# Automated Extraction of Inductive Invariants to Aid Model Checking

Michael L. Case Alan Mishchenko Robert K. Brayton
Department of EECS, University of California, Berkeley
{casem, alanmi, brayton}@eecs.berkeley.edu

***Abstract*— Model checking can be aided by inductive invariants, small local properties that can be proved by simple induction. We present a way to automatically extract inductive invariants from a design and then prove them. The set of candidate invariants is broad, expensive to prove, and many invariants can be shown to not be helpful to model checking. In this work, we develop a new method for systematically exploring the space of candidate inductive invariants, which allows us to find and prove invariants that are few in number and immediately help the problem at hand. This method is applied to interpolation where invariants are used to refute an error trace and help discard spurious counterexamples.**

## I. Introduction

Formal model checking of safety properties in a sequential machine is often intractable. In practice, prior knowledge about the design and/or property being checked greatly reduces verification time. Often, this knowledge is all that makes formal verification practical.

Frequently this prior knowledge comes in the form of hints from the designer, but it is difficult to identify helpful hints, and there is a real danger of either expressing hints that do not help or that are not true. Extraneous hints may slow down the verification tool by consuming valuable resources; all hints must be proved to verify that the designer did not err, possibly increasing the complexity of the overall problem.

An ideal solution would be to automatically extract useful hints from the design. Such a method should focus only on those hints that immediately help the verification and whose proof is simple. This paper proposes a method to do exactly that.

We focus on simple design properties called *inductive invariants*. These are properties that hold in every reachable state and can be proved by *simple induction*, temporal induction using only a 1-step (current state, next state) model. Often a verification tool can be assisted by the knowledge that a small set of states in the design is unreachable. For example, if an abstraction reaches a bad state, we would like to know if any state on the error trace is unreachable. If there exists an inductive invariant that demonstrates this unreachability, then the counterexample is spurious, and there is no need to refine the model.

To harness this idea, we have built a tool that is able to show that a single, user-specified state is unreachable. It does this by finding and proving inductive invariants. If for some reason it is unable to complete the proof of the invariants, it will find and prove other, secondary invariants that enable the first proof to proceed. This method gives a hierarchy of proofs that when complete will yield a set of inductive invariants $\{P\}$ with the following properties:

- $\bigwedge_{p\in\{P\}} p$ implies that the user-specified state is unreachable.
- $\bigwedge_{p\in\{P\}} p$ can be proved with simple induction

This paper provides the theory behind this invariant generation method and explores how specific inductive invariants can help interpolation, a method for unbounded verification of safety properties.

## II. Related Work

An excellent background on formal verification can be found in [1], and [2] is a good overview of modern unbounded verification techniques. [3] describes the status of formal verification inside IBM. It describes the problem of extraneous hints and specifications in the following phrase: "person-months spent developing formal specs, merely to choke [the] FV tool." The above papers establish the basics and identify one of the key problems.

This paper will discuss a specific verification algorithm called interpolation. This method, originally proposed by McMillan in [4], is fast and compares favorably to other techniques [5], and thus it was chosen for study in this work.

This paper builds on our previous work [6] in which Boolean implications between design nodes are discovered through random simulation and proven with simple induction. The method was inspired by techniques proposed by van Eijk in [7]. Bjesse proposes a way to strengthen simple induction in [8]. We will strengthen simple induction in a different manner.

Finally, this work was motivated by Wedler [9] who discusses using implications between design nodes to form inductive invariants. He also attempts to strengthen simple induction in a manner similar to Bjesse. However, that work is limited in scope (1-hot machines), while our work is more general and provides a solid theoretical foundation.

## III. Applications of Inductive Invariants

Inductive invariants are useful in many applications. Here we examine two such applications: simple induction and interpolation. By exploring these applications, we can expose some of their weaknesses and motivate the need for inductive invariants.

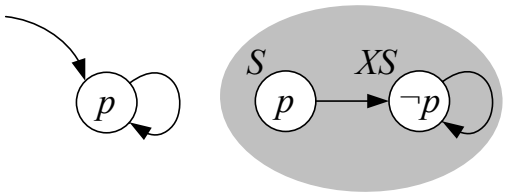


Fig. 1. The shaded states are unreachable but will make an inductive proof of $p$ impossible.

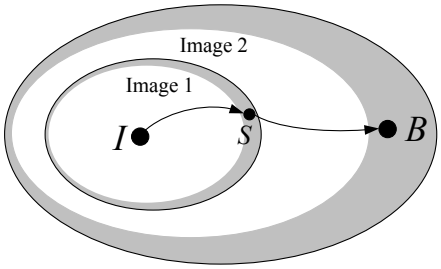


Fig. 2. Interpolation has erroneously reached a bad state.

### A. Simple Induction

Simple induction is a method of proving that a property holds for all reachable states. It is easy to formulate and often executes quickly. It is an incomplete method but can be strengthened by the introduction of inductive invariants.

First, we define some basic notation. If $S$ is a state and $p$ a property then $S \models p$ shall mean that $S$ satisfies the property $p$. No guarantees are made for other states. We also refer to states $XS$ as any state that can be reached from $S$ in one state transition. In general, $XS$ is not unique.

In simple induction, a property $p$ is proven in a two step process:

**Base Case:** Show $\forall$ initial states $i$, that $i \models p$.
**Inductive Step:** Show $\forall\, S$ s.t. $S \models p$, $\forall\, XS$, that $XS \models p$

This is typically implemented by unrolling the transition relation and then formulating a SAT problem to check both conditions of the proof.

The technique is incomplete since there are properties that hold in every reachable state which simple induction will fail to prove. For example, Figure 1 shows a state transition graph on which a proof of $p$ will fail. The shaded states are unreachable, but because of these unreachable states $\exists\, S$ and $XS$ such that $S \models p$ and $XS \not\models p$. The inductive step fails.

**Algorithm 1** Modified simple induction.

```
1: // Let p be the property to be proved.
2: if ∃ initial state i, i ⊭ p then return "falsified"
3: if ∃ possibly reachable S, XS s.t. S ⊨ p, XS ⊭ p then
4:     if can prove S or XS unreachable then // Section IV
5:         record new invariants, goto 3 // See Section IV-D
6:     return "inconclusive"
7: end if
8: return "verified"
```

To prove $p$ in Figure 1, we may try $k$-step induction as described in [8], but in [6] we found this to be a very expensive solution. Instead, suppose we are able to find an inductive invariant that shows that either $S$ or $XS$ is unreachable. If known-unreachable states are disallowed from entering the inductive step (with an extra constraint on the SAT solver), then simple induction will be able to prove $p$. This is demonstrated in Algorithm 1 where the standard simple induction algorithm is improved by the addition of lines 4 and 5. What is needed is a tool that will generate specific inductive invariants that can demonstrate the unreachability of $S$ or $XS$. Such a tool is described in this paper.

### B. Interpolation

Interpolation is a method that has been found useful for unbounded verification of safety properties. It works by using an over-approximation to the image operation. By applying this image operator iteratively starting from the initial state, either a fixed point is reached or a bad state is encountered. If a fixed point is reached, it is an over-approximation to the set of reachable states such that no bad states are contained in this approximation. The design is verified.

If an approximate image contains a bad state, interpolation may have found a counterexample. However there is in general no way to know if this is a real counterexample or a spurious one. Take for example Figure 2. Two image operations are applied to the initial state $I$, and a bad state $B$ is found in the second image. The shaded region represents the over-approximation inherent in the image operation, and the white shows the true image of the reachable states. We do not know which states of the image lie in the over-approximation, nor do we know if the over-approximated states are reachable. In Figure 2 we have the error trace $I \rightarrow S \rightarrow B$, but if either $S$ or $B$ lie in the over-approximation and is not truly reachable then the counterexample is spurious.

**Algorithm 2** Modified interpolation.

```
1: // Let p be the property to be proved.
2: set parameters for over-approximate image operator
3: {S} := I
4: while (1) do
5:     {S}' := {S} ∪ approxImage({S})
6:     if {S}' == {S} then return "verified"
7:     if ∃ a bad state "near" {S} then
8:         if {S} == I then return "falsified"
9:         if can prove s unreachable then // See Section IV
10:            record new invariants, goto 7 // See Section IV-D
11:        tighten over-approximation parameters, goto 3
12:    end if
13:    {S} := {S}'
14: end while
```

Unless specific conditions are met (line 8 of Algorithm

2), the interpolation algorithm has no way of knowing if a counterexample is spurious or true. Therefore, it discards all work up to that point and begins anew with a tighter approximation to the image operator. This restart is costly, and it is a major hot-spot in the performance of the algorithm.

In this work, we have augmented interpolation to call our invariant finding tool as shown in Algorithm 2. On lines 9 and 10, it will find specific inductive invariants that imply that a state along the error trace is unreachable. If invariants are found, the error trace must be spurious, and interpolation is free to proceed without the costly restart on line 11.

## IV. The Proof Graph

This section describes the basics of our tool to automatically find and prove useful inductive invariants. A graph structure called the *proof graph* is the core of our method.

### A. Basic Definitions

The proof graph is a bipartite directed graph with the following node types:

- States in the sequential design. In practice this is a cube of states, but to simplify this discussion, consider only a single state. This constraint will be relaxed in Section V-C.
- Sets of candidate properties. These properties are yet to be proved, but if we can prove them then they are invariants. In this discussion, consider the properties to be chosen from a specific domain. More details on the implemented domain (implications) and details on how to find candidate properties will be given later in Section V-B.

The root of the graph is a single state node. This corresponds to the user-specified state that should be proved unreachable. This root node comes from an outside source – in this work it is a state along the error trace in interpolation. The leaves, i.e. the nodes without outgoing edges, are property sets.

The meaning of the graph lies in its connectivity, specifically in the meanings of edges from states to properties and from properties to states. Being a bipartite graph, there are no other edge types.

*Definition 1 (Edges to Properties):* Let a directed edge from a state $S$ to a set of properties $\{P\}$ mean that:

$$\forall\, p \in \{P\}, \quad S \not\models p$$

That is, all properties $\{P\}$ fail to hold in $S$.

The properties $\{P\}$ may or may not hold in all reachable states, but if any such $p \in \{P\}$ can be proved then $S$ is unreachable. We refer to $\{P\}$ as a set of *covering properties* for $S$.

*Theorem 1 (Proving a State Unreachable):* Let a property $p$ fail in a state $S$ ($S \not\models p$). If $p$ is proved to hold in every reachable state then $S$ is unreachable.

Therefore the structure $S \rightarrow \{P\}$ provides a method to show $S$ unreachable.

*Definition 2 (Edges to States):* Let a directed edge from a set of properties $\{P\}$ to a state $S$ mean that:

$$\forall\, p \in \{P\}, \quad \exists \text{ a successor state } XS \text{ such that } (p \models S) \wedge (p \not\models XS)$$

That is, all properties hold in $S$ but fail in a successor state of $S$.

In the structure $\{P\} \rightarrow S$, $S$ is the reason that the inductive proof of $\{P\}$ was not successful. In fact, $S$ is the counterexample to the inductive hypothesis of the proof.

Proving $S$ to be unreachable is a necessary but not sufficient condition for proving a $p \in \{P\}$. Clearly, the proof of $p \in \{P\}$ cannot succeed if $S$ may be reachable. Conversely, if $S$ is known to be unreachable, we have no evidence that a proof of $p \in \{P\}$ will fail. However, another counterexample $S'$ may exist.

### B. An Example

Figure 3 shows an example of a proof graph and how it might evolve over time as our algorithm is run. A sample execution is given here.

1) Suppose interpolation reaches a bad state and $S_0$ is a state on the error trace. We would like to show that $S_0$ is unreachable. (See Section III-B.)
2) Our tool is called to prove that $S_0$ is unreachable. We set $S_0$ to be the root of our graph, and through simulation we generate the covering properties $\{P_0\}$. In this simulation, we select properties that appear to hold in every reachable state but fail in $S_0$. This gives us Graph (1) in Figure 3. (See Section V-B.)
3) We attempt to prove the properties $\{P_0\}$ by simple induction. Suppose that this proof fails, and there are three counterexamples $S_1$, $S_2$, and $S_3$ in the inductive hypothesis. Each counterexample is responsible for disproving a subset of $\{P_0\}$, and the proof technique as implemented in [6] will result in these subsets being pair-wise disjoint. We therefore split $\{P_0\}$ into $\{P_{0_1}\}$, $\{P_{0_2}\}$, and $\{P_{0_3}\}$ such that:
   - $\{P_{0_1}\} \cap \{P_{0_2}\} = \emptyset$, $\{P_{0_1}\} \cap \{P_{0_3}\} = \emptyset$, $\{P_{0_2}\} \cap \{P_{0_3}\} = \emptyset$
   - $\{P_{0_1}\} \cup \{P_{0_2}\} \cup \{P_{0_3}\} = \{P_0\}$
   - $\forall\, j \in \{1,2,3\}$, $S_j$ causes the inductive proof of $\{P_{0_j}\}$ to fail.

   Recording this information in the proof graph gives us Graph (2) in the figure. The existence of these counterexamples does not imply that the properties $\{P\}$ are not true but instead that we need more invariants to prove them. (See Sections III-A and IV-E.)
4) Next, cover $S_1$, $S_2$, and $S_3$ with properties, giving us $\{P_1\}$, $\{P_2\}$, and $\{P_3\}$ respectively. These properties provide a way to show that the new states are unreachable. (Like in Step 2, see Section V-B.)

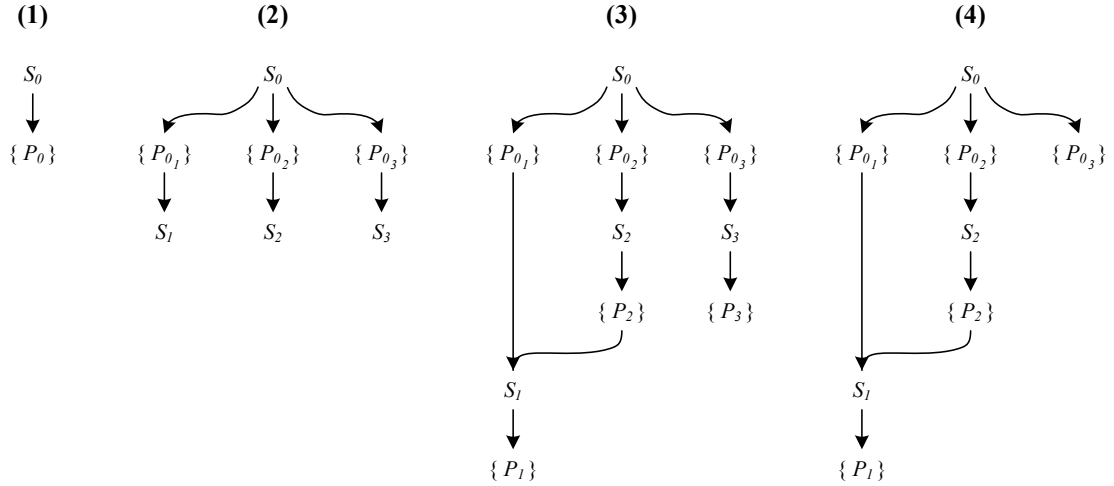


Fig. 3. Sample evolution of a proof graph over time.

5) By simulating each new state, we can check to see if it might be responsible for a future failure to prove any of the property sets. Suppose we find that $S_1$ is a counterexample in the inductive proof of every $p_2 \in \{P_2\}$. This induces the edge $\{P_2\} \rightarrow S_1$, and the result is Graph (3) in the figure. (See Section V-D.)
6) We attempt to prove the properties $\{P_3\}$. Suppose we find at least one property to be true for all reachable states. Now $S_3$ is known to be unreachable, and it can be removed from the proof graph. This gives Graph 4. (See Section IV-D.)
7) Attempt the proof of $\{P_{0_3}\}$ again. This proof was first attempted in Step 3, but now the reason that the original proof failed, $S_3$, is gone and we may re-attempt the proof. Suppose that this time we find that at least one property holds for all reachable states. This implies that $S_0$, the root node, is unreachable. At this point, we have achieved our objective and we may return the new invariants to the calling routine, interpolation in this case. (See Section IV-F.)

### C. Selecting Which Properties to Prove

The proof graph in general contains several property set nodes, and the tool must pick one single node to attempt as the next proof. Selecting that node is fairly simple once some basic properties of the proof graph are explored.

*Theorem 2 (Proofs on Leaves Only):* Given the sets of properties $\{P_0\}$ and $\{P_1\}$, a state $S$, and the graph structure $\{P_0\} \rightarrow S \rightarrow \{P_1\}$. If no $p_1 \in \{P_1\}$ are proved to hold in every reachable state then $\forall p_0 \in \{P_0\}$, it is not possible to prove $p_0$ by simple induction.

*Proof:* If $\exists p_1 \in \{P_1\}$ that has been proved, then that would be a guarantee that $S$ is unreachable. However, because no such proved properties exist, the reachability of $S$ is unknown. To be conservative, we must allow $S$ to be a counterexample in the inductive step of the proof of the properties $\{P_0\}$. Therefore, the proof will fail $\forall p_0 \in \{P_0\}$. ■

The above theorem defines an order in which the proofs must be attempted. Specifically, if a property node has an outgoing edge to a state then any proof attempt is in vain. In a chain of the graph, only the leaves (nodes without outgoing edges) may be considered as proof candidates. The situation is a bit more complex for a cycle however.

*Theorem 3 (Cycles in The Graph):* Suppose there are property sets $\{P_0\}, \ldots, \{P_n\}$, unique states $S_0, \ldots, S_n$, and the cyclic graph structure $\{P_0\} \rightarrow S_0 \rightarrow \cdots \rightarrow \{P_n\} \rightarrow S_n \rightarrow \{P_0\}$.

If $\exists j \in \{0, \ldots, n\}$ such that $\forall p_j \in \{P_j\}$, $p_j$ cannot be proved by simple induction
then $\forall k \in \{0, \ldots, n\}, \forall p_k \in \{P_k\}$, $p_k$ cannot be proved by simple induction

*Proof:* The failure to prove $\{P_j\}$ results in not being able to prove $\{P_{(j-1 \mod n)}\}$ by Theorem 2. This establishes a base case of the inductive proof of this theorem.

Now let $k \in \{0, \ldots, n\}$ and suppose $\{P_{(k+1 \mod n)}\}$ cannot be proved. By Theorem 2, $\{P_k\}$ cannot be proved. Theorem 3 is now proved by induction. ■

Theorem 3 says that in a cycle with $n$ property set nodes, we must attempt to prove the union of all the property sets at the same time. This simple induction will either successfully prove $\geq n$ properties or 0 properties because if any properties hold for all reachable states then at least one property in each set must be true.

Cycles must be treated differently from leaves in that the union of the cycle nodes must be proved simultaneously. However, this can be generalized as illustrated in the following example:

1) Suppose the current proof graph is that shown in Graph (1) of Figure 4.
2) Suppose we find that $S_0$ can act as the counterexample in the inductive hypothesis for all $p_1 \in \{P_1\}$. This induces

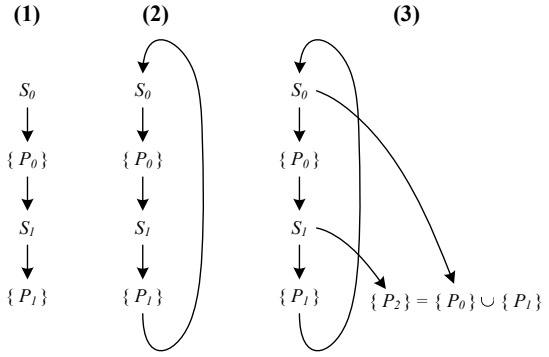


Fig. 4. A cycle has developed in the proof graph.

a cycle in the graph as shown in Graph (2).

3) Now create a new leaf node $\{P_2\} = \{P_0\} \cup \{P_1\}$ and insert it into the proof graph. This records the following information:
   - Both $\{P_0\}$ and $\{P_1\}$ must be proved at the same time.
   - A successful proof will imply that both $S_0$ and $S_1$ are unreachable.

   The updated proof graph is shown in Graph (3).

If cycles are abstracted as illustrated in Figure 4 then a proof of the union of the property sets in the cycle is equivalent to a proof of the new leaf node. After this generalization is made, only the leaves in the proof graph are eligible for an inductive proof.

Suppose a proof graph has multiple leaves, and one leaf must be selected for the simple induction prover. The unique leaf to be given to the induction engine is selected as follows:

- Let $d(\cdot)$ denote the distance from the root node to a property set node. That is, $d(\{P\})$ is the number of edges in the shortest directed path from the root node to $\{P\}$. Given this metric, the property set $\{P\}$ which minimizes $d(\{P\})$ should be selected because it requires fewer inductive proofs to achieve the overall goal, to prove that the root node is unreachable.
- Ties should be broken by selecting the $\{P\}$ with the greatest cardinality. In the absence of other information about the design, this property set has the greatest chance of having at least one property successfully proved.

### D. Upon a Successful Proof

Suppose a property set $\{P\}$ in the proof graph has been selected, given to the simple induction prover, and a property $p \in \{P\}$ has been proved. This proof is independent from any assumptions, and the proved property is guaranteed to hold for all reachable states. The property is therefore used to simplify all future problems, both simple induction and interpolation. Enforcement of the property can be accomplished by the addition of constraint clauses in each respective SAT instance. This extra clause is maintained throughout the remainder of the execution, effectively utilizing the new invariant in all future problems.

This successful proof also allows the proof graph to be pruned. Theorem 1 implies that all state node parents of $\{P\}$ are now known to be unreachable. These can be removed from the proof graph, along with any dangling nodes that result. This can create new property set leaves in the graph, enabling the proofs of some property sets to be re-attempted. This happened in Step 6 of the example shown in Section IV-B.

### E. Upon an Unsuccessful Proof

Suppose in attempting to prove $\{P\}$, the simple induction engine fails to prove any of the properties. From Section III-A we know that we can help simple induction by proving that the counterexample state that satisfies the inductive hypothesis is unreachable.

In failing to prove a set of properties, simple induction will produce a set of counterexamples $\{S_0, S_1, \ldots, S_n\}$. In this case, for each $p \in \{P\}$, $\exists\, S_j$ such that $S_j$ is the reason the inductive proof of $p$ failed.

To accurately record the relationship between the properties $\{P\}$ and the states $\{S_0, S_1, \ldots, S_n\}$, $\{P\}$ must be split into $n$ subsets, one for each of $\{S_0, S_1, \ldots, S_n\}$. We modify the proof graph by splitting $\{P\}$, adding $\{S_0, S_1, \ldots, S_n\}$ along with edges to demonstrate the failed proofs, and lastly we find covering properties $\{P_j\}$ for each new $S_j$. This is illustrated in Steps 3 and 4 of the example in Section IV-B.

One may view each new structure $S_j \rightarrow \{P_j\}$ as a subgraph rooted at $S_j$, the state that the induction engine wants to have shown unreachable. In this way, proving unreachable states for use in the simple induction solver is a sub-problem in the task of proving unreachable states for interpolation.

### F. Termination Conditions

The above process describes a proof graph that grows as counterexamples are discovered in inductive proofs and shrinks as properties are successfully proved. The proof graph oscillates in size until one of two termination conditions are satisfied:

- If we run out of proof candidates, the overall proof is impossible. This can happen if at some point there are no more leaves in the proof tree. In practice, this means that either the root state node being proved unreachable was in fact reachable or our candidate properties did not provide sufficient information to show this. Absence of leaves is not a guarantee that the root state is reachable.
- If a covering property of the root node is proven to hold for all reachable states then the proof graph algorithm will remove the root from the graph. If the root has been deleted, we can conclude that the root has been proven unreachable and we may stop.

## V. Advanced Details

This section is concerned with advanced details in a practical implementation of the method described above. Many details in the previous sections were left abstract, and here we provide the level of detail necessary to implement the proof graph algorithm.

**Algorithm 3** The proof graph algorithm.

```
1: // Let S_0 be the state to prove unreachable
2: root := S_0
3: cover root with properties
4: while (1) do
5:   if (root == ∅) then return "root unreachable"
6:   if (no leaves) then return "root may be reachable"
7:   {P} := selectBestLeaf()
8:   (proved, {S}) := simpleInductionProve({P})
9:   if proved then
10:    delete parents of {P}
11:  else
12:    for all counterexamples s ∈ {S} do
13:      make new proof graph node for s
14:      cover s with a new set of properties {P}
15:      Simulate S, try to break proofs of all property sets
16:      update proof graph
17:    end for
18:  end if
19: end while
```

### A. Choosing a Property Domain

The proof graph derivation makes use of sets of properties, but the domain of these properties to this point has not been specified. Although an implementation is free to use any domain, ours uses Boolean implications between gates. These were explored in [6]. Similarly, we allow implications to exist between any pair of gates in the design (not limited to the latches). However, we make the restriction that both gates are selected from the same time frame.

Implications were selected because they are expressive and easy to prove. They are much more numerous than alternatives such as node equivalences, yet are more manageable than relations between three nodes. Thus, they are a good trade-off between algorithmic efficiency and expressiveness.

Note that the use of implications as the property domain makes our implementation incomplete. For certain designs, $\exists$ states $S_1, S_2$, such that $\forall$ sets of implications $\{P\}$:

$$\left(S_1 \models \bigwedge_{p\in\{P\}} p\right) \iff \left(S_2 \models \bigwedge_{p\in\{P\}} p\right)$$

That is, implications cannot resolve all pairs of states and so would be unable to demonstrate that exactly one of $\{S_1, S_2\}$ is reachable because any set of implications would either be satisfied by both states or fail in both states.

### B. Selecting Covering Properties

In the proof graph algorithm, we often need to find the covering properties $\{P\}$ for a state $S$ that have a high probability of holding in all reachable states yet fail to hold in $S$. Implications are easy to check with random simulation, and we use simulation extensively to derive covering properties.

Because covering properties will need to be found for many states over the execution of the proof graph algorithm, it is advantageous to pre-compute a set of implications that have a high probability of holding in all reachable states. We refer to these as *candidate properties*. By constraining random simulation to only simulate reachable states, a set of likely implications can be extracted from a design. This set of implications can be refined through a mix of more random simulation and bounded model checking. In practice, after adequate tuning of the simulation and bounded model checking, this candidate set was of a manageable size for all the benchmarks examined in this paper. See the column "cand. props." in Table I below.

Using the candidate properties, the covering properties for any particular state $S$ can be easily derived. The design is simulated using $S$ as the input vector, and the candidate properties that fail in this simulation are selected as the covering properties for $S$.

### C. State Cubes

Section IV described a proof graph where state nodes consisted of exactly one state in the design. In an implementation this can be relaxed. Instead of being a single state, suppose that a state node represents a cube of states. Whenever possible, a state can be expanded to a cube of states that serves the same purpose, either breaking a simple induction proof, or leading interpolation to an error state. The covering properties can be selected such that any property failing in any state covered by the cube is considered.

Introducing state cubes changes the proof graph theory slightly by introducing a special type of graph cycle.

*Claim 1:* Let $\{P\}$ be a set of properties and $S$ a state cube (with $> 1$ minterms). If there is $S \rightarrow \{P\} \rightarrow S$ then the proof of $\{P\}$ is impossible.

This new condition must be checked in the code, and in practice occurs quite frequently. This is another source of graph compression because the property sets that are impossible to prove can be deleted.

### D. Handling the Large Number of Counterexamples

Inductively proving a set of properties $\{P\}$ can lead to a large number of counterexamples. This can lead to a blowup in the size of the proof graph, but a few tricks can help to make the size of the proof graph manageable.

**Counterexample Cubes:** The inductive hypothesis of an inductive proof is checked by forming a combinational circuit with a single output that is 1 if and only if the inductive hypothesis is violated for a given set of properties $\{P\}$. For every satisfying assignment found for this circuit, we can expand the assignment into a cube by simply discarding primary inputs not appearing on a controlling path in the circuit. This expanded cube is able to violate the inductive hypothesis for a larger subset of $\{P\}$ because in practice the counterexamples found are usually clustered. By using counterexample cubes, the total number of counterexamples found is reduced.

**Proof Graph Bounding:** In the proof graph, the primary goal is to prove the root state to be unreachable. To do

TABLE I

PERFORMANCE ON A SAMPLING OF HARD ACADEMIC PROBLEMS

| Design Properties | | | | Standard Interpolation | | | Interpolation + Proof Graph | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Design | And Gates | Latches | Prop. Type | MB | Sec. | Refines | MB | Sec. | Refines | Cand. Props. | Props. Proved |
| cmu_dme1_B | 236 | 61 | Unknown | 2484 | 7200 | 5 | 2487 | 7200 | 5 | 390 | 0 |
| cmu_dme2_B | 296 | 63 | Unknown | 2507 | 7200 | 7 | 2674 | 7200 | 7 | 604 | 0 |
| eijk_S1423_S | 902 | 159 | True | 2481 | 7200 | 1 | 139 | 77.93 | 0 | 10078 | 2400 |
| eijk_S208_S | 109 | 22 | True | 2451 | 7200 | 4 | 38 | 60.62 | 0 | 1668 | 454 |
| eijk_S208c_S | 111 | 23 | True | 2469 | 7200 | 7 | 30 | 59.04 | 0 | 1864 | 660 |
| eijk_S382_S | 230 | 57 | True | 2480 | 7200 | 5 | 228 | 102.17 | 0 | 23144 | 4176 |
| eijk_S420_S | 243 | 50 | True | 2500 | 7200 | 7 | 148 | 191.27 | 0 | 11000 | 2250 |
| eijk_S444_S | 240 | 57 | True | 2491 | 7200 | 5 | 224 | 507.37 | 0 | 38530 | 28972 |
| eijk_S838_S | 480 | 106 | True | 2570 | 7200 | 2 | 1199 | 370.63 | 0 | 152734 | 27308 |
| eijk_bs1512_S | 866 | 158 | Unknown | 2471 | 7200 | 0 | 2475 | 7200 | 0 | 46108 | 60 |
| eijk_bs3271_S | 1841 | 305 | Unknown | 1822 | 7200 | 1 | 2514 | 7200 | 1 | 6544 | 772 |
| irst_dme4_B | 593 | 124 | Unknown | 2515 | 7200 | 4 | 2558 | 7200 | 4 | 1894 | 0 |
| irst_dme5_B | 790 | 165 | Unknown | 2562 | 7200 | 5 | 528 | 7200 | 4 | 16590 | 0 |
| irst_dme6_B | 1181 | 245 | Unknown | 2564 | 7200 | 5 | 440 | 7200 | 0 | 126850 | 0 |
| nusmv_brp_B | 375 | 52 | Unknown | 2467 | 7200 | 1 | 805 | 7200 | 1 | 9140 | 1128 |
| nusmv_queue_B | 1310 | 84 | True | 2459 | 7200 | 1 | 95 | 151.78 | 0 | 3690 | 480 |
| nusmv_reactor_6_C | 903 | 76 | True | 2478 | 7200 | 7 | 52 | 140.66 | 0 | 6228 | 482 |
| vis_bakery_E | 284 | 25 | Unknown | 2454 | 7200 | 3 | 2582 | 7203.06 | 5 | 4018 | 626 |

this, only one path of successful proofs leading back to the root node is needed. This means that most of the counterexamples appearing in the proof graph are not needed to show that the root state is unreachable. Harnessing this idea, the implementation bounds the number of counterexample nodes appearing in the proof graph. If adding a new counterexample would cause the number of counterexample nodes to exceed this bound then the new counterexample is ignored. By using a sufficiently high counterexample node bound we can dramatically reduce the size of the proof graph and minimally impact the invariants found by our algorithm.

**Simulated Inductive Proofs:** Often a counterexample encountered previously can serve to break the inductive proof of a newly derived set of covering properties. Similarly, new counterexamples from the simple induction engine can often serve to break the proof of many sets of covering properties, more than just what the induction engine is currently processing. To detect these cases, random simulation is used to determine if a specific counterexample can satisfy the inductive hypothesis of a specific set of covering properties. This encourages re-use of old counterexamples rather than forcing new counterexamples to be derived, and in practice it works quite well. For an example, see Step 5 of Section IV-B.

## VI. EXPERIMENTAL RESULTS

For this work, we implemented two C++ plugins for the ABC Logic Synthesis and Verification System [10]. The first plugin implements the interpolation algorithm as described in [4], and the second plugin implements the invariant discovery method proposed in this paper. The plugins are interfaced as described in Algorithm 2 to provide inductive invariants for aiding interpolation.

We experimented with a suite of 154 academic designs that had been annotated with safety properties [12]. The designs ranged in size from 10 to 689 latches. After the designs were combinationally synthesized into And-Inverter Graphs, they had between 43 and 3716 And nodes. Each design in this benchmark suite contains a single safety property, which include 95 true properties, 34 false properties, and 25 properties of unknown nature.

The technique described in this paper can greatly speed-up model checking, but also it imposes some overhead to find and prove inductive invariants. The algorithm is best suited to run as an option in a verification package that can be invoked after more conventional methods have been exhausted. To emulate this type of flow, we attempted to verify all 154 benchmarks with standard interpolation. 132 finished in less than 10 minutes, and 18 failed to verify in 2 hours. It is on these 18 that we then applied our method.

Table I shows these 18 benchmarks on which interpolation times-out after 2 hours. As discussed in Section III-B, the most expensive part of the interpolation algorithm is the model refinement. The number of refinements done in the standard interpolation algorithm is shown in the table, and in some cases this number is quite high.

If specific inductive invariants can be found, then refinement can be avoided. In the last part of the table, we show the statistics for an implementation of interpolation that utilizes the proof graph. Whenever interpolation reaches a bad state, it will find and prove appropriate inductive invariants. In half of the designs, proving a small number of properties was sufficient to allow all model refinement steps to be skipped. Runtime was dramatically improved in those cases, and the inductive invariants proved to be the difference between a time-out and a successful verification run.

Interestingly, no false properties are present in Table I. The technique presented here favors true properties because for these, any trace into a bad state must contain unreachable states and so there is an opportunity to find invariants to cover those states. With a falsifiable property, the error trace will only contain reachable states and the proof graph method will waste resources attempting to show that these states are

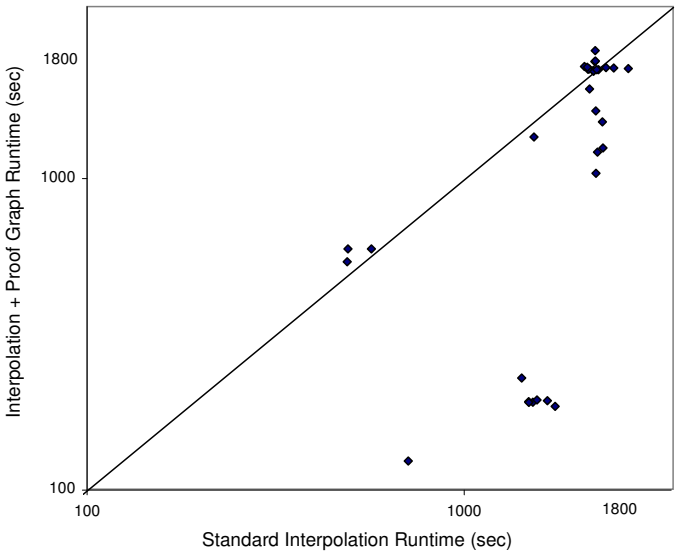


Fig. 5. Performance On A Sampling of Hard Industrial Problems

unreachable. Consequently, if there are indeed false properties in this subset of the benchmark suite, these probably appear as time-out cases.

Further experimentation was done using 43 industrial designs. Each contains multiple properties which are to be proven one-by-one. These designs were chosen because they are the hardest available, each having runtimes in excess of 8 minutes when processed with standard interpolation.

Figure 5 shows a scatter plot comparison of standard interpolation versus interpolation aided with the proof graph. Each verification run was given a time-out of 30 minutes, and the proof graph successfully prevented 5 of the designs from exceeding the time limit. Even on designs that did not time-out with standard interpolation, using the proof graph significantly helped the runtime.

Future work will include trying this technique on more industrial benchmarks, specifically ones that are harder for standard interpolation.

## VII. Conclusion

This paper outlined a method to automatically extract and prove inductive invariants. These invariants are used to show that a single, application-specified state is unreachable, and in this way only invariants that are immediately helpful to the problem at hand are found. This is accomplished by use of the proof graph, a structure whose theory and application were thoroughly explored here.

The proof graph shows that there is a fine relationship between different properties of a design. Some can be independently proved by simple induction. Others require that disjoint sets of properties be proved in a particular order. Some properties can only be proved in conjunction with other properties. Proving local properties with simple induction was also explored in our previous work [6], but this work provides us with a more complete understanding of this proof process. Specifically, the proof graph is an effective tool to capture all of the dependencies between the inductive proofs of different property sets.

This technique was applied to the interpolation algorithm, and the feasibility of modifying this algorithm to request and utilize specific invariants was demonstrated. It should be possible to incorporate our proof graph algorithm into any tool that needs specific invariants, and thus the methods presented in this work could have widespread application.

## Acknowledgments

The authors would like to acknowledge the Semiconductor Research Corporation (SRC) for their support through contracts 1361.001 and 1444.001. Also, we would like to thank Sanjit Seshia for his guidance in the class where this project began.

## References

[1] G. Hasteer, "Efficient Equivalence Checking in a Modular Design Environment," PhD Thesis from University of Illinois at Urbana-Champaign, 1998.

[2] M. Prasad, A. Biere, A. Gupta, "A Survey of Recent Advances in SAT-Based Formal Verification," in Software Tools for Technology Transfer (*STTT*), Vol. 7, No. 2, 2005. pp. 156-173

[3] J. Baumgartner, "Integrating FV Into Main-Stream Verification: The IBM Experience," Tutorial Given at *FMCAD* 2006

[4] K.L. McMillan, "Interpolation and SAT-Based Model Checking." in Proc. *CAV* 2003, pp. 1-13.

[5] N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan, and K.L. McMillan, "An Analysis of SAT-based Model Checking Techniques in an Industrial Environment," in *CHARME* 2005.

[6] M.L. Case, A. Mishchenko, and R.K. Brayton, "Inductively Finding a Reachable State Space Over-Approximation," in *IWLS* 2006.

[7] C. A. J. van Eijk, "Sequential Equivalence Checking without State Space Traversal," In Proceedings of *DATE*, February 1998.

[8] P. Bjesse and K. Claessen, "SAT-based Verification without State Space Traversal," in *FMCAD*, 2000.

[9] M. Wedler, D. Stoffel, and W. Kunz, "Exploiting state encoding for invariant generation in induction-based property checking," in *ASP-DAC*, Jan. 2004.

[10] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/

[11] R.K. Brayton, "The real reason why McMillan's construction is an interpolant." ERL Technical Report, EECS Dept., UC Berkeley.

[12] Property Checking Benchmark Suite, http://www.cs.chalmers.se/ een/Tip/

[13] Niklas Een, Niklas Sorensson, MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[14] F. Lu, M. K. Iyer, G. Parthasarathy, and K. T. Cheng, "An efficient sequential SAT solver with improved search strategies," in *DATE* 2005.