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 25% on average in large industrial designs, even after other aggressive optimizations such as min-register retiming have been applied.**

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.
- Because the dependent state 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 dependent state elements 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 Section V. While the invariants here are derived to benefit dependent state state elimination, they have application to many 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 25% 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 method offers substantially greater scalability than possible with BDD-based analysis.

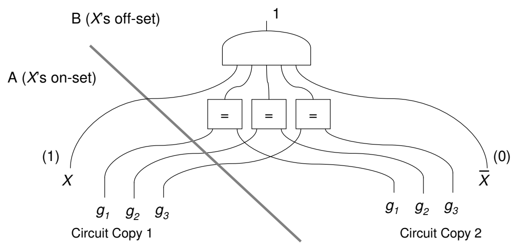

Fig. 1. SAT-based dependency formulation [3]

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 combina-

tional 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

```
function sequentialResynthesize(design)
  invariants := ∅
  repeat
    invariants += gatherInvariants(design, invariants)
    design := eliminateRegisters(design, invariants)
    design := combinationalSynthesis(design)
  until (No change in design size)
  return design
end function

function gatherInvariants(design, invariants)
  while (Config file calls for more invariants) do
    family, parameters := readConfigFile()
    candidates := getCandidates(family, parameters)
    candidates := reduceToBest(candidates, invariants)
    candidates := testBaseCase(candidates)
    repeat
      candidates := testIndStep(candidates, invariants)
    until (No change in candidate set)
    invariants += candidates
  end while
  return invariants
end function

function eliminateRegisters(design, invariants)
  depends := ∅
  for all (registers $R$ in design) do
    test := buildResubTest($next(R)$, other next-states)
    if (satSolve(test) == unsat) then
      proof := getResolutionProof()
      next := getDependencyFunc($next(R)$, proof)
      notNext := getDependencyFunc($\neg next(R)$, proof)
      curr := getDependencyFunc($R$, proof)
      notCurr := getDependencyFunc($\neg R$, proof)
      depends += pickBest(next, notNext, curr, notCurr)
    end if
  end for
  depends := makeCompatible(design, depends)
  return simplifyDesign(design, depends)
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 is similar to Figure 1 but modified in several ways. This

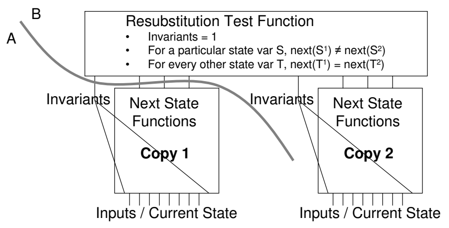

Fig. 2. Our enhanced resubstitution framework

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 σ_1 for which there does not exist a state σ_2 such that σ_1 is reachable from σ_2 in one transition.

Note that every no reachable state, with the possible exception of the design's initial state(s), is 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 σ_1 be a state of the design and a concrete valuation of the registers S, T_1, …, T_n. If σ_1 is not an orphan state then there exists a state σ_2 such that σ_1 can be reached in one transition from σ_2. Let $X(\sigma_n)$ denote the valuation of register X in state σ_n and note that there exists

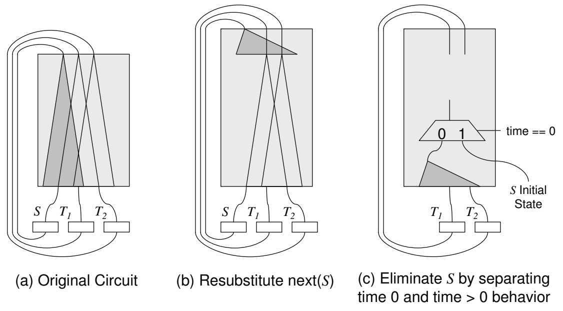

Fig. 3. Register elimination process

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 is illustrated on a set of industrial designs in Table I. For

TABLE I. Compatible dependencies on a set of IBM benchmarks

	Total Deps.		Compatible Deps.	
Design	Count	Sum Score	Count	Sum Score
Set1 / IBM01	348	-830	295	405
Set1 / IBM02	310	-17248	269	-17324
Set1 / IBM03	442	-17452	385	-17390
Set1 / IBM04	800	1058	571	985
Set1 / IBM05	34	303	22	219
Set1 / IBM06	184	1496	123	966
Set1 / IBM07	142	0	102	-242

each design, the found dependencies and compatible subset of these found dependencies are examined. Usually only a small percentage (14%) 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 large SAT problems may also be large, and this may make the interpolant and resulting dependency function much more complex than necessary. 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, in our experience, usually highly redundant and amenable to combinational synthesis techniques. 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.

In addition to using incremental SAT, we propose a more intelligent approach to mitigate logic bloat. For a particular

[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.	Average Score, By Replacement Type			
Design	Count	S	$\overline{S}$	$next(S)$	$\overline{next(S)}$
Set1 / IBM01	261	1.05	0.44	-2.57	-2.29
Set1 / IBM02	232	-8.64	-26.65	-12.18	-11.94
Set1 / IBM03	331	-6.03	-18.62	-9.46	-9.10
Set1 / IBM04	600	1.63	2.34	-1.82	-1.56
Set1 / IBM05	25	2.64	5.54	0.27	0.14
Set1 / IBM06	138	2.89	5.65	-0.32	-0.33
Set1 / IBM07	106	1.48	1.60	-1.95	-2.00

register Sis, resubstitution using a problem formulation similar to Figure 2 gives an alternate implementation of $next(S)$. Section IV-A tells how to use the dependency function to obtain an alternate implementation of S. Usually replacing S is advantageous because register S can be removed. However, if the initial state of S must be corrected then an additional register may be introduced to accomplish this, thereby cancelling the reduction in the number of registers in the design. For this reason it is advantageous to, on a dependency-by-dependency basis, choose to replace either $next(S)$ or S.

We gain additional freedom in our use of the dependency function by considering how the interpolant is extracted from Figure 2. Note that because $next(S)$ spans the two partitions of Figure 2, the interpolant will be a function of $next(S)$. More specifically, the interpolant will give us a dependency function F such that

$$next(S) \subseteq F(next(S), next(T_1), \ldots) \subseteq \overline{next(s)}$$

Assigning Boolean values to $next(S)$ gives two possible interpretations of the dependency function:

1) $next(S) = 1$ means that F can replace $next(S)$:
$\text{onset}(next(S)) \subseteq F(1, next(T_1), \ldots) \subseteq \text{offset}(\overline{next(S)})$
2) $next(S) = 0$ means that F can replace $\overline{next(S)}$:
$\text{offset}(next(S)) \subseteq F(0, next(T_1), \ldots) \subseteq \text{onset}(next(S))$

This gives the additional flexibility to use F to replace either $next(S)$ or $\overline{next(S)}$.

Combining the above concepts, a single dependency function F can be used to replace one of the 4 signals: S, $\overline{S}$, $next(S)$, or $\overline{next(S)}$. Each of these 4 replacements affects 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 found on a next-state function $next(S)$ is scored as if it were used to replace one of the four signals: S, $\overline{S}$, $next(S)$, or $\overline{next(S)}$, 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 4 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 [12] 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.

[2]This can be strengthened to *unique state induction* by only considering simple paths [12]. Here we omit that constraint for computational reasons.

Constants are nodes that appear to take the same value in all reachable states. Typically there are few sequentially constant nodes in a design, but the constants are fast to compute and prove.

Equivalences are pairs of nodes that appear equivalent in every reachable state [13]. The number of equivalences available depends on the design, but like constants these are usually fast to compute and are used for low-effort invariant generation.

k-cuts are candidate invariants that are derived from k-feasible cuts of nodes in the network [14]. 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. This makes the proof of such invariants manageable while providing a good reachability approximation.

Implications are 2-literal clauses over pairs of gates in a design [15], [16]. Because implications are found exhaustively, there may be a quadratic number of candidate implication invariants. This can make the proof very slow, but the candidate invariants found are usually able to significantly refine the current reachability approximation.

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. There are typically very many random clauses that appear to hold as candidate invariants, but they are effective in refining the proved invariants in ways that the other property families are not capable of.

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.

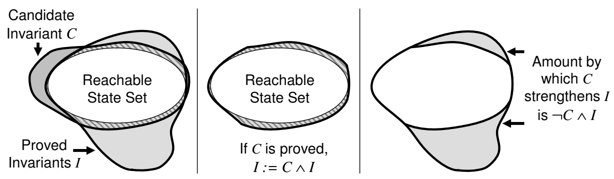
Candidate Invariant C
Reachable State Set
Proved Invariants I
Reachable State Set
If C is proved, $I := C \wedge I$
Amount by which C strengthens I is $\neg C \wedge I$

Fig. 4. Filtering candidate invariants

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:

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: the time needed to derive candidate invariants, 2) Prove Time: the time needed to prove those candidates, 3) Candidate Invariants: the total number of candidate invariants after filtering, and 4) Proved Invariants: the 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.

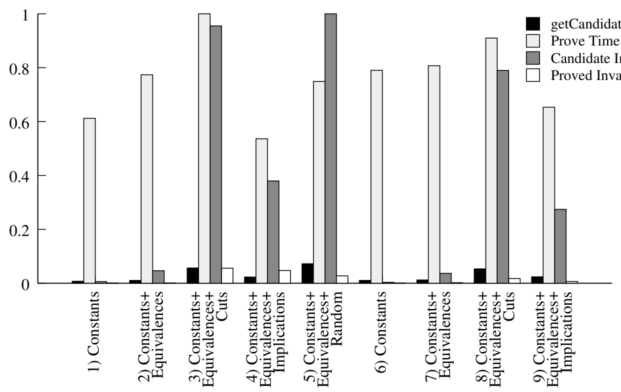

Fig. 5. Invariant generation process

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 accordingly. This is not as powerful as sequential SAT sweeping [13] because some useful candidate invariants may be removed by the filtering of Section V-B, but detection of the invariants with corresponding synthesis optimizations requires little overhead and occasionally helps to reduce the design size. Additionally, previously proved invariants strengthen our inductive formulation and so occasionally we find more equivalences than SAT sweeping.

VI. Experimental Results

The algorithms discussed in this paper were implemented in the IBM internal verification tool *SixthSense* [11]. All experiments were run on a 1.83 GHz laptop running Linux 2.6. Three sets of challenging IBM benchmarks were synthesized using our register resubstitution framework. The benchmarks were selected from a collection of designs with properties that are known to be difficult to prove or falsify.

The proposed synthesis method is able to improve upon the best-known synthesis methods to date, and to demonstrate this the benchmarks were aggressively preprocessed before our synthesis was applied. The preprocessing steps included: combinational synthesis, min-register retiming, combinational synthesis, removal of sequentially equivalent registers, and finally, one more round of combinational synthesis. These steps are able to dramatically reduce the size of the design, and after preprocessing the existing synthesis methods cannot optimize these designs further.

Our proposed synthesis algorithm, illustrated in Algorithm 1 was applied to the preprocessed designs twice. In the first pass, invariant generation was disabled in order to isolate the effects of dependent state element elimination. In the second pass, the design was reverted to the preprocessed snapshot and Algorithm 1 was rerun with invariant generation enabled. The invariant generation algorithm was given 180 seconds to find invariants using the scheme discussed in Section V-C, and the

TABLE III. Performance on three sets of IBM benchmarks

	Preprocessed Design				Algorithm 1 Without Invariants			Algorithm 1			
Design	Inputs	Regs	ANDS	Time	Regs	ANDS	Time	Invars.	Regs	ANDS	Time
Set1 / IBM01	59	386	1721	8.54	238	1582	12.98	806	238	1561	211.53
Set1 / IBM02	44	457	3292	4.73	294	6235	32.36	1102	295	6266	241.84
Set1 / IBM03	44	625	5982	4.61	429	9332	50.10	2031	430	9301	301.57
Set1 / IBM04	34	743	3693	4.70	457	4131	86.47	25	457	4073	225.30
Set1 / IBM05	104	744	7520	5.56	736	5885	176.34	161	731	6092	401.57
Set1 / IBM06	18	836	6312	9.62	775	6337	164.21	29	775	6309	374.02
Set1 / IBM07	44	241	1127	1.07	189	1463	17.20	872	190	1022	173.42
Set1 Summary[1]		1.00	1.00		0.75	1.23			0.75	1.17	
Set2 / IBM08	8	104	535	1.16	104	529	0.97	101	104	526	201.77
Set2 / IBM09	16	703	3401	4.72	696	3410	139.56	3110	Unreachable[2]		293.93
Set2 / IBM10	21	836	4351	5.12	828	4361	133.82	59	714	3098	300.83
Set2 / IBM11	19	540	2384	1.54	536	2399	46.04	670	534	3346	293.40
Set2 / IBM12	126	194	847	1.38	194	848	2.09	508	194	838	202.28
Set2 Summary[1]		1.00	1.00		0.99	1.00			0.96	1.02	
Set3 / IBM13	187	431	3836	4.00	431	3836	15.74	72	431	3827	248.58
Set3 / IBM14	38	474	2803	1.80	474	2803	7.23	299	455	2740	225.03
Set3 / IBM15	152	297	2162	2.50	263	2274	12.18	810	263	2184	217.10
Set3 / IBM16	63	674	6200	6.56	666	6105	81.01	90	665	6102	283.58
Set3 / IBM17	68	690	2849	4.74	688	2864	82.20	48	688	2851	307.74
Set3 / IBM18	65	778	6930	11.38	763	5365	128.61	534	685	4519	316.37
Set3 Summary[1]		1.00	1.00		0.98	0.97			0.95	0.94	

[1] Ratios are relative to the preprocessed AIG size.
[2] Synthesis proved all design properties to be unreachable. After COI reduction, the design has 0 registers and 0 ANDs.

number of proved invariants is given in the *Invars* column. The results of these experiments are given in Table III.

The effectiveness of our technique is highly dependent on the benchmark set. In *Set1*, all designs have many functionally dependent state elements, even after powerful sequential synthesis techniques such as min-register retiming were applied. On this benchmark set, dependent state element elimination was very effective, even without invariants. However, the invariants did help the state element elimination algorithm mitigate the increase in ANDs.

In the benchmarks *Set2* and *Set3*, there exist very few registers that can be identified as dependent state elements without the use of invariants. Enabling invariant generation triples the number of dependent state elements that can be identified, on average. This indicates that all of the "easy" state element elimination was actually performed by min-register retiming during preprocessing. While dependent state elements still exist after min-register retiming, their discovery requires the use of reachability invariants.

VII. Conclusion

This paper developed a method to eliminate functionally dependent state elements in a sequential design. The method is inspired by [1] but has several enhancements that make it effective in practice:

- Dependent state elements can be identified and directly removed, thereby reducing the total 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 show that while the effectiveness of this technique is highly benchmark dependent, it can reduce the number of registers in industrial designs by 25% even after powerful sequential synthesis methods such as min-register retiming have been applied, an area where we have found [1] to be ineffective.

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] N. Eén and N. Sörensson, "Temporal Induction by incremental SAT solving," in *Proc. Workshop on Bounded Model Checking* 2003.
[13] C.A.J. van Eijk, "Sequential equivalence checking based on structural similarities," in *TCAD* 2000.
[14] M.L. Case, A. Mishchenko and R.K. Brayton, "Cut-Based Inductive Invariant Computation," at *IWLS* 2008.
[15] M.L. Case, A. Mishchenko and R.K. Brayton, "Inductively Finding a Reachable State Space Over-Approximation," in *IWLS* 2006.
[16] M.L. Case and R.K. Brayton, "Maintaining A Minimum Equivalent Graph In The Presence of Graph Connectivity Changes," UC Berkeley Technical Report, 2007.