eijk, our history aided verification approach (HSEC) can be characterized as follows:

- All nodes created during synthesis are recorded, instead of adding a set of ad-hoc signals.
- Each synthesis step records a sequential equivalence that *should* hold if the implementation of the synthesis algorithm is correct. A side benefit is that if an equivalence does not hold, the implementation is incorrect and the source of the error can be isolated.
- The invariant is the set of all equivalences recorded.
- The invariant is sufficient to prove sequential equivalence of the two machines by induction without counter-examples.
- The invariant can be proved easily by proving each equivalence, one at a time. The proofs are local and hence fast, and can be done in parallel.

Section 2 surveys the background. Section 3 shows how to efficiently record the history of synthesis by integrating two AIG managers. Section 4 details the use of the recorded history in sequential verification. Section 5 discusses other uses of a recorded history. Section 6 reports experimental results and Section 7 concludes the paper and outlines future work.

2 Background

2.1 Boolean networks

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. 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 subset to the POs passes through n. Informally, the MFFC of a

node contains all the logic used exclusively by the node. When a node is removed, the logic in its MFFC can be removed.

Merging node n onto node m is a structural transformation of a network that transfers the fanouts of n to m and 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 if its fanin cone does not contain any other node of the same class. In this work, the representative is the node of the class that appears first in a topological order.

There are different forms of *sequential equivalence* for FSMs [23]. We use the traditional notion, which states that two FSMs are equivalent if they produce the same output sequences for the same input sequence starting from their two initial states.

2.2 And-Inverter Graphs

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. *Structural hashing* of AIGs ensures that, for each pair of nodes, all constants are propagated and there is at most one AND node having them as fanins (up to permutation). Structural hashing is performed by one hash-table lookup when AND nodes are created and added to an AIG manager. When an AIG is incrementally rehashed, the changes are propagated to the fanouts, which may lead to rehashing large portions of AIG nodes.

The *size* (area) of an AIG is the number of its nodes; the *depth* (delay) is the number of nodes on the longest path from the PIs to the POs. The goal of AIG optimization by local transformations of an AIG is to reduce both area and delay.

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. Previous work on sequential AIGs [1][5] applies on-the-fly forward retiming to the registers along with the combinational structural hashing of the AIG nodes.

However, in this paper, we use *simplified sequential AIGs* where registers are represented traditionally as additional terminal nodes of the AIG. An additional data-structure identifies the I/O pair associated with a register's input and output. The PIs and register outputs are called *combinational inputs* (CIs) and the POs and register inputs are called *combinational outputs* (COs). Although mostly representing the combinational logic, simplified sequential AIGs are still suitable for sequential transformations. For example, for retiming, the operation is decomposed into individual register moves. Each move adds new registers to the register boundary while the old registers are removed.

In this paper, the registers are assumed to have a fixed binary initial state[1]. 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 special register that produces 0 in the first frame and 1 afterwards.

2.3 SAT Sweeping and Induction

Combinational SAT sweeping is a technique for detecting and merging nodes that are equivalent up to complementation in a combinational network [13][15][17][18]. SAT sweeping is based on 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 determine the status of 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 can be used as a robust combinational equivalence checking technique and as a building block in k-step induction [4].

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 to be verified. BMC can be implemented as a combinational SAT sweeping applied to several unrolled timeframes with initial state applied in the first frame.

k-step *induction* over timeframes is a method for proving sequential properties, such as sequential equivalence of two nodes in the network [9]. A property or a set of properties are proved inductively if the following two cases hold:

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

A SAT-based *inductive prover* [4] is based on simulation and combinational SAT sweeping [18]. Speculative reduction [21] is a key ingredient of an efficient inductive prover because it reduces the runtime by several orders of magnitude and allows sequential SAT sweeping to work for large industrial design. Basically, it uses the simple device of moving all fanouts of a set of candidate equivalent nodes to one representative of the class.

Sequential SAT sweeping is similar to the combinational one, except that it detects and merges sequentially equivalent nodes[2]. In general, combinationally equivalent nodes are also sequentially equivalent, but not vice versa. Thus, it is helpful to apply combinational SAT sweeping before sequential sweeping. The implementation of sequential SAT sweeping uses k-step induction and an efficient implementation makes use of a SAT-based inductive prover.

3 Recording Synthesis History

AIGs are used increasingly in CAD tools as a unifying data structure for applications dealing with logic synthesis and formal verification. As a circuit representation, AIGs provide uniformity, fast manipulation, low memory requirements, straight-forward construction for both logic networks and mapped netlists, and the possibility of combining them with efficient simulators and SAT solvers, leading to a *semi*-canonical representation that can replace BDDs in many applications [17].

In the context of AIG-based synthesis, recording synthesis history can be done using two AIG managers: a Working AIG (WAIG), which represents the current state of the synthesis, and a History AIG (HAIG), which records all AIG nodes ever encountered during synthesis.

The following rules are followed in a WAIG:

- New logic nodes are added as synthesis proceeds.
- Old logic cones are periodically replaced by new logic cones. When this happens, (a) the old root node is replaced by the new root node, and (b) the fanouts of the old root are transferred to be fanouts of the new root.
- Nodes without fanout (such as the old root) are immediately removed. This helps maintain accurate metrics (node count, logic depth, etc)

---

[1] A motivation of this restriction for industrial designs is given in [2].

[2] The nodes are *sequentially equivalent* if they compute the same value, up to complementation, in all states reachable from the initial state.

The following rules are followed in a HAIG:

- New logic nodes are added as synthesis proceeds.
- Each time a new node is created in the WAIG, a corresponding node is either found or created in the HAIG, **and** a link between the two nodes is established using procedure **setWaigNodeMapping**.
- Old nodes are **not** removed and fanouts are **not** transferred.
- When a node replacement is performed in the WAIG, the two corresponding nodes in the HAIG are **linked** (indicating that they should be sequentially equivalent) using procedure **setHaigNodeMapping.**

Thus two node mapping are supported in a WAIG / HAIG pair:

- Each WAIG node points to a corresponding HAIG node, which was created when the WAIG node was created.
- Some of the HAIG nodes point to other HAIG nodes. This node mapping is created between the corresponding HAIG nodes when a WAIG node is replaced by another WAIG node. The resulting pair of HAIG nodes should be sequentially equivalent if synthesis is correct. These equivalences will be proved during HAIG-based verification, as described in Section 4.

Table 1 establishes a correspondence between the AIG procedures of the WAIG and HAIG. These are the only ones needed for implementing any sequential synthesis algorithm.

Table 1. Relation between WAIG and HAIG procedures.

Working AIG	History AIG
aigManagerCreate (the first call)	**aigManagerCreate**
aigManagerCreate (other calls)	do nothing
aigManagerDelete (other calls)	do nothing
aigManagerDelete (the last call)	**aigManagerDelete**
aigNodeCreate	**aigNodeCreate** and **setWaigNodeMapping**
aigNodeReplace	**setHaigNodeMapping**
aigNodeDelete_recursive	do nothing

The first four lines of Table 1 describe what happens when the WAIG is created and deleted. At the first creation of WAIG, the HAIG manager is created also. On subsequent duplications of the WAIG, the HAIG is unchanged, but the CIs/COs of the new WAIG are remapped to point to the CIs/COs of the HAIG. On the last deletion of any associated WAIG, its HAIG is deleted also.

When a WAIG node is created, a corresponding HAIG node is created and put in correspondence with the WAIG node. When one WAIG node replaces another WAIG node, nothing is done in the HAIG, except establishing the mapping between the corresponding HAIG nodes. Finally, when a WAIG node is recursively deleted, the HAIG remains unchanged.

3.1 Recording combinational synthesis

Recording the history during combinational synthesis involves three steps shown in Figure 3.1. First, logic cone A is re-synthesized, and a new logic cone B is constructed. Note that at this point B has no fanouts. Both cones are present in both the WAIG and HAIG because creating a new WAIG node always results in creating a matching HAIG node. Second, the fanout of logic cone A is transferred to logic cone B in the WAIG. The HAIG is unchanged, except the mapping (indicating equivalence) is established between the old root and the new root in the HAIG. Finally in the WAIG, logic node A is removed and subsequent

new logic may be constructed in the WAIG on top of the new logic cone. No nodes are removed from the HAIG. Subsequent new logic is constructed in the HAIG on top of a new logic cone.

3.2 Recording retiming

Retiming [14] can be decomposed into forward and backward retiming. Each of these retimings can be decomposed into atomic register moves. An atomic move involves transferring registers forward or backward over one AIG node. In forward retiming, the initial state of the new register is trivial to compute. In backward retiming, the initial state is typically computed by formulating a SAT instance. If the SAT instance is satisfiable, the computed initial state is assigned to the new register.
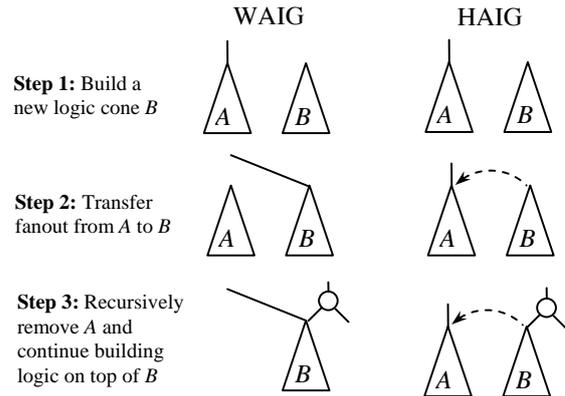

Figure 3.1. Example of history recording in WAIG and HAIG.

Individual register moves are recorded similarly to recording combinational synthesis. In this case, the role of the combination logic cones A and B is played by the AIG node before and after retiming, as shown in Figure 3.2. Note that, in the case of retiming, the equivalence pointers in the HAIG connecting A and B are "asserting" sequential equivalence. Also, note that sequential transformations, like retiming can create new registers which create new CIs / COs pairs in the HAIG.
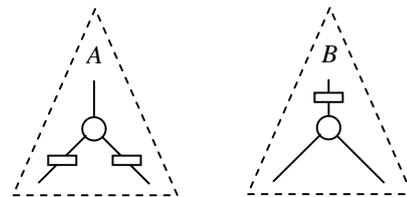

Figure 3.2. Logic cones for one forward retiming move.

3.3 Recording window-based transformations

To ensure scalability, some synthesis transformations are applied to a node or a group of nodes in the context of a *window* rather than the whole network. A window is computed using a set of user-specified parameters, such as a limit on the number of levels of logic to be included on the fanin/fanout side of the node(s), a limit on the window size, and the presence and length of reconvergent paths or sequential loops subsumed in the window. For a overview of windowing, see [20].

A key to recording window-based transforms is to record the whole logic structure of the window after the transform and only assert in the HAIG, sequential equivalence of the window's outputs before and after the transformation. Corresponding internal nodes may not be equivalent if don't cares were used.

3.4 Recording transformations involving ODCs

Combinational or sequential synthesis may involve the use of observability don't-cares computed for a node or a group of nodes. In this case, nodes after synthesis may have different Boolean functions in terms of the CIs. Such nodes cannot be recorded as equivalent to the original ones in the HAIG. However, for the computation of ODCs to be scalable, there always exists a scope, in which the functionality is preserved. This may include a window, a timeframe, or the whole sequential circuit. In all cases, the primary outputs of the scope should be sequentially equivalent before and after the ODC-based synthesis, and can be recorded as in the case of windowing.

3.5 Recording sequential SAT sweeping

When a circuit is transformed by sequential SAT sweeping, the nodes belonging to an equivalence class are merged onto the class representative. Typically, this operation computes many equivalences (or inverted equivalences) at once. In the implementation, the classes are computed first and then the AIG is duplicated while substituting (in a corresponding polarity) the representative for each node in the equivalence class. The pseudo-code of this procedure is shown in Figure 3.5.

New HAIG nodes are created inside procedure **aigAnd**. The mapping of new HAIG nodes into equivalent old HAIG nodes is set by the procedure **setHaigNodeMapping.** This is the same procedure that is called inside **aigNodeReplace**. The pseudo-code is listed to clarify exactly how this is done.

```
node aigNodeCopyWithEquivalences( aig B, node n, classes C )
{
        // if n is already visited, return its copy
        if ( n->copy != NULL )
                return n->copy;
        // if n belongs to an equivalence class, return its representative
        r = getRepr( C, n );
        if ( r != NULL ) {
                if ( n has the same polarity as r )
                        return r;
                else
                        return aigNot(r);
        }
        // solve the problem for fanins of n
        m0 = aigDuplicationWithEquivalences( B, n->fanin0, C );
        m1 = aigDuplicationWithEquivalences( B, n->fanin1, C );
        // create the copy, save it in the node, and return
        n->copy = aigAnd( m0, m1 );
        setHaigNodeMapping( n, n->copy );
        return n->copy;
}
aig aigCopyWithEquivalences( aig A, classes C )
{
        // start the new AIG manager
        aig B = aigStart();
        // clear the copy pointers for all nodes in the old AIG
        for each object n of aig A
                n->copy = NULL;
        // create combinational inputs and make old nodes point to them
        for each combinational input n of aig A
                n->copy = createCi( B );
        // recursively construct the AIG from the combinational outputs
        for each combinational output n of aig A
                aigNodeCopyWithEquivalences( B, n, C );
        return B;
}
```

Figure 3.5. Copying AIGs while merging equivalent nodes.

4 Using the HAIG for Verification

A history AIG (HAIG) is an AIG containing the original version of the design, the final one, and all the intermediate logic derived during synthesis. It is a sequential circuit in every sense (e.g. an initial state for every register), but with a lots of redundancy. Like a miter, it contains both the original and final designs. Sequential verification of the original against the final one can be performed by proving equivalence of all candidate pairs of HAIG nodes recorded during synthesis.

4.1 Theory

Definition: Unlike combinational synthesis, a window in sequential synthesis can cross the register boundary several times. The *sequential depth* of a window-based sequential synthesis transform is the largest number of registers on any path from an input to an output of the window. Currently, loops in a window are not allowed.

Theorem 1: If transforms recorded in a HAIG have sequential depth less or equal to k, the equivalence classes of the HAIG nodes can be proved by k-step induction.

Theorem 2: If the inductive proof of the candidate equivalences in a HAIG passes without counter-examples, then all synthesis steps have been performed correctly (which implies that the original design and final design are sequentially equivalent).

The proof of Theorem 1 is straightforward. The formal proof of Theorem 2 can be found in [5].

4.2 Implementation

Figure 4.2 shows the pseudo-code of a simple inductive prover used to verify the candidate equivalences recorded in the HAIG. This prover is much simpler than the general-case prover [9][4][21][16] because it does not rely on the detection and refinement of candidate equivalence classes. The candidate classes are already recorded in HAIG using procedure **setHaigNodeMapping**, and would rarely need to be refined.

```
status inductiveVerification( aig HAIG, int k )
{
    // run BMC for k-1 initialized timeframes
    status = performBMC( HAIG, k-1 );

    // return the status of sequential verification after BMC
    if ( status == "encountered a counter-example" )
            return ID of the synthesis transform that failed BMC;

    // do speculative reduction for k-1 uninitialized timeframes
    aig HAIG_SR = speculativeReduction( HAIG, k-1 );

    // derive SAT solver containing CNF of the reduced timeframes
    solver S = transformAIGintoCNF( HAIG_SR );

    // check candidate equivalences in k-th timeframe
    status = performSatSweepingWithConstraints( S, HAIG );

    // return the status of sequential verification after SAT sweeping
    if ( status == "encountered a counter-example" )
            return ID of the synthesis transform that failed induction;
    return "equivalence check succeeded";
}
```

Figure 4.2. Simple inductive prover to verify HAIG.

The simple inductive prover makes use of speculative reduction [21], resulting in substantially reduced runtime. There is no need for iterative refinement of the equivalence classes because, if synthesis was performed correctly, counter-examples are never produced. If a counter-example is detected, the ID of the corresponding synthesis transform is returned for help in debugging the synthesis code. We note that even the $k+1^{st}$ copy

can be speculatively reduced. Further, each equivalence in this copy can be solved in parallel.

It is significant that the prover that can used in verification of the HAIG can be so simple because this inductive prover should not be same as that used in synthesis, otherwise the same bug may appear in both and make it not observable.

5 Other Uses of a HAIG

A HAIG can be used in several other applications, e.g., to improve the quality of technology mapping or to perform incremental changes to netlists after physical design (ECO).

Using synthesis history to overcome structural bias inherent in cut-based structural mapping leads to substantial improvements in delay and area [6]. It was shown that further iterating HAIG-based synthesis and tech-mapping tends to gradually improve the quality of mapping. This happens because the logic structure of the AIG after each technology mapping is recorded in the HAIG, and the AIG tends to evolve gradually to reflect the implementation technology. In [19], it was shown that sequential mapping combining technology mapping and retiming [24] can be extended to use the HAIG in a way similar to that for combinational mapping [6].

Another application could be design debugging after physical synthesis, which requires tracing some logic gates back to the lines of the original HDL code, which produced them. For such application, additional APIs would allow the designer to use the HAIG to efficiently iterate through the synthesis steps forward or backward, and trace the dependence of a node in the final AIG to the original source code. Another application may explore the impact of a particular synthesis transform on the final result and possibly incrementally undo that transform to improve the result.

6 Experimental results

History recording and HAIG-based sequential verification are implemented in ABC [3]. The SAT solver used is a modified version of MiniSat-C_v1.14.1 [8]. Experiments were run on an IBM ThinkPad laptop with 1.6GHz Intel CPU and 2Gb RAM.

The benchmarks used for the experiments were taken from the ITC'97, ISCAS'89, and IWLS'05 benchmark suites [10]. The results of synthesis were verified with and without using history.

Table 2 shows the number of primary inputs (column "PIs"), primary outputs (column "POs"), and registers (column "Reg").

Additionally, Table 1 shows the runtimes of three operations:

- Column "Sig Corr" stands for the signal-correspondence computation (command *ssw*), which detects and merges sequentially equivalent nodes and records the history of this transformation. This is the only type of synthesis that was used in these experimental results.
- Column "GSEC" is an efficient integrated general-case sequential equivalence checker based on *k*-step induction (command *dsec*). The command was developed along the lines suggested in [9][4][16][21]. This checker does not rely on the information from synthesis.
- Column "HSEC" is the proposed HAIG-based verification. The runtime in this column includes the time needed to construct the speculatively reduced model for the HAIG plus the runtime of the SAT solver needed to prove each pair of the HAIG nodes that are deemed equivalent.

Row "Ratio" at the bottom of Table 2 shows the ratios of the averages of runtimes in the corresponding columns to the average runtime of the last column "HSEC". The ratios show that the HSEC is two orders of magnitude faster than GSEC. Moreover,

the runtime of HSEC is negligible, compared to the runtime of Sig Corr synthesis.

Memory requirement for a general AIG manager are roughly 32 bytes per AIG node stored. However, a HAIG can get by with 8 bytes per node. The largest benchmarks in the set had about 20K AIG nodes. Assuming two copies of the circuit stored in a HAIG, yields 2 * 20,000 * 8 = 320Kb. AIGs also lead to significant compaction as shown in the program AIGer. The runtime of HAIG recording is negligible.

The final version of the paper will contain experimental results when many other types of synthesis transforms are used. We conjecture that the ratio of the runtimes of HSEC versus GSEC will be close to those reported in Table 2 because speculative reduction will still be effective and the individual equivalence proofs will still be local.

7 Conclusions and future work

We proposed a transparent synthesis process, which efficiently records the history of synthesis transformations. We showed how this history can simplify sequential verification. We proposed a simple elegant format for storing a history as an AIG and described how this can be done easily by orchestrating computations in two related AIG managers. Finally, we demonstrated that the use of a history leads to substantial savings in the runtime for sequential verification, compared to the runtime of an efficient but general-purpose equivalence checker.

Typical questions and concerns about a history-based sequential verification process are:

1) Can't incorrect information be passed inadvertently from the synthesis tool to the verification tool?
2) Might the same bugs in the synthesis tool also exist in the verification tool, thereby cancelling each other out and leading to false positives?
3) Won't the memory required to record the history explode on large examples?
4) If a synthesis tool does not use AIGs can one still use this methodology?

First, we emphasize that the synthesis history is used simply as a set of hints for verification. Every step recorded in the history must be proved, and should be proved using a *different* prover compared to the one used in synthesis. Fortunately the inductive prover needed in HSEC is much simpler than in GSEC because induction for a HAIG should succeed without counter-examples. A simple HAIG prover in ABC is only about 100 lines of code (not counting the AIG package and the SAT solver), which is much more than about 2000 lines of code needed to implement a general-case inductive prover. The simplicity of the HSEC prover makes it easy to debug and more reliable. Finally, at 8 bytes per node, memory requirements for a HAIG are very light, can be compacted significantly, and can be stored on disk without cache interference during history recording. Finally, we envision a history package based on AIGs which is a stand-alone module and can be called by any synthesis tool. Although this methodology might not be acceptable for commercial synthesis companies, it can be incorporated in in-house tools where exposing synthesis know-how may not be a problem.

Also, the absence of counter-examples ensures fast runtimes of the HSEC solver. This is supported by experimental results. In contrast, a GSEC prover for large industrial circuits is much slower because of the runtime spent generating and simulating counter-examples, and refining the equivalence classes. A counter-example would be extremely rare and identify an incorrect implementation of a synthesis transformation.

HSEC is fast because speculative reduction effectively reduces the HAIG to a single copy of the original circuit, except for the miters that are necessary to state the equivalences. In other words, if these signals are swept away by removing the miter outputs, the HAIG will collapse to a single copy of the original circuit. Even in the last, $k+1^{st}$ timeframe, the circuit can be speculatively reduced. For further speed, all equivalences can be proved in parallel and in the rare case that one does not hold, the first one in topological order identifies a bug in the synthesis code. This is sufficient for debugging the synthesis code.

Although we have not explored other ways of recording synthesis history, the use of AIGs seems to provide an elegant method for doing this. AIGs are becoming increasingly accepted in both synthesis and verification communities, efficient AIG packages are being developed and improved, and AIGs are being used as an intermediate format for circuit logic representation.

Future work in this area will include:

- Polishing the HAIG interface and releasing it as a stand-alone package ready for integration into non-AIG based synthesis tools.
- Supporting a larger set of sequential transforms, in particular, window-based transforms, such as re-encoding, ODC-based resynthesis, etc, for which history is currently not recorded.
- Conducting extensive experiments on industrial benchmarks while recording long sequences of synthesis transforms.
- Exploring the potential of using a partial HAIG. In particular, (a) developing methods to record a minimal history needed to ensure inductiveness and (b) investigating if some history information can be used to speed up the general-case SEC.

Acknowledgements

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] R. Brayton and A. Mishchenko, "Scalable sequential verification", ERL Technical Report, EECS Dept., UC Berkeley, 2007. http://www.eecs.berkeley.edu/~alanmi/publications/2007/tech07_ssv.pdf

[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] M. Case, A. Mishchenko, and R. Brayton, "Inductively finding a reachable state space over-approximation", *Proc. IWLS '06*, pp. 172-179.

[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] C. E. Leiserson and J. B. Saxe. "Retiming synchronous circuitry", *Algorithmica*, 1991, vol. 6, pp. 5-35.

[15] 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.

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

[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*, pp. 354-361.

[20] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis", *Proc. IWLS '07*, pp. 358-364. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fpga08_imfs.pdf

[21] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. "Exploiting suspected redundancy without proving it". *Proc. DAC'05*, pp. 463-466.

[22] M. Mneimneh and K. Sakallah, "REVERSE: Efficient sequential verification for retiming", *Proc. IWLS '03*, pp. 133-139.

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

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

Table 2. Register count, area, and delay after the proposed synthesis for academic benchmarks.

Example	Statistics			Runtime, sec		
	PI	PO	Reg	Sig Corr	GSEC	HSEC
b15	36	70	449	6.26	11.79	0.13
b17	37	97	1415	11.32	19.23	0.16
s13207	31	121	669	0.22	0.31	0.05
s35932	35	320	1728	1.83	5.43	0.07
s38417	28	106	1636	2.65	8.59	0.12
s38584	12	278	1452	1.12	4.11	0.05
systemcaes	260	129	670	1.42	2.81	0.10
tv80	14	32	359	0.94	1.28	0.05
usb_funct	128	121	1746	12.65	29.64	0.08
wb_conmax	1130	1416	770	17.08	2.03	0.05
wb_dma	217	215	563	0.49	1.29	0.15
aes_core	259	129	530	1.25	1.65	0.12
ethernet	98	115	10544	3.68	19.66	0.08
Ratio				60.49	90.87	1.00