# Invariant-Strengthened Elimination of Dependent State Elements

Michael L. Case[1,2] Alan Mishchenko[1] Robert K. Brayton[1] Jason Baumgartner[2] Hari Mony[2]
[1] Department of EECS, University of California at Berkeley, CA
[2] IBM Systems and Technology Group, Austin, TX

***Abstract*— This work presents a technology-independent synthesis optimization that is effective in reducing the total number of state elements of a design. It works by identifying and eliminating *dependent state elements* which may be expressed as functions of other registers. For scalability, we rely exclusively on SAT-based analysis in this process. To enable optimal identification of all dependent state elements, we integrate an inductive invariant generation framework. We introduce numerous techniques to heuristically enhance the reduction potential of our method, and experiments confirm that our approach is scalable and is able to reduce state element count by 12% on average in large industrial designs, even after other aggressive optimizations such as min-register retiming have been applied. The method is effective in simplifying later verification efforts.**

## I. Introduction

Logic synthesis and formal verification are closely related fields. Verification tools often rely on technology-independent synthesis optimizations to reduce the size of the design being verified, thereby enhancing the scalability of the verification process. In this work we present a technology-independent synthesis optimization that can reduce the number of state elements in a design, thereby enhancing the scalability of verification tools. The focus of this paper is on synthesis for verification.

A *dependent state element* is one which may be expressed as a function over other state elements in the design. Once identified, the overall state element count of the design may be reduced by replacing the dependent elements by the corresponding functions over the remaining elements. Many verification algorithms are highly sensitive to the number of state elements present in the design. For example, BDD-based reachability analysis generally requires exponential resources with respect to state element count, hence it may dramatically benefit from the elimination of dependent state elements [1]. The effectiveness of induction is also generally sensitive to the implementation of the logic, particularly in the presence of unreachable states [2]. Dependent state element elimination inherently reduces the fraction of unreachable states of a design, thereby enhancing inductiveness.

Dependency is traditionally identified using BDD-based algorithms (e.g., [1]), which practically limits its application to smaller designs or requires approximate compositional analysis, resulting in suboptimalities. Recently, it has been demonstrated that *combinational dependency* of next-state functions may be identified using purely SAT-based analysis [3], enabling the analysis to scale to much larger designs. However, the previous research lacks several elements that make this method effective in practice. In this paper, we address the topic of sequential dependent state variable elimination using SAT-based techniques. Our contributions are as follows.

- To enhance state element reduction capability, we formulate the dependency check in a way that allows our formulation to directly reduce the total number of state elements, discussed in Section IV-A.
- Because the dependent state element elimination process provides network simplifications that may not be *compatible*, we introduce a technique to heuristically arrive at a legal netlist of minimal size by finding a high-quality compatible subset of simplifications in Section IV-C.
- SAT-based dependent state element elimination can introduce logic bloat in the design, and to mitigate this we introduce a way to leverage flexibilities in the dependency check to enable heuristically greater combinational logic reductions (Section IV-D).
- To enable the identification of state elements which are dependent in the reachable states though not necessarily in arbitrary states, we integrate an unreachability invariant framework to approximate unreachable state *don't cares*. To enable a purely SAT-based method, our invariant generation framework uses random simulation and $k$-step induction to derive *inductive invariants* in a format suitable for efficient SAT-based dependency computation. This will be discussed in Sections IV-B and V. While the invariants here are derived to benefit dependent state state element elimination, they have application in many other verification contexts as well.

The algorithms described in this work can be partitioned into two main components: 1) dependent state element identification and 2) invariant generation. A synthesis loop alternates between these two components along with combinational synthesis until no further design reductions are possible. The technique has been shown to provide significant reductions even after a design has first been heavily synthesized. The number of registers in a design can be reduced by 12% on average, even after optimizations such as min-register retiming have been applied. The dependent state elements in our experience are re-expressed as complex functions over the remaining elements, indicating that these dependencies cannot be found by simpler techniques such as register correspondence.

## II. Background and Related Work

In this section we review fundamental synthesis concepts and terminology used throughout this paper. We also provide

a review of prior SAT-based resubstitution research.

### A. Boolean Networks and And-Inverter Graphs

A Boolean Network (BN) is a directed acyclic graph (DAG) composed of a set of input signals and Boolean functions [4]. Each Boolean function in the BN has an *on-set* (*off-set*) which is the set of valuations of the functions inputs that cause the function to evaluate to 1 (0). By the definition of a function, the on and off-sets must be disjoint, but there may be valuations of the inputs for which the value of the function is undefined. These input valuations are not contained in either the on- or off-sets and represent *don't care* conditions against which an implementation of the function may be optimized. In synthesis, such don't cares arise for several reasons, and two such reasons that are important in this work are: 1) There may be *sequential don't cares* in the form of *unreachable states* against which combinational logic functions may be minimized, and 2) There may be *satisfiability don't cares* where the sub-BN's that drive the function inputs may be incapable of generating certain input patterns.

In this work we operate on a particular type of BN referred to as an *And/Inverter Graph (AIG)*. An AIG is a DAG where each node is either a 2-input AND gate or an input to the AIG. Inverters may be present and are indicated with a complementation attribute on the edges in the DAG. A *sequential AIG* is an AIG that also contains state elements, hereafter referred to as *registers*. The register $r$ is the only state-holding AIG primitive, which has an associated *initial state* $init(r)$, defining its time-0 behavior, as well as a *next-state function* $next(r)$ which defines the value of $r$ at the following time-frame.

### B. SAT-Based Resubstitution

In logic synthesis, *resubstitution* refers to a process that recasts a Boolean expression as a function over other pre-existing Boolean expressions [4]. For example, suppose there are Boolean signals $X$, $g_1$, $g_2$, ..., $g_n$. Resubstitution may be used to build a function $F(\cdot)$ such that $X = F(g_1, g_2, \ldots, g_n)$, or to prove that no such function exists. The functions $g_1$, $g_2$, ..., $g_n$ are referred to as the *basis* and $F(\cdot)$ as the *dependency function*. Upon finding $F$, the old implementation of $X$ can be removed and replaced with with the new implementation of $F$. Often resubstitution yields a reduction in the size of the AIG, and for this reason has been the focus of much synthesis research.

Traditionally, resubstitution is performed using Binary Decision Diagrams (BDDs) [1]. While efficient for small functions, the scalability of BDD-based techniques is often limited to modestly sized functions and bases, preventing optimal dependency identification in larger netlists. Recently it has been demonstrated that resubstitution may be cast as a Boolean Satisfiability (SAT) problem [3]. The formulation builds a combinational test circuit and uses SAT to determine if the circuit's single output is satisfiable. If the answer is `unsat` then the dependency function $F(\cdot)$ exists and can be extracted from the proof of unsatisfiability through interpolation. This

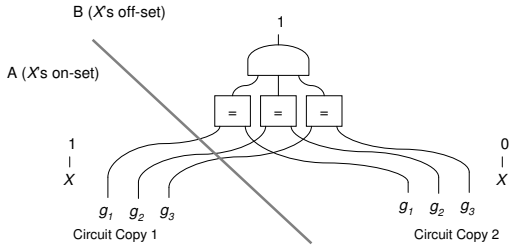

Fig. 1. SAT-based dependency formulation [3]

method offers substantially greater scalability than possible with BDD-based analysis.

Suppose that we wish to express $X$ as a function over signals $g_1, \ldots, g_3$. This is possible if and only if for each valuation to signals $g_1, \ldots, g_3$ there is a single possible $X$ valuation. We can test if such a resubstitution exists using the circuit shown in Figure 1. Two separate copies of $X$, $g_1$, $g_2$, and $g_3$ are instantiated. Each pair of $g$'s is constrained to have the same value, and the pair of $X$'s is constrained to have differing values. A resubstitution exists if and only if this test circuit is unsatisfiable. Many SAT solvers can be configured to record a proof of unsatisfiability [5], and interpolation on this proof provides the dependency function.

Given Boolean formulas $A(x, y)$ and $B(y, z)$, if $A(x, y) \cdot B(y, z) = 0$ then there exists a *Craig Interpolant* [6] $I$ such that $I$ refers to only the common variables $y$ of $A$ and $B$, and $A \implies I \implies \overline{B}$. [7] provides an algorithm to extract the interpolant $I$ from the proof of unsatisfiability of $A \cdot B$. This technique was first introduced to the verification community in a SAT-based unbounded verification algorithm [8] where the interpolant represents an overapproximate image computation.

In the resubstitution context, interpolation will be used to derive a dependency function. We may partition the resubstitution test circuit in Figure 1 into two halves $A$ and $B$. $A$ represents the set of $g$'s where $X = 1$, the on-set of $X$. Similarly, $B$ represents the off-set of $X$. Because $A \implies I \implies \overline{B}$, $I$ is a function that lies in between the on and off-sets of $X$, and we can replace $X$ with $I$. Furthermore, $I$ only refers to the common variables between $A$ and $B$, namely the $g$'s, hence $I$ provides the dependency function.

While this SAT-based formulation of [3] enables substantially greater scalability than BDD-based techniques, this formulation is limited in four key ways:

1) The prior research can only re-express combinational logic and cannot directly eliminate registers. In this work we simplify verification problems, and reducing the number of registers is important to increasing the scalability of many verification algorithms. Section IV-A will discuss how we directly eliminate registers.
2) The prior research cannot identify dependencies which hold in all reachable states but not in arbitrary unreachable states. In our experience, the reduction potential of this combinational analysis is often a subset of that possible using min-register retiming with integrated resynthesis [9]. We use invariants to overcome this limitation, as discussed in Sections IV-B and V.
3) The prior research does not address incompatibilities

present in the set of found dependencies. Often, dependencies must be discarded to avoid creating combinational cycles in the logic. We discuss an efficient way to compute a compatible set of dependencies in Section IV-C.

4) The prior research does not address the logic bloat that may result from interpolation. In general, logic generated by interpolation is highly redundant. We discuss a method to mitigate this logic bloat in Section IV-D.

## III. Dependent Register Elimination Algorithm

Our overall dependent register elimination routine is illustrated in Algorithm 1. The method used to eliminate dependent registers will be detailed in Section IV, and the method used to compute invariants will be discussed in Section V. At the top level, these two methods are iterated along with combinational synthesis (e.g., [10]) until design size is no longer reduced.

**Algorithm 1** Dependent register elimination

```
 1: function sequentialResynthesize(design)
 2:   invariants := ∅
 3:   repeat
 4:     invariants += gatherInvariants(design, invariants)
 5:     design := eliminateRegisters(design, invariants)
 6:     design := combinationalSynthesis(design)
 7:   until (No change in design size)
 8:   return design
 9: end function
10:
11: function gatherInvariants(design, invariants)
12:   while (Config file calls for more invariants) do
13:     family, parameters := readConfigFile()
14:     candidates := getCandidates(family, parameters)
15:     candidates := reduceToBest(candidates, invariants)
16:     candidates := testBaseCase(candidates)
17:     repeat
18:       candidates := testIndStep(candidates, invariants)
19:     until (No change in candidate set)
20:     invariants += candidates
21:   end while
22:   return invariants
23: end function
24:
25: function eliminateRegisters(design, invariants)
26:   depends := ∅
27:   for all (registers R in design) do
28:     test := buildResubTest(next(R), other next-states)
29:     if (satSolve(test) == unsat) then
30:       proof := getResolutionProof()
31:       curr := getDependencyFunc(R, proof)
32:       notCurr := getDependencyFunc(¬R, proof)
33:       depends += pickBest(curr, notCurr)
34:     end if
35:   end for
36:   depends := makeCompatible(design, depends)
37:   return simplifyDesign(design, depends)
38: end function
```

## IV. Resubstituting for Optimal Logic Reduction

In this section, we discuss our enhanced resubstitution procedure (function `eliminateRegisters` in Algorithm 1). Our resubstitution setup is illustrated in Figure 2, which

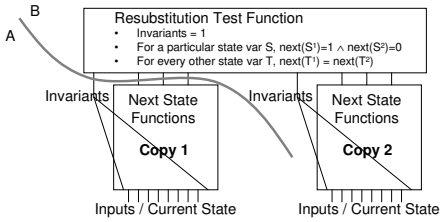


Fig. 2. Our enhanced resubstitution framework

is similar to Figure 1 but modified in several ways. This section will discuss how we target the formulation to find dependent registers as well as enhancements that make SAT-based resubstitution effective in practice. We iteratively call this procedure for every next-state function in the design in order to find the set of all dependent registers.

### A. Register Elimination

If the resubstitution formulation illustrated in Figure 2 is unsatisfiable, then a dependency function will be obtained that may be used as a replacement for $next(S)$. Trading the existing implementation of $next(S)$ with the dependency function may yield a savings in the number of ANDs in the AIG. This paper is targeted to aiding verification where the number of registers often is more important than the number of ANDs. Here we present a formulation by which a dependency function can be used to directly eliminate a register in the design.

Consider Figure 3a where the logic needed to implement $next(S)$ is highlighted. If a dependency function exists, it will express $next(S)$ as a function of the other two next-state signals $next(T_1)$ and $next(T_2)$. The implementation of $next(S)$ may be replaced with this dependency function, as illustrated in Figure 3b. We can further simplify the design by expressing this dependency function over the current states instead of the next-states, thereby eliminating register $S$.

*Definition 1:* Let an *orphan state* be any state $\sigma_1$ for which there does not exist a state $\sigma_2$ such that $\sigma_1$ is reachable from $\sigma_2$ in one transition.

Note that every reachable state, with the possible exception of the design's initial state(s), is not an orphan state.

*Theorem 1:* For registers $S$, $T_1$, ..., $T_n$, if there exists an $F(\cdot)$ such that $next(S) = F(next(T_1), \ldots, next(T_n))$ then for every state that is not an orphan state $S = F(T_1, \ldots, T_n)$.

*Proof:* Let $\sigma_1$ be a state of the design and a concrete valuation of the registers $S$, $T_1$, ..., $T_n$. If $\sigma_1$ is not an orphan state then there exists a state $\sigma_2$ such that $\sigma_1$ can be reached in one transition from $\sigma_2$. Let $X(\sigma_j)$ denote the

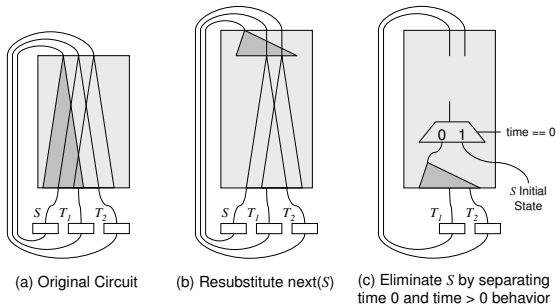


Fig. 3. Register elimination process

valuation of register $X$ in state $\sigma_j$ and note that there exists inputs such that $next(X(\sigma_2)) = X(\sigma_1)$. From the hypothesis we have $next(S(\sigma_2)) = F(next(T_1(\sigma_2)), \ldots, next(T_n(\sigma_2)))$. Rewriting we see that $S(\sigma_1) = F(T_1(\sigma_1), \ldots, T_n(\sigma_1))$. ■

Theorem 1 allows for the dependency function computed over next-state functions to be expressed over current states, provided the initial state(s) are accounted for. The result of this process is illustrated in Figure 3c. The dependency function between Figure 3b and 3c is identical; only the logic driving its inputs has changed. The register $S$ has been completely eliminated at the cost of initial state correction logic. To correct the initial state, we introduce a multiplexor which at time 0 will drive the initial value of the register being eliminated, and thereafter will drive the dependency function for the next-state function of the eliminated register. To enable selection of these two values, a register may need to be introduced to the design which initializes to 1 and thereafter drives 0. This register is reused across all resubstitutions.

While Theorem 1 is not guaranteed to hold in the initial state, in some cases the initial value may be preserved by that dependency function. That is, the value produced by the dependency function at time 0 may be identical to the initial value of the register being removed, and the initial state correction logic can be omitted. On a benchmark suite for which 3,390 resubstitutions were performed, the initial state had to be corrected 67% of the time. Our designs had complex initialization functions due to retiming [11] whose value the dependency function could replicate with relatively low probability. This illustrates the power of our technique enhance register reduction capability particularly in the presence of complex initial states.

### B. Optimal Dependency Identification via Invariants

A key strength of our formulation is its use of unreachability invariants, as illustrated in Figure 2. An unreachability invariant may be synthesized into a gate $I$ over registers in the design such that $I = 0$ only on unreachable states. Through the use of an adequately strong set of unreachability invariants, our formulation may optimally identify all dependent registers. However, due to resource limitations, an incomplete set of unreachability invariants will often be used to identify many but not necessarily all dependent registers. We discuss the invariant generation scheme which we have found effective in this application in Section V.

### C. Compatible Dependencies

The `eliminateRegisters` function from Algorithm 1 will attempt to resubstitute each next-state function present in the design. Through this process, a large number of dependency functions may be identified that can replace existing registers as depicted in Figure 3. Unfortunately, the set of dependencies found in this manner are generally not *compatible*, and if multiple dependency simplifications are performed simultaneously then often a combinational cycle will be created in the AIG resulting in an illegal design. A compatible set of dependencies is one in which all dependencies can be applied simultaneously with no resultant combinational cycles. Therefore, once the dependencies have been identified, one must identify a subset of compatible dependencies contained therein, and this chosen subset may impact the size of the resulting design.

**Algorithm 2** Selecting a set of compatible dependencies

```
1: function makeCompatible(design, dependencies)
2:    scored := ∅, compatible := ∅
3:    for all (Dep. D in dependencies) do
4:       red = D.redundant_AIG_node
5:       repl = D.replacement_AIG_node
6:       gain = aigSize(design) - aigSize(design - red + repl)
7:       scored[D] = scoreFunction(gain.regs, gain.ANDs)
8:    end for
9:    sortDescending(scores)
10:   for all (Dep. D in scored) do
11:      red = D.redundant_AIG_node
12:      repl = D.replacement_AIG_node
13:      if (!isCyclic(repl, red, compatible)) then
14:            compatible += D
15:      end if
16:   end for
17:   return compatible
18: end function
```

To illustrate the notion of incompatible resubstitution, consider a design with registers $R_1, \ldots R_n$ which have a one-hot encoding where in every reachable exactly one of these $n$ registers will evaluate to 1. Given adequate invariants to characterize this one-hot condition, the following dependencies may be identified:

$$R_1 = \overline{R_2} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots, \quad R_2 = \overline{R_1} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots$$

It is not possible to express $R_1$ as a function of $R_2$ and simultaneously express $R_2$ as a function of $R_1$ without creating a combinational cycle.

Finding a compatible subset of dependencies is a computationally difficult task. Finding an optimal subset would entail enumerating and testing every possible subset, and this is feasible for only very small sets of dependencies. In this work we utilize a heuristic to quickly find a near-optimal subset of compatible dependencies.

After the complete set of dependencies is found, we reduce this to a set of compatible dependencies as illustrated in Algorithm 2. Each found dependency consists of two signals: a *redundant* signal that will be eliminated and a *replacement* signal that will be introduced in its place. We first sort the dependencies in the order of their ability to simplify the circuit, computed as a function `scoreFunction`. (Experimentally, we have found that the function $20 \cdot$ *gain*.regs + *gain*.ANDs works well.) The list of sorted dependencies is then iterated over, and a subset of compatible dependencies is greedily found. For each dependency, we test if performing this optimization in the presence of the other *compatible* dependencies will introduce a combinational cycle using `isCyclic`. If so, the candidate merge is discarded. Otherwise the merge is added to the *compatible* set.

While this search is greedy, prioritizing the dependencies by score enables the algorithm to capture most of the optimization potential present in the original set of dependencies. This

TABLE I. Compatible dependencies on a set of IBM benchmarks

| | Total Deps. | | Compatible Deps. | |
|---|---|---|---|---|
| Design | Count | Sum Score | Count | Sum Score |
| IBM01 | 376 | -1714 | 325 | -91 |
| IBM02 | 194 | -659 | 154 | -725 |
| IBM03 | 428 | -18278 | 374 | -16950 |
| IBM04 | 678 | 643 | 579 | 909 |
| IBM05 | 35 | 312 | 22 | 258 |
| IBM06 | 102 | 573 | 58 | 308 |
| IBM07 | 142 | 139 | 102 | -97 |

is illustrated on a set of industrial designs in Table I. For each design, the found dependencies and compatible subset of these found dependencies are examined. Usually only a small percentage (24%) of the total dependencies must be discarded to form a compatible subset. If the total gain is summed over all possible dependencies, we see that the sum gain from the compatible dependencies is similar. This indicates that most of the AIG optimization potential present in the full set of dependencies was captured by the compatible subset.

### *D. Reducing Dependency Function Interpolants*

The dependency functions are obtained from the interpolant of a proof of unsatisfiability. Using the method given in [7], the resultant logic will have size that is linear in the size of the resolution proof. The proof of unsatisfiability for complex SAT problems may be large, and this may result in an interpolant and resulting dependency function that are very large. Here we explore several ways to control the size of the obtained dependency functions.

The most basic way to control the size of the dependency functions is with combinational synthesis. Logic that comes from interpolants is usually highly redundant and amenable to combinational synthesis techniques [12]. While combinational synthesis is effective in reducing the size of these interpolants, it is too slow to be used on every interpolant prior reducing to a compatible subset in Section IV-C. Here we focus on ways to more directly optimize our dependency functions before combinational synthesis is applied.

One simple way to control the size of the interpolants before synthesis is applied is to use incremental SAT. Our implementation attempts to resubstitute each next-state function in the sequential AIG, and through this process many similar SAT problems are encountered. Using one incremental solver instance to solve all of these problems is advantageous for two reasons:[1]

1) One incremental solver typically learns fewer clauses than many non-incremental solvers. The size of an interpolant is related to the number of learned clauses, and using incremental SAT will result in a reduction in the total size of all interpolants.
2) In incremental SAT the learned clauses from one problem are preserved and may contribute toward the search for a satisfiable solution to a future problem. If the same learned clause participates in two proofs of unsatisfiability then the two interpolants will share common logic. This also reduces the total cost of the logic needed to implement all interpolants.

[1]Incremental SAT is generally preferred, but such solvers can store a large number of learned clauses. If memory is a concern, the solver instance may need to be periodically refreshed.

TABLE II. Dependency function use on a set of IBM benchmarks

| | Depend. | Avg. Score by Repl. Type | |
|---|---|---|---|
| Design | Count | $S$ | $\overline{S}$ |
| IBM01 | 1128 | 0.70 | -0.87 |
| IBM02 | 582 | 0.78 | 0.04 |
| IBM03 | 1284 | -6.56 | -20.20 |
| IBM04 | 2034 | 1.71 | 2.11 |
| IBM05 | 104 | 3.10 | 6.06 |
| IBM06 | 306 | 2.42 | 4.44 |
| IBM07 | 426 | 1.44 | 2.21 |

In addition to using incremental SAT, we propose a more intelligent approach to mitigate logic bloat. Consider the resubstitution framework shown in Figure 2 that is able to eliminate a register $S$ by resubstituting $next(S)$. Copy 1 of the transition relation represents the on-set of $next(S)$, and Copy 2 represents the off-set. Using the partitioning that separates Copy 1 from the rest of the circuit as shown, we get an interpolant $I$ such that $next(S) \implies I \implies \overline{next(S)}$, and the resulting dependency function is able to replace $S$ using the concepts of Section IV-A. However, the problem is symmetric and we could alternatively partition to separate Copy 2 from the rest of the circuit. In this case, $\overline{next(S)} \implies I \implies next(S)$, and the dependency function is be able to replace $\overline{S}$.

Thus, we have the flexibility to compute the interpolant in two different ways to either replace $S$ or $\overline{S}$. These two replacements affect the size of the modified AIG in different ways, and to quantify this each possible replacement is scored in a manner identical to Section IV-C. By selecting the highest-scoring replacement, the dependency function can be used to its best advantage.

This is examined on a suite of industrial benchmarks in Table II. For each design, the number of found dependencies is given. Each dependency is scored as if it were used to replace one of the two signals: $S$ or $\overline{S}$, and the average score for each replacement type is given. A negative score indicates that the number of ANDs introduced to build the dependency function was greater than the cost of the logic being removed. Note that the signal we would prefer to replace is benchmark-dependent. In general, it is also dependency-specific, and our implementation individually scores each dependency in two ways in order to best utilize each dependency function.

## V. Invariant Generation

Unreachability invariants are essential to the reduction potential of SAT-based dependent register elimination. Given invariants which adequately characterize the unreachable states of a design, the formulation of Figure 2 is able to optimally identify all dependent registers in a design. However, in practice resource limitations will entail that the set of invariants may be a subset of those which truly hold of the design. The invariant generation approach which we have found useful for dependent register identification is detailed in this section.

The invariant generation algorithm is depicted in function `gatherInvariants` of Algorithm 1. The set of invariants is found over several iterations of a basic invariant discovery

algorithm. By finding invariants over several iterations, the overall framework is made considerably faster as the number of candidate invariants to be proved in each iteration decreases. Also, after each iteration completes a new set of invariants is found, and this provides a place that the computation can be safely terminated if computational resources are exceeded. After each iteration, a set of new invariants have been obtained that can then be added to the global collection of proved invariants.

When discovering new invariants, the first step is gathering the *candidate invariants*, properties that are suspected to hold but have not yet been proved. A set of properties from a particular property family, described in more detail in Section V-A, are first validated against a small set of simulation vectors. Each vector is derived from a random walk on the reachable state space and is therefore guaranteed to visit only reachable states. If any candidate invariants are found invalid by simulation, they are immediately discarded.

Next, the remaining candidates are filtered, as described in Section V-B. The goal is to remove candidates that are easily falsifiable or that are not capable of refining the current reachable state set over-approximation, given by the conjunction of all the already proved invariants. By filtering the candidates in this way, the proof of these candidates is made more scalable while the candidate set's ability to strengthen the current set of proved invariants changes only very little.

The final step involves proving that the candidate invariants hold in all reachable states. There are a variety of unbounded verification algorithms that could be used, but in general induction is the most scalable for large industrial designs. In this work we use $k$-induction [13] which involves proving the following:

**Base Case** The candidate invariants hold for all states reachable in $k$ or less transitions from the initial state(s).

**Inductive Step** For all paths[2] of length $k$ on which the candidate invariants hold, the invariants also hold in all states reachable in the next time step.

### A. Property Families

The candidate invariants are local properties over the nodes in the design's AIG. In our implementation there are five property domains we consider: constants, equivalences, $k$-cuts, implications, and random clauses. Each family exhibits a different proof complexity and ability to refine the set of proved invariants.

**Constants** are nodes that appear to take the same value in all reachable states.

**Equivalences** are pairs of nodes that appear equivalent in every reachable state [14].

**$k$-cuts** are candidate invariants that are derived from $k$-feasible cuts of nodes in the network [15]. The set of $k$-feasible cuts for all nodes in a user-defined part of the AIG are enumerated. For each cut, candidate invariants are derived by forming a clause from the OR of the cut nodes. $2^k$ such candidates are derived, one for each polarity assignment to the cut nodes. In practice, for small $k$ the number of cuts is approximately linear in the size of the AIG, and therefore the number of candidate invariants is approximately linear as well.

**Implications** are 2-literal clauses over pairs of gates in a design [16], [17]. Because implications are found exhaustively, there may be a quadratic number of candidate implication invariants.

**Random clauses** are clauses formed from random sets of nodes. The size of such a set and the circuit location from which the nodes come is parametrized. These invariants are effective in refining the current reachability approximation in ways that the other property families are not capable of.

[2]This can be strengthened to *unique state induction* by only considering simple paths [13]. Here we omit that constraint for computational reasons.

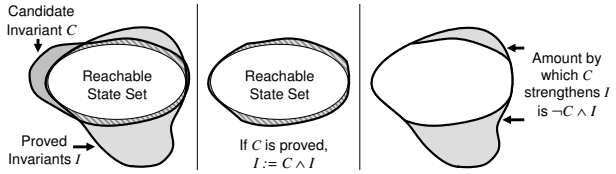


Fig. 4. Filtering candidate invariants

### B. Candidate Filtering

The number of candidate invariants generated may be very large, and it may be computationally infeasible to prove that each of these candidates are true in every reachable state. Additionally, not every candidate invariant is effective in characterizing unreachable states which are not already characterized by other already proved invariants. Therefore it is practically necessary to filter the candidates before they are proved.

Let $I$ be the conjunction of all previously proved invariants, as illustrated in Figure 4. The on-set of $I$ contains the set of reachable states, and we would like to find new invariants which are able to reduce this on-set to better approximate the reachable states.

The amount by which a new candidate invariant $C$ may strengthen the current reachability approximation $I$ is equal to the size of the on-set of $\neg C \wedge I$. The size of the on-set is difficult to precisely compute, though it can be estimated with random simulation. For each candidate invariant $I$, the number of times random simulation asserts $\neg C \wedge I$ is recorded as the candidate's "score. " The top-scoring candidate invariants are selected for the inductive proof attempt. These are the candidates that, if proved, can best refine the current reachability approximation.

### C. Invariant Generation Process

In this section we detail the algorithmic parameters which we have found useful for invariant generation. The basic invariant discovery loop (`gatherInvariants` of Algorithm 1) is iterated several times in order to quickly form a high-quality reachability approximation. In our experiments the following cycle was repeated until until a user-specified time limit was reached:

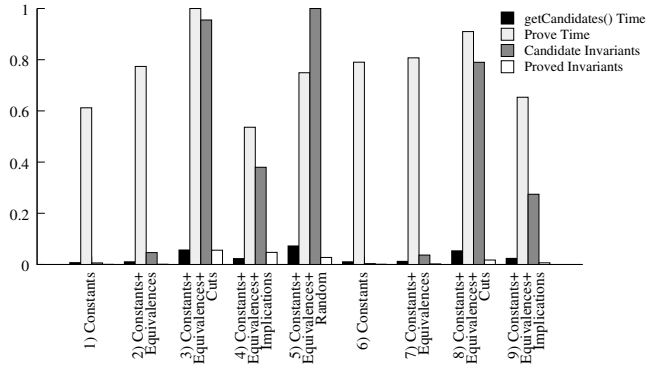


Fig. 5. Invariant generation process

1) Look for constants. Keep the 5000 "best" candidates (Section V-B), and prove them using 1-step induction.
2) Repeat step 1, and enable equivalences.
3) Repeat step 2, and enable 4-cuts. Restrict the search for cuts to the nodes near the registers (lower 8 AIG levels).
4) Repeat step 2, and additionally look for Boolean implications between registers.
5) Repeat step 2, and additionally look for random 3-literal clauses. Restrict the nodes that can participate in these clauses to be near the registers (lower 8 AIG levels).
6) Increment the $k$ in $k$-step induction, and go to step 1.

These invariant generation iterations are illustrated for one IBM benchmark in Figure 5. Four statistics are shown: 1) `getCandidates()` Time: time needed to derive candidate invariants, 2) Prove Time: time needed to prove those candidates, 3) Candidate Invariants: total number of candidate invariants after filtering, and 4) Proved Invariants: total number of proved invariants. These four statistics were collected over 9 iterations of the basic invariant discovery loop. Iterations 1 - 5 use $k = 1$ induction while 6 - 9 use $k = 2$ induction.

Note that iterations 3 and 8 compute the invariants over the same families, but the number of candidates in iteration 8 is significantly less than in iteration 3. This is because iterations 1-7 have derived a tighter reachability approximation than iterations 1-2 alone, and so in iteration 8 there are fewer candidates that are able to refine this reachability approximation and the filtering described in Section V-B is more effective.

On the industrial design examined in Figure 5, invariants successfully proved in the following three families: $k$-cuts, random clauses, and implications. In our experience all 5 families provide useful invariants in general, and cycling through different property families allows them to complement each other. The resultant reachability approximation is more effective in strengthening dependent state element elimination than invariants derived from any single family alone.

### D. Extracting Synthesis Properties

The constants and equivalences property families give invariants that can be directly used to simplify the circuit. Specifically, each invariant of this type implies that a node in the AIG can be removed and replaced with either another node or a constant.

Our implementation will detect such simple conditions and optimize the AIG. This is not as powerful as sequential SAT sweeping [14] because some useful candidate invariants may be removed by the filtering of Section V-B, but the synthesis optimizations require little overhead and occasionally help to reduce the design size. Additionally, previously proved invariants strengthen our inductive formulation and so occasionally we find more equivalences than sequential SAT sweeping.

## VI. Experimental Results

The algorithms discussed in this paper were implemented in the IBM internal verification tool *SixthSense* [11]. The methods are intended to simplify a design in order to reduce the computational expense of subsequent verification algorithms. To demonstrate this ability, two sets of experiments are presented. The first examines the power of our techniques to reduce the size of a design, the second examines the effect of these simplifications on verification algorithms. We additionally provide analysis of the qualitative nature of the reductions achieved in these experiments.

### A. Reduction Potential

The proposed synthesis method is able to yield reductions beyond those possible with state-of-the-art synthesis methods. To demonstrate this, our benchmarks were aggressively preprocessed before our dependent register elimination algorithm was applied. The preprocessing steps included: combinational synthesis, min-register retiming, removal of sequentially equivalent gates, and input reparameterization [11], [18]. These steps are able to dramatically reduce the size of the design, and after this preprocessing these existing synthesis methods cannot optimize these designs further.

Our proposed synthesis algorithm, illustrated in Algorithm 1, is evaluated on the preprocessed design in Table III. First we run Algorithm 1 with invariants disabled in order to explore the benefits of dependent state element elimination in the absence of invariants. We limit ourselves to two calls to function `eliminateRegisters` in Algorithm 1. Next, we revert to the preprocessed design then rerun with invariants enabled. The invariant generation algorithm was given 60 seconds to find invariants using the scheme discussed in Section V-C, and the number of proved invariants is given in the *Invars* column.

The effectiveness of our technique is highly dependent on the benchmark set. In *Set1*, the designs have many functionally dependent state elements, even after powerful sequential synthesis techniques were applied. On this benchmark set, dependent register elimination was very effective, even without invariants. However, the invariants did help mitigate the increase in AND gates which may occur through our dependent state element elimination.

In the benchmarks *Set2* and *Set3*, there exist very few registers that can be identified as dependent without the use of invariants. Enabling invariant generation more than doubles the number of dependent state elements that can be eliminated, on average[3]. This indicates that while dependent

[3]Occasionally, enabling invariant generation degrades our results. This is due to the heuristic and greedy nature of the algorithms described in Sections IV-C and IV-D.

TABLE III. Performance on three sets of IBM benchmarks[1]

| | Preprocessed Design | | | | Algorithm 1 Without Invariants | | | Algorithm 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Design | Inputs | ANDs | Regs | Time | ANDs | Regs | Time | Invars. | ANDs | Regs | Time |
| Set1 / IBM01 | 36 | 2083 | 393 | 158.85 | 1997 | 223 | 15.33 | 190 | 1995 | 224 | 91.53 |
| Set1 / IBM02 | 26 | 906 | 223 | 107.77 | 931 | 144 | 9.63 | 389 | 911 | 146 | 82.05 |
| Set1 / IBM03 | 44 | 5977 | 625 | 106.95 | 9602 | 429 | 69.90 | 166 | 9520 | 439 | 197.52 |
| Set1 / IBM04 | 34 | 3536 | 704 | 16.83 | 3600 | 415 | 52.87 | 58 | 3588 | 414 | 128.94 |
| Set1 / IBM05 | 189 | 19989 | 743 | 125.14 | 13828 | 733 | 77.24 | 514 | 14611 | 728 | 370.94 |
| Set1 / IBM06 | 40 | 4373 | 698 | 110.98 | 4330 | 669 | 32.44 | 17 | 4321 | 669 | 104.35 |
| Set1 / IBM07 | 44 | 1124 | 241 | 27.27 | 1082 | 189 | 10.79 | 288 | 1030 | 190 | 92.89 |
| **Set1 Summary[2]** | | | | | 1.04 | 0.75 | | | 1.03 | 0.75 | |
| Set2 / IBM08 | 8 | 502 | 101 | 101.22 | 499 | 101 | 6.22 | 12 | 499 | 101 | 73.17 |
| Set2 / IBM09 | 16 | 3258 | 683 | 16.01 | 3238 | 678 | 48.81 | 10 | 3232 | 677 | 175.71 |
| Set2 / IBM10 | 89 | 4463 | 823 | 104.53 | 4471 | 814 | 55.29 | 29 | 3164 | 690 | 141.08 |
| Set2 / IBM11 | 26 | 2402 | 530 | 102.77 | 2433 | 523 | 37.10 | 522 | 2386 | 521 | 214.00 |
| Set2 / IBM12 | 114 | 770 | 199 | 104.69 | 768 | 197 | 7.69 | 431 | 754 | 195 | 88.15 |
| Set2 / IBM13 | 189 | 5111 | 193 | 18.17 | 4878 | 167 | 11.84 | 160 | 4882 | 166 | 100.60 |
| **Set2 Summary[2]** | | | | | 0.99 | 0.97 | | | 0.94 | 0.94 | |
| Set3 / IBM14 | 38 | 2773 | 470 | 103.15 | 2773 | 470 | 10.76 | 123 | 2735 | 461 | 100.83 |
| Set3 / IBM15 | 125 | 15796 | 668 | 120.69 | 11972 | 655 | 113.74 | 399 | 13113 | 655 | 256.80 |
| Set3 / IBM16 | 68 | 2757 | 680 | 21.55 | 2743 | 675 | 36.55 | 16 | 2740 | 675 | 98.53 |
| Set3 / IBM17 | 127 | 15867 | 654 | 125.49 | 11882 | 640 | 99.87 | 464 | 13017 | 640 | 521.74 |
| Set3 / IBM18 | 125 | 15675 | 668 | 128.19 | 11977 | 658 | 127.87 | 306 | 13165 | 658 | 336.07 |
| Set3 / IBM19 | 84 | 5958 | 776 | 111.55 | 6219 | 762 | 93.52 | 146 | 5560 | 689 | 154.88 |
| **Set3 Summary[2]** | | | | | 0.88 | 0.99 | | | 0.90 | 0.97 | |

[1] All experiments were run on a 1.83 GHz laptop running Linux 2.6. [2] Ratios are relative to the preprocessed AIG size.

state elements abound, their discovery requires the use of reachability invariants.

### B. Impact on Verification

Enabling reductions in the number of registers and possibly AND gates in a sequential AIG is advantageous in numerous application domains. Our goal in this paper is to leverage such reductions to simplify an overall verification task. In order to quantify the impact of our reduction on verification complexity, we ran 267 IBM designs, each with a collection of safety properties, through a rugged Transformation-Based Verification (TBV) [11] setup. This setup involves numerous sequential synthesis techniques as mentioned in Section VI-A (Algorithm 1 not included) along with SAT-based bounded model checking (BMC), induction, and interpolation. BMC was limited to 90 seconds, and $k$-induction was run for 300 seconds, both increasing bounds until the time limit was reached. Interpolation works on each property separately and was limited to 300 seconds per property. It is known that reductions in the number of registers can additionally aid BDD-based verification, but these designs were large and consistently beyond the capacity BDDs.

Of the 267 designs processed in this manner, only 141 had unsolved properties after the rugged TBV script. On these remaining 141 examples, we applied Algorithm 1 and repeated the same rugged TBV script. Algorithm 1 further simplifies the designs, and any properties that are solved in the second TBV pass are a direct result of these simplifications.

The second TBV pass solved properties in 10 of the remaining 141 designs, or 7.1%, as illustrated in Table IV. The cone-of-influence reduced design size after the initial TBV processing is illustrated first, followed by the size after the reduction of Algorithm 1. We next illustrate the extent to which our dependent register elimination may synergistically enhance the reduction potential of subsequent synthesis algorithms, and finally we indicate which verification algorithms were successful in solving properties in the second TBV pass. Note that Algorithm 1 is able to reduce the number of registers in these examples, and also enables further register reductions by more conventional sequential synthesis techniques[4].

In total, 11 properties are solved as a result of the simplifications Algorithm 1. Eight properties are solved with interpolation, and the time needed to solve these properties is well below the 300 second timeout. One property was solved with induction, again far below the 300 second timeout. There were two properties for which a counterexample was "hit." One hit was with BMC, indicating that BMC was able to process more time steps on the reduced design than it was on the original design, and a counterexample was present in these extra time steps. The other hit occurred as simulation run during redundancy removal analysis.

Overall, we have found this technique to be a useful component of our multi-algorithm verification system. Since it is nontrivially expensive in itself, and due to the potential of AND gate increase through interpolant synthesis, in practice we have found it useful to leverage *after* a preprocessing of quicker transformations and proof / falsification algorithms similarly to the algorithm flows used in these experiments. While the percentage of designs for which we were able to solve additional properties through our technique may seem limited (7.1%), we are encouraged by this result since these 141 designs comprise very difficult verification problems which otherwise are extremely difficult or impossible to solve.

### C. Characterizing the Simplifications

In order to more fully understand the type of reductions enabled by Algorithm 1, we studied several examples to attempt to characterize the nature of the reductions. Recall that

[4]The occasional bloat in AND count during this final synthesis phase is primarily due to retimed initial value computation and input reparameterization [11], [18].

TABLE IV. Effect on Verification[1]

| Specified Design | | After TBV | | | After Algorithm 1 | | After Resynthesis | | |
|---|---|---|---|---|---|---|---|---|---|
| Design | Properties | Properties | ANDs | Regs | ANDs | Regs | ANDs | Regs | After More TBV |
| IBM20 | 6 | 2 | 758 | 245 | 1100 | 241 | 697 | 229 | 1 proof w/ interpolation (36 sec) |
| IBM21 | 17 | 3 | 5754 | 457 | 5810 | 455 | 8173 | 417 | 2 proofs w/ interpolation (37 sec, 24 sec) |
| IBM22 | 387 | 5 | 13620 | 1147 | 13550 | 1137 | 24917 | 1126 | 1 hit w/ BMC (depth 44, 90 sec) |
| IBM23 | 6 | 3 | 1853 | 242 | 2915 | 223 | 1815 | 223 | 1 proof w/ interpolation (165 sec) |
| IBM24 | 17 | 3 | 6581 | 455 | 6626 | 449 | 8067 | 435 | 1 proof w/ interpolation (237 sec) |
| IBM25 | 1 | 1 | 1002 | 197 | 1223 | 195 | 1010 | 195 | 1 proof w/ interpolation (182 sec) |
| IBM26 | 6 | 2 | 758 | 245 | 1100 | 241 | 697 | 229 | 1 proof w/ interpolation (36 sec) |
| IBM27 | 2 | 2 | 4350 | 820 | 3073 | 687 | 3252 | 680 | 1 proof w/ induction ($k$=37, 203 sec) |
| IBM28 | 41 | 24 | 26932 | 3259 | 26953 | 3249 | 26872 | 3247 | 1 hit with random simulation |
| IBM29 | 1 | 1 | 1002 | 197 | 1223 | 195 | 1010 | 195 | 1 proof w/ interpolation (179 sec) |

[1] All experiments were run on a cluster of IBM workstations running 64-bit AIX 5.3 on POWER 5 CPUs ranging from 2.1 to 2.2 GHz.

our experiments included aggressive synthesis preprocessing, hence most of the simpler reduction potential that could be claimed as dependency was already eliminated.

The designs that we report in our experiments are effectively *testbenches* comprising a hardware logic component, a *driver* which constrains legal input stimulus, and a *checker* which encodes correctness properties. One source of dependencies that we discovered was inherent in this testbench setting: first, the logic used to specify the checker and driver is not highly hand-optimized, instead comprising logic such as sparse decodings of values driven or sampled from the design. Second, in cases dependencies arise between the specification logic and the design; e.g., perhaps the checker tracks a "design is busy" condition which happens to be equivalent to the OR of several state bits inside the design. Finally, in cases the input constraints used in a driver may create dependencies that otherwise would not exist in the design.

We also discovered similar dependencies within the design logic itself. For example, instruction-decode logic and one-hot state machines create sparse state vectors which can often be re-expressed with fewer registers. This redundancy is desirable in high-performance circuitry to minimize combinational delays but presents opportunity for reductions using Algorithm 1.

There is substantial motivation for the development of automated techniques to efficiently identify and eliminate those dependencies. While some of these reduction opportunities could be readily manually identifiable, some of the dependencies identified in our benchmarks were intricate relations over dozens of other registers, which would have been impractical to attempt to identify by hand.

## VII. Conclusion

This paper developed a method to eliminate functionally dependent state elements in a sequential design. The method extends prior work in dependency identification [1] with several enhancements:

- Dependent state elements can be identified and directly removed, reducing the number of registers in the design.
- A method to identify a compatible set of dependencies is discussed. This method is fast and effective in reducing the found dependencies to a compatible subset without sacrificing significant optimization potential.
- A method is developed that can effectively mitigate the logic bloat that comes from interpolation.
- The dependent state element elimination is strengthened with an invariant generation framework, enabling the detection of unreachable state invariants which extend this purely SAT-based optimization technique into a sequential synthesis transformation.

Experiments demonstrate that this technique can significantly reduce the number of registers in industrial designs even after powerful sequential synthesis methods such as min-register retiming and redundancy removal have been applied, an area where we have found the technique of [1] to be ineffective. This reduction is also shown to synergistically enable greater synthesis optimizations, and to enhance a variety of verification algorithms.

## Acknowledgements

The authors would like to thank Per Bjesse for all his help in preparing this paper for publication.

## References

[1] J. Jiang and R.K. Brayton, "Functional Dependency for Verification Reduction", at *CAV* 2004.
[2] M. Wedler, D. Stoffel and W. Kunz, "Exploiting state encoding for invariant generation in induction-based property checking," in *ASP-DAC*, 2004.
[3] C. Lee, J. Jiang, C. Huang and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *ICCAD* 2007.
[4] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," in *TCAD* 1987.
[5] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications," in *DATE* 2003.
[6] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," in *J. Symbolic Logic* 1957.
[7] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations", in *J. Symbolic Logic* 1997.
[8] K.L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV* 2003.
[9] J. Baumgartner and A. Kuehlmann, "Min-Area Retiming on Flexible Circuit Structures," in *ICCAD*, 2001.
[10] A. Mishchenko, S. Chatterjee and R.K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, 2006.
[11] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *FMCAD* 2004.
[12] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *ICCAD* 2006.
[13] N. Eén and N. Sörensson, "Temporal Induction by incremental SAT solving," in *Proc. Workshop on Bounded Model Checking* 2003.
[14] C.A.J. van Eijk, "Sequential equivalence checking based on structural similarities," in *TCAD* 2000.
[15] M.L. Case, A. Mishchenko and R.K. Brayton, "Cut-Based Inductive Invariant Computation," at *IWLS* 2008.
[16] M.L. Case, A. Mishchenko and R.K. Brayton, "Inductively Finding a Reachable State Space Over-Approximation," in *IWLS* 2006.
[17] M.L. Case and R.K. Brayton, "Maintaining A Minimum Equivalent Graph In The Presence of Graph Connectivity Changes," UC Berkeley Technical Report, 2007.
[18] J. Baumgartner and H. Mony, "Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies," in *CHARME* 2005.