Cut-Based Inductive Invariant Computation

Alan Mishchenko[1] **Michael Case**[1,2] **Robert Brayton**[1]

[1] Department of EECS, University of California, Berkeley, CA

[2] IBM Systems and Technology Group, Austin, TX

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

Abstract

This paper presents a new way of computing inductive invariants in sequential designs. The invariants are useful for strengthening inductive proofs in difficult unbounded model checking instances. The proposed computation is scalable and can flexibly trade computational effort for the expressiveness of invariants proved. Experimental results on several benchmark families show that the proposed strengthening proves many hard properties, unsolved by other model checkers. The implementation is publicly available in the synthesis and verification system ABC. Runtimes are reasonable: the hardest problem with 5K primary inputs, 3K registers, and 64K AIG nodes takes 6 minutes.

1 Introduction

Model checking [11][28] safety and liveness properties involves proving that a safety property holds on all reachable states [2]. Many safety properties can be verified by proving the property on an inductive superset of reachable states. If the superset can be represented compactly, then such a method is easier and more scalable than deriving the exact set of reachable states. Finding such an inductive superset is called *inductive strengthening*.

This paper introduces a new way of deriving and proving an additional inductive property, or *invariant*, that (1) can be effective for inductive strengthening and (2) leads to a flexible and scalable computation, trading computational effort for increased expressive power of the invariant derived. As a byproduct, the same invariant can be used as a source of external don't cares for circuit restructuring in sequential logic synthesis.

The proposed invariant consists of a set of clauses derived using *m*-input cuts of nodes in the sequential circuit. A cut is a boundary separating the node from the primary inputs and register outputs. Therefore the invariant is expressed in terms of groups of adjacent nodes in the network.

This computation illustrates the synergy between logic synthesis and verification [8]. In the past, external don't cares for logic synthesis were obtained by computing the set of unreachable states characterized by a function of the register outputs. However, even if these can be computed, scalability motivates the use of windowing where the computation is temporarily restricted to a group of nodes surrounding a node (or another window) to be optimized. Thus, to use the external don't cares, they must be projected onto the inputs of the window. Therefore, the useful unreachable states have nontrivial projections onto the inputs of the windows, which form cuts in the network. The new idea is to skip computing the unreachable states and directly compute its projections onto various cuts. We expected this invariant to be useful for inductive strengthening, which turned out to be true.

Induction [15][5][12] is a practical model checking method, applicable to large designs whose size and logic complexity often causes other methods (such as BDD-based reachability, interpolation, localization, etc) to fail. A property is *inductive* if it satisfies two conditions: (*base case*) it holds in the initial state, and (*inductive case*) if it holds in a state, then it holds in all states reachable from that state in one transition. Induction is scalable because both the base and inductive cases can be formulated as incremental instances of Boolean satisfiability (SAT) [12], which can be solved efficiently using modern SAT solvers [13].

The rationale behind inductive strengthening is that a set of properties can be inductive when the individual properties are not. This is because the conjunction of a set of properties characterizes a smaller state space and may exclude states that fail induction for individual properties.

In our method, the properties are clauses plus a target property, and the groups of variables participating in the clauses are derived using efficient *m*-cut computation, which is adopted from LUT-based technology mapping [24]. It avoids exhaustive cut enumeration [27] and computes only a small subset of useful cuts using priority heuristics similar to those in [14].

The initial set of candidate clauses is detected using two types of random simulation, combinational and sequential. Minterms at a cut that appear under combinational but never under sequential simulation are recorded. A candidate clause is the complement of such a minterm. This initial candidate set is iteratively refined using SAT-based induction. The greatest fixed-point yields the proposed inductive invariant, which over approximates reachable states if the initial state satisfies the invariant.

To make this computation efficient, a flexible framework has been developed for trading the number and expressiveness of the set of clauses for computation time. The clauses are proved in batches, each of which successively refines the already computed approximation of the reachable states. The process is stopped when the target property becomes inductive, or when a sufficient number of clauses is successfully proved.

Scalability is achieved by using heuristics for candidate clause generation and filtering. One heuristic limits clauses to those derived for cuts a few logic levels from the register outputs. Inductive proofs for such shallow clauses can be processed efficiently by partitioning the design and solving partitions in parallel without compromising the completeness of the result. A similar approach was used in [25] to partition inductive proofs for register correspondence.

The rest of the paper is organized as follows. Section 2 describes further background and relations with previous work. Section 3 describes the algorithms used for inductive strengthening. Section 4 discusses application to logic synthesis. Section 5 reports experimental results. Section 6 concludes the paper and outlines future work.

2 Background and Related Research

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, design and circuit 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. It is assumed that each node has a unique integer called *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 *topological order* of nodes in the network is an order in which any node follows all its fanins.

If the network is *sequential*, the memory elements are assumed to be D-flip-flops. The terms memory elements, flop-flops, and registers are used interchangeably in this paper. The registers are assumed to have a fixed binary initial state. If a register has an unknown or a don't-care initial state, it can be transformed to have 0-initial state by adding a new PI and a MUX controlled by a special register that produces 0 in the first frame and 1 afterwards. So without loss of generality, we consider only registers with a 0 initial state. *The set of reachable states* includes the initial state and all the states reachable from it by any input sequence.

An *And-Inverter Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters represented as complemented attributes on the edges. *Structural hashing* of AIGs ensures that all constants are propagated and that each AND-node is structurally unique, that is, there is no other node having the same 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 modified and incrementally rehashed, the changes are propagated to the fanouts, which may lead to rehashing large portions of AIG nodes.

A *cut C* of node *n*, called *root*, is a set of nodes of the network, called *leaves*, such that each path from a PI to *n* passes through at least one leaf. A cut is *m-feasible* if its size does not exceed *m* and is *dominated* if it contains a cut of the same root.

A *literal* of a Boolean variable is the variable or its complement. Given a set of variables *x* (e.g. the set of leaves of cut *C*), a *minterm* is a product of literals, one for each variable in the set. The complement of a minterm is a *clause*. A product of clauses is a *Conjunctive Normal Form* (CNF). *Boolean satisfiability* (SAT) is the problem of determining whether a CNF represents the constant 0 function. *SAT solving* is the process of solving this problem. *SAT-based method* is a method that reduces a given problem to SAT and solves it using a SAT solver.

Simulation is a way of computing node values in a circuit under given input values. *Random simulation* uses random or biased random input values. Simulation assigns values at the inputs and evaluates the internal nodes in a topological order. Simulation is typically performed *bitwise*, where 32 or 64 input patterns are evaluated using a single machine operation. Simulation information of a node is stored in a bit-string composed of many machine words. It is computed by a sequence of bitwise operations using the simulation information of the node's fanins.

A *combinational invariant* is a relation among arbitrary signals in the network that holds in all states. A *sequential invariant* is a relation that holds in all reachable states. A sequential invariant can be seen as a characterization of a set of states for which it holds. This set includes the set of reachable states and possibly some unreachable states. A typical sequential invariant is equality among two registers outputs that does not hold combinationally (in all states) but holds sequentially (in all reachable states).

A *candidate sequential invariant* is an invariant that has not yet been proved (e.g. by induction or interpolation) but is suspected to hold (e.g. after several rounds of simulation). Such invariants express properties that should be proved, e.g. mutual exclusion of the values at two primary outputs. *Model checking* focuses on proving user-specified properties.

Induction is often used to prove sequential invariants. Its use for sequential designs was pioneered in [15] and further developed in [5][26][17]. A sequential invariant is *inductive* when: (*base case*) it holds in the initial state, and (*inductive case*) if it holds in a state, then it holds in all states reachable from that state in one transition.

When an invariant is proved inductively, the base and the inductive cases are formulated as SAT instances and solved by a SAT solver. The solution is incremental because each property in the invariant is checked independently. This can be done efficiently using an incremental interface of a modern SAT solver [13]. More details on SAT-based induction can be found in [25].

The set of all reachable states is an inductive sequential invariant. However, not every sequential invariant is inductive. For example, consider an unreachable state *s* that has a transition into it from state *t*. The complement of the minterm composed of register variables representing *s* is a sequential invariant because it holds in all reachable states. This invariant is not inductive because it holds in *t* but not in *s*. The state failing the inductive case (in this case *t*) is called an *induction leak*.

Several ways of strengthening induction are known:

- Extending simple induction to *k*-step induction ($k > 1$) [15].
- Using unique-state constraints [12].
- Using equivalences expressed over register outputs (register correspondence) or over arbitrary signals in the network (signal correspondence) [15][26][17][25].
- Applying signal correspondence after timeframe expansion, hoping this will capture additional equivalences among signals across different timeframes in the original design.
- Using implications of signals in the network [5][9][10].
- Using *p*-th invariants, that is, invariants that hold starting from frame *p* from the initial state ($p > 1$) [16].
- Incrementally computing inductive clauses in terms of register variables using counter-examples to induction [7].

We propose using inductive strengthening, based on generating an invariant in the form of a set of *m*-literal clauses. The method is a generalization of [9], [10], and [7], as shown in Section 3.4.

3 Computing inductive invariants

This section presents a new algorithm for computing inductive invariants in a sequential network. The algorithm is presented for AIGs but it is equally applicable to general logic networks.

The overall pseudo-code of the algorithm is shown in Figure 3. Details are given in the subsections listed in the parentheses.

The computation starts by enumerating for each node a subset of *m*-cuts using procedure **aigEnumerateCuts** (Section 3.1).

Next, two rounds of simulation are performed. For each cut of size *m*, all 2^m value assignments of the cut leaves are considered. Simulation information is used to determine if each assignment is likely to appear only under unreachable states. A clause derived from each such assignment by complementing its literals yields a candidate invariant. A number of candidate clauses are collected and filtered using simulation information in the procedure **aigComputeCandidates** (Section 3.2).

A set of clauses representing the candidate invariant is checked by the base case and then by an iterative refinement procedure **performInductiveCase,** similar to that of van Eijk [15]. When this procedure terminates, the set of remaining clauses, if it is non-empty, represents an inductive invariant. If strengthening is not sufficient (determined by procedure **checkSufficient)**, another round of invariant computation is performed. The candidates considered next are those not contained in the already proved set. As a result, new invariants that are proved provide ever tighter approximations of the state space (Section 3.3).

```
set of clauses computeInvariants( aig N, parameters P )
{
    // compute m-cuts for all nodes
    aigEnumerateCuts( N, P );

    // perform two rounds of simulation
    aigSimulateComb( N, P );
    aigSimulateSeq( N, P );

    // iterate while the set of clauses is not sufficient
    set of clauses S = ∅;
    while ( !checkSufficient( S ) ) {

        // compute candidate clauses
        set of clauses C = aigComputeCandidates( N, P );

        // refine the candidates using the base case
        C = performBaseCase( C, N, P );

        // refine the candidates using van Eijk's loop
        do {
            C = performInductiveCase( C, N, P );
        }
        while ( checkChanges( C ) );

        // add newly proved invariant to the set
        S = S ∪ C;
    }
    return S;
}
```

Figure 3. Pseudo-code for computing inductive invariants.

3.1 Cut computation

For two sets of cuts A and B, the operation $A \Diamond B$ is defined as:

$$A \Diamond B = \{ u \cup v \mid u \in A, v \in B, |u \cup v| \le m \}.$$

Let $\Phi(n)$ denote the set of m-feasible cuts of node n. If n is an AND node, let n_1 and n_2 denote its fanins. $\Phi(n)$ is computed using the sets of cuts of its fanins:

$$\Phi(n) = \begin{Bmatrix} \{\{n\}\} & : n \in \text{PI} \\ \{\{n\}\} \cup \Phi(n_1) \Diamond \Phi(n_2) & : \text{otherwise} \end{Bmatrix}.$$

Performing cut computation for the nodes in a topological order guarantees that the fanin cuts, $\Phi(n_1)$ and $\Phi(n_2)$, are available when the node cuts, $\Phi(n)$, are computed. The set of computed cuts is filtered by removing dominated cuts. This reduces runtime and memory without sacrifiying the expressiveness of cuts computed.

The above complete cut enumeration [25] is practical for small m ($m < 6$). For larger m, the above procedure can be supplemented with a method to compute a subset of all m-cuts meeting some criteria. These cuts are called *priority cuts* [24]. The criterion used to prioritize the cuts for invariant computation is to prefer cuts with a larger average number of fanouts of the leaves of a cut. A similar criterion was used in [14].

3.2 Collecting candidates

Two rounds of simulation are performed, combinational and sequential. *Combinational simulation* assumes random values at the primary inputs and register outputs, which are treated as additional primary inputs. *Sequential simulation* assumes random values at the primary inputs while the register outputs are set to the initial state. This simulation iterates over the circuit several times, setting the register outputs to the register inputs computed on the previous step. The difference between these two types of simulation is that combinational simulation produces values under any state while sequential produces values under reachable states.

Candidate clauses are collected by considering the m-cuts of all nodes in the AIG. Each node has two types of simulated minterms. A cut is analyzed to determine what values appear at the cut inputs. Suppose assignment $x_0 x_1 \ldots x_{m-1}$ appears N times at the cut inputs under combinational simulation but does not appear under sequential simulation. This indicates that this assignment may be produced by N or fewer states that are unreachable from the initial state. Thus, this assignment is likely never produced on the reachable states and the complement of this assignment, the clause $\bar{x}_0 \vee \bar{x}_1 \vee \ldots \vee \bar{x}_{m-1}$, is likely true for all reachable states. All such clauses are accumulated and used as candidates.

It should be noted that this approach misses some combinational minterms due to the fact that combinational simulation is not exhaustive, but this only affects the completeness of the method. Moreover, since such minterms do not readily appear under combinational simulation, they are not likely to substantially refine the characterization of the state space.

Except for small circuits and small values of m, the number of candidate clauses can be large. For example, on a circuit with 1K registers and 15K AIG nodes, there may be 50K candidate clauses computed using the set of all 4-cuts. In such cases, the invariants can be filtered by the following heuristic. The larger is N, the more likely is that the minterm characterizes unreachable states. Our implementation has a user-controlled parameter, which limits the number of the highest-scoring clauses considered. This heuristic plays an important role in selecting useful candidates.

The set of candidate clauses can lead to a stronger inductive invariant if it is supplemented with the candidate clauses expressing one-hotness conditions. These conditions are two-literal clauses involving register outputs and can be easily computed using sequential simulation information. Most of these additional clauses cannot be collected as candidates using cuts because cuts include literals in the vicinity of some node, while one-hotness, if applicable, can relate registers that are far apart.

We found that adding the candidate one-hotness conditions often improves the performance of the algorithm. One reason for this is that many industrial designs use one-hot encoding for at least some of the registers.

3.3 Proving candidates

The well-known van Eijk procedure [15] is used to process the candidates and prove some of them. First, those candidates that do not satisfy the base case are removed. Second, the inductive case is performed by asserting the clauses in the first frame and checking them in the next frame. The counter-examples are used to refine the remaining properties to be proved. The failing clauses are removed and refinement is iterated as long as the set of clauses keeps changing. When a fixed point is reached, the resulting set, if it is non-empty, represents an inductive invariant.

To derive a sufficiently tight invariant, the van Eijk procedure can be applied to one set after another. An invariant proved in a previous run is assumed in the next run. Since the previous clauses form an invariant, there is no need to re-prove them; only new clauses need to be proved. This results in accumulating clauses, which increasingly refine the invariant. New candidate clauses are collected only if they refine the current invariant. If cuts of the given size (*m*) do not yield additional clauses, the cut size can be increased to find new candidates to continue refining the invariant. This strengthening enhances van Eijk's procedure and allows tighter invariants to be found efficiently.

The "sufficiency" of the resulting invariant depends on the application. In model checking, it is sufficient if the invariant implies the target property. In logic synthesis, it is sufficient if it contains "enough" flexibility to do substantial logic restructuring.

In model checking, the procedure can stop as soon as the proved invariant implies the target property. For this, the target property is added to the set of candidate clauses. If the property remains in the fixed point, it is proved. Otherwise, a new set of clauses is considered that provides a tighter approximation of the reached state set and has a better chance to prove the target property.

3.4 Comparison with previous work

For a description of other SAT-based approaches to model checking, refer to [28] and for an overview of recent work in induction strengthening refer to [10][7].

The proposed approach can be seen as a generalization of three previous approaches [9][10][7]. The following is a comparison:

- Computation of *m*-cuts scales better than that of implications between signal pairs because priority cuts [24] only take linear-time in circuit size to compute while computing implications takes quadratic-time [10].
- The *m*-literal clauses have more expressive power because implications used in [9][10] are two-literal clauses.
- Our flexible framework for inductively proving groups of *m*-literal clauses is similar to [9], with novel heuristics to prioritize clauses according to their expressive power.
- The *m*-clauses are computed in terms of internal variables rather than register outputs as done in [7], which increases the expressive power of the invariants.
- The *m*-clause candidates are computed by simulation rather than from counter-examples as done in [7], which is less time-consuming and avoids the risk of not having inductive sub-clauses.
- The inductive proof for *m*-clauses, with the cuts limited to a few levels from the register outputs, can use partitioning similar to [25] which increases the possibility that the proposed approach works for designs of any size.
- Adding signal-correspondence and one-hotness invariants, which was not used in [9][10][7] gives additional strength to the proposed approach.

4 Application to Logic Synthesis

The inductive invariants proved by this method compactly represent unreachable state information useful as flexibility in circuit restructuring during logic synthesis with don't-cares [21].

The following are advantages of this approach compared to using other types of sequential flexibility:

- **Complete set of unreachable states**

  Except for small circuits, the reachable state set is hard or impossible to obtain. BDD-based methods for computing this set mostly fail on circuits with more than a 50-100 registers. Another disadvantage is that, if the unreachable state information represented with BDDs is used in sequential synthesis, sequential equivalence checking (SEC) is very hard because it doubles the number of registers. In contrast, when the proposed invariants are used, sequential verification tends to be easier because the inductive nature of the invariants tends to increase inductiveness of the associated SEC problems.
- **Equivalences in terms of internal signals**

  Signal equivalences in terms of internal signals (signal correspondences) have been shown to be a powerful vehicle for capturing sequential flexibility. Sequential synthesis based on this flexibility can lead to substantial reductions in area and register count [25]. However, the best use of this flexibility for circuit restructuring, is to collapse the equivalent nodes into a single node and remove the others. This reduces the circuit but does not allow for a more fine-grain circuit restructuring afforded by the *m*-cut invariants. This is why signal equivalence should be computed and used as a preprocessing step before using the proposed inductive invariant.
- **Implications in terms of internal signals**

  Signal implications among internal signals provide additional expressive power, compared to signal equivalences and can be useful in logic synthesis [9]. Detection of implications can be done similarly to the proposed invariants, using simulation information. However, *m*-literal clauses are more expressive compared to implications (2-literal clauses). In addition, collecting implications is harder and may require a procedure quadratic in the number of nodes, while collecting *m*-literal clauses is linear when priority cuts are used.

5 Experimental Results

The proposed algorithms are implemented in ABC [1] as command *indcut*. The SAT solver is a modified version of MiniSat-C_v1.14.1 [13]. The workstation used has two dual-core AMD Opteron 2218 CPUs with 16Gb RAM, and runs x86_64 GNU/Linux. Only one core was used in the experiments.

Experiments were performed using several suites of model checking benchmarks: (1) Intel benchmarks from the hardware model checking competition at CAV '07 [4], (2) a set of PicoJava II benchmarks [19], and (3) TIP benchmarks [12]. The model checking competition [4] included three other benchmark suites: (a) the TIP benchmarks, (b) the AMBA benchmarks (all *unsat*), and (c) the L2S benchmarks (9 *unsat* cases). The unsat cases from the latter two suites could be solved easily using signal correspondence (command *ssw*) [25] after combinational synthesis (command *dcompress2*). Since the proposed algorithm is developed as a method to be applied when other methods fail, we do not report its performance on the AMBA and L2S suites.

Command *indcut* was used in all reported experiments with the following default set of parameters: induction depth (K = 1), cut size (M = 4), the limit on the number of candidate clauses collected (C = 5000), the maximum level of the nodes whose cuts are considered (L = 8), the number of times invariant computation was iterated (B = 1).

5.1 Intel benchmarks

This set includes 42 *unsat* model checking benchmarks, out of which only 9 were solved by solvers submitted to the model checking competition [4]. The proposed algorithm solved the complete set of 42 benchmarks in 40 minutes. The detailed results are reported in Table 5.1.

The following notation is used in the table. Column "Example" lists the name of a benchmark. Columns "PI"/"PO"/"Reg"/"AIG"

show the number of primary inputs, primary outputs, registers and 2-input AND nodes in the And-Inverter Graph (AIG) that represented the circuit after fast preprocessing that was done using commands *scl –l* and *dcompress*2 in ABC. The first of these commands performs sequential cleanup, merges registers with identical combinational inputs and sweeps away stuck-at-constant registers. Column "Cut" shows the total number of 4-input cuts computed for the AIG. Columns "Clauses" shows the number of clauses. Subcolumns "Cand" and "Proved" list candidate clauses collected and clauses proved inductively, respectively.

Finally, the runtime in seconds is reported in the last three columns. These columns contain the runtime of preprocessing, the runtime of the inductive case, and the total runtime of command *indcut*. The preprocessing runtime involves cut computation, candidate clause generation, and filtering clauses using the base case of the inductive procedure (bounded model checking).

5.2 PicoJava benchmarks

The complete set of PicoJava benchmarks includes 20 *unsat* problems. After preprocessing with register sweep (command *scl –l*), combinational synthesis (command *dcompress2*) and signal correspondence (command *ssw*), 9 out 20 problems were already solved. This preprocessing for the complete set of 20 problems took 214 seconds.

The remaining 11 benchmarks after preprocessing were solved by command *indcut*. The detailed results are shown in Table 5.2 where the notation is the same as in Table 5.1.

Table 5.2 shows that, for many benchmarks, the computation of candidate clauses was stopped after reaching the limit (5000). Approximately 45% of the candidates were proved inductively. Although the set of proved clauses is incomplete, it was sufficient to imply the target property for all of the considered problems.

5.3 TIP benchmarks

These benchmarks are among the smallest and the most well-studied model checking benchmarks [12]. The original set of 158 testcases includes both *sat* and *unsat* problems. First, this set was filtered by removing all problems provable by signal correspondence with induction depth K = 4 (command *ssw -F* 4) or disproved by BMC of depth 100 (command *bmc –F* 100). This led to a subset containing 51 "hard" TIP problems.

Applying *indcut* with default settings these 51 problems solved 41 of them, with runtime for each benchmark not exceeding 1 second. Interestingly, some of the benchmarks solved by *indcut* could also be proved by signal correspondence with very large induction depth. Thus, *cmu_periodic_N* could be proved by *ssw -F* 96 (*K* = 96) in 30 sec, while *indcut* solved it in 0.2 sec.

The 10 remaining benchmarks not solved by *indcut* are: *cmu_dme*1*_B, cmu_dme*2*_B, irst_dme*4*_B, irst_dme*5*_B, irst_dme*6*_B, nusmv_dme*1-16*_B, nusmv_dme*2-16*_B, texas_two_proc_*6*_E, vis_coherence_*3*_E, vis_coherence_*4*_E*. These benchmarks could not be solved by *indcut* even when we modified the default set of parameters. In all cases, a subset of clauses was proved inductively, but the resulting invariant was not sufficient to imply the target property, while other candidate clauses implying it were not inductive. We believe that none of the model checkers submitted to the model checking competition [4] was able to solve these 10 benchmarks.

6 Conclusions and Future Work

This paper proposes a new method for inductively strengthening model checking of safety properties. The method supplements existing methods and is useful for proving hard *unsat* problems.

In combination with other synthesis and verification algorithms implemented in ABC, the proposed method solved 334 of the 344 benchmarks from the model checking competition [4]. The remaining 10, plus another 27 of the 344 problems solved by the proposed method, were not solved by any of the entrants in the 15 minutes allowed for each example. The hardest Intel example took 6 minutes while the hardest PicoJava example took 10 seconds.

In summary, the contributions of this paper are:

- A new efficient method for expressing candidate invariants using *m*-clauses formulated for the nodes in the circuit.
- A scalable hierarchical approach to proving the candidate invariants, which trades off computational effort for the number and expressiveness of invariants generated.
- Experiments using several benchmark suites to show that the proposed method can solve many difficult problems.

Future work will include:

- Further experiments and fine tuning using benchmarks contributed by industrial collaborators.
- Integrating the induction strengthening engine into robust equivalence and model checkers.
- Using the computed invariant clause sets as don't-cares for circuit restructuring in logic synthesis.
- Performing direct comparison with industrial model checkers.

Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the California Micro Program with industrial sponsors Actel, Altera, Calypto, Intel, Magma, Synopsys, Synplicity, and Xilinx.

References

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

[2] A. Biere, C. Artho, V. Schuppan, "Liveness checking as safety checking". Proc. Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'02), ENTCS, Vol. 66(2).

[3] A. Biere. *AIGER format and toolbox*. http://fmv.jku.at/aiger/

[4] A. Biere and T. Jussila. *Hardware model checking competition at CAV'06*. http://fmv.jku.at/hwmcc/

[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] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction", *Proc. FMCAD '07*.

[8] R. Brayton, "The synergy between logic synthesis and equivalence checking", *Keynote at FMCAD'07*. http://www.cs.utexas.edu/users/hunt/FMCAD/2007/presentations/fmcad07_brayton.ppt

[9] M. L. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation", *Proc. IWLS '06*, pp. 172-179. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_inv.pdf

[10] M. L. Case, A. Mishchenko, and R. K. Brayton, "Automated extraction of inductive invariants to aid model checking", *Proc. FMCAD '07*, pp. 165-172. http://www.eecs.berkeley.edu/~alanmi/publications/2007/fmcad07_ind.pdf

[11] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[12] N. Een and N. Sörensson, “Temporal induction by incremental SAT solving”, *Proc. BMC’03*, ENTCS, Vol. 89(4).
[13] N. Een and N. Sörensson, “An extensible SAT-solver”. *SAT ‘03*. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
[14] N. Een, “Cut sweeping”, *Cadence Technical Report 2007*. http://minisat.se/downloads/CutSweeping.ps.gz
[15] C. A. J. van Eijk. “Sequential equivalence checking based on structural similarities”, *IEEE TCAD*, Vol. 19(7), July 2000, pp. 814-819.
[16] F. Lu and K.-T. Cheng. “Sequential equivalence checking based on *k*-th invariants and circuit SAT solving”. *Proc. HLDVT’05*.
[17] F. Lu and T. Cheng. “IChecker: An efficient checker for inductive invariants”. *Proc. HLDVT ’06*, pp. 176-180.
[18] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification", *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.
[19] K. L. McMillan and N. Amla, “Automatic abstraction without counterexamples.” *Proc. TACAS ‘03*, LNCS, Vol. 2619, Springer, pp. 2-17.
[20] K. L. McMillan. “Interpolation and SAT-Based model checking”. *Proc. CAV’03*, pp. 1-13.
[21] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *Proc. DATE '05*, pp. 418-423.
[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. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf
[23] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843 http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cec.pdf
[24] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Combinational and sequential mapping with priority cuts”, *Proc. ICCAD ’07*, pp. 354-361. http://www.eecs.berkeley.edu/~alanmi/publications/ 2007/iccad07_map.pdf
[25] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Submitted DAC'08*. http://www.eecs.berkeley.edu/~alanmi/publications/2008/dac08_vss.pdf
[26] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. “Exploiting suspected redundancy without proving it”. *Proc. DAC’05*, pp. 463-466.
[27] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA ’98*, pp. 35-42.
[28] M. Prasad, A. Biere, and A. Gupta. ”A survey of recent advances in SAT-based formal verification”, *Intl. Journal on Software Tools for Technology Transfer (STTT)*, Springer 2005, Vol. 7 (2), pp. 156-173. http://fmv.jku.at/papers/PrasadBiereGupta-STTT-7-2-2005.pdf
[29] E. Sentovich et al. “SIS: A system for sequential circuit synthesis”. *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, UC Berkeley, 1992.

Table 5.2. Experimental results for PicoJava benchmarks [19].

Example	PI	PO	Reg	AIG	Cut	Clauses		Iter	Runtime, sec		
						Cand	Proved		Prepro	Induct	Total
pj005	439	1	342	7486	29855	4706	2686	4	3.00	2.36	5.36
pj006	1277	1	703	17542	28881	5000	1762	4	3.21	4.70	7.91
pj007	396	1	314	7224	28865	5000	2522	4	2.90	1.75	4.65
pj008	446	1	338	7555	29722	4682	2712	4	2.99	1.71	4.70
pj009	336	1	269	6844	27092	5000	1955	6	2.69	1.69	4.38
pj010	366	1	295	7493	27128	5000	2171	5	2.75	1.68	4.43
pj015	1322	1	775	18964	33495	5000	1575	5	3.78	6.62	10.40
pj016	1190	1	671	17000	28214	5000	2535	4	3.07	5.41	8.48
pj017	626	1	440	12345	11290	5000	1613	4	1.35	2.62	3.97
pj018	514	1	386	9461	9619	5000	2248	5	1.09	2.03	3.12
pj019	476	1	383	10467	40885	4322	2415	5	4.06	2.43	6.49
Ratio						1.00	0.45		0.50	0.50	1.00

Table 5.1. Experimental results for Intel benchmarks [4].

Example	PI	PO	Reg	AIG	Cut	Clauses		Iter	Runtime, sec		
						Cand	Proved		Prepro	Induct	Total
intel_001	31	1	23	162	631	284	273	1	0.06	0.00	0.06
intel_002	72	1	44	632	2427	397	387	1	0.20	0.01	0.21
intel_003	82	1	47	739	2953	474	450	2	0.25	0.00	0.25
intel_004	82	1	41	479	1628	438	431	1	0.15	0.00	0.15
intel_005	165	1	69	1290	4333	343	315	1	0.40	0.00	0.40
intel_006	345	1	182	2394	6023	913	873	2	0.56	0.03	0.59
intel_007	1302	1	608	10192	39845	2655	2588	2	3.54	0.32	3.86
intel_009	5400	1	2924	64112	144072	3418	1710	6	45.79	314.31	360.10
intel_010	534	1	367	6096	19820	1820	1686	2	1.85	0.56	2.41
intel_011	528	1	361	6015	19420	1901	1851	2	1.82	0.49	2.31
intel_012	5874	1	3153	61530	136138	3863	1435	5	14.61	36.10	50.71
intel_013	13284	1	7134	138310	306933	5000	1298	8	44.80	227.61	272.41
intel_014	4293	1	2376	43362	100189	3841	1331	8	10.80	34.20	45.00
intel_015	548	1	381	5621	18019	1993	1912	2	1.68	0.37	2.05
intel_016	2232	1	1353	19701	60045	5000	1620	15	6.06	43.68	49.74
intel_017	613	1	401	4192	12285	2196	1780	2	1.15	0.43	1.58
intel_018	486	1	321	4537	13722	1578	1518	2	1.26	0.27	1.53
intel_019	505	1	338	4669	14437	1617	1554	2	1.33	0.29	1.62
intel_020	349	1	233	3837	11864	1030	1006	2	1.10	0.13	1.23
intel_021	360	1	244	3949	12619	1099	1044	2	1.16	0.19	1.35
intel_022	525	1	358	5865	19424	1762	1684	2	1.84	0.48	2.32
intel_023	353	1	240	3872	12763	1266	1218	2	1.18	0.17	1.35
intel_024	352	1	239	3887	12720	1181	1129	2	1.18	0.14	1.32
intel_025	1112	1	654	9343	29196	3011	2921	2	2.75	0.75	3.50
intel_026	486	1	349	4118	11337	1787	1722	2	1.04	0.17	1.21
intel_027	5127	1	2783	56164	121856	4030	1477	8	13.25	39.93	53.18
intel_028	7426	1	3951	77104	168010	3588	1480	6	18.57	55.44	74.01
intel_029	559	1	389	5804	18626	2024	1938	2	1.73	0.45	2.18
intel_030	5400	1	2922	64053	145282	3021	2701	2	43.55	75.02	118.57
intel_031	523	1	359	5960	19031	1899	1724	3	1.78	0.79	2.57
intel_032	890	1	636	10532	34871	5000	1002	27	3.48	33.39	36.87
intel_033	4419	1	2414	53334	119353	3368	2873	2	32.56	53.63	86.19
intel_034	3292	1	2413	20944	52652	5000	2557	2	5.53	12.46	17.99
intel_035	4407	1	2414	52786	121561	2970	2671	2	33.00	48.50	81.50
intel_036	5807	1	3179	68984	157770	3527	2163	3	51.18	167.81	218.99
intel_037	5911	1	3196	62306	138475	4120	1380	7	14.95	54.87	69.82
intel_038	8992	1	4888	86703	198254	5000	954	5	21.04	94.87	115.91
intel_039	9493	1	5170	89510	209100	5000	1690	5	27.18	110.21	137.39
intel_040	9499	1	5179	88096	203717	5000	1497	4	20.90	70.60	91.50
intel_041	9261	1	5022	88455	204509	5000	1347	5	24.03	99.74	123.77
intel_042	8994	1	4884	86659	198813	5000	1382	4	22.72	80.21	102.93
intel_043	7213	1	3830	74804	162046	3513	1561	5	18.15	42.28	60.43
Ratio						1.00	0.71		0.57	0.43	1.00