Reconsidering CEGAR: Learning Good Abstractions without Refinement [*]

Anubhav Gupta
Carnegie Mellon University, Pittsburgh
anubhav@cs.cmu.edu

Edmund Clarke
Carnegie Mellon University, Pittsburgh
emc@cs.cmu.edu

Abstract

Abstraction techniques have been very successful in model checking large systems by enabling the model checker to ignore irrelevant details. Most abstraction techniques in literature are based on refinement*. We introduce the notion of* broken traces *which capture the* necessary and sufficient *conditions for the existence of an error path in the abstract model. We formulate abstraction as* learning *the abstract model from* samples *of broken traces. Our iterative algorithm for abstraction-based model checking is not based on refinement and can generate the* smallest *abstract model that proves the property. We present an implementation of this algorithm for the verification of safety properties on gate-level net-lists with* localization abstraction*. Experimental results prove the viability of our techniques.*

1. Introduction

Model checking [1] is a formal verification technique that automatically decides whether a model satisfies a temporal property by an exhaustive state-space traversal. Our approach focuses on *safety properties*, which specify that there are no paths to some error states from the initial states in the model.

The major capacity bottleneck for model checking is the state-space size, therefore the state-space size is a good characterization for the *size of a model.* Abstraction techniques [2] have been very successful in combating the state-explosion problem. The idea behind abstraction is to build a smaller model, called the abstract model, by grouping together multiple states in the concrete model into a single abstract state. The abstract model preserves all the behaviors of the concrete model by allowing a transition from an abstract state $\hat{s}$ to an abstract state $\hat{t}$ if there is some concrete state in $\hat{s}$ that has a concrete transition to some concrete state in $\hat{t}$. This ensures that if the model checker proves the safety property on the abstract model, it also holds on the concrete model. However, this can introduce a path to an error state in the abstract model, even when there is no such path in the concrete model. Abstraction techniques search over a predefined set of abstract models, looking for one that proves that property and is small in size.

This paper introduces the notion of *broken traces* which capture the *necessary and sufficient* conditions for the existence of an error path in the abstract model. This is the motivation behind our abstraction methodology: we compute the smallest abstract model that *eliminates* all broken traces. This corresponds to the smallest abstract model that can prove the property. The naive method of computing this model by generating and *eliminating* all broken traces is infeasible because the set of broken traces is too large to enumerate. Instead, we *learn* this model by generating a set of sample broken traces such that the abstract model that eliminates the broken traces in this set also eliminates all other broken traces. Starting with an empty set, we iteratively generate this set of samples. In each iteration of the loop, we compute an abstract model that eliminates all broken traces in the sample set, and then use the counterexample produced by model checking this abstract model to guide the search for new broken traces that are not eliminated by the current abstract model. The loop terminates when no counterexample is present in the abstract model, or a broken trace corresponding to a real bug is generated. We evaluate our algorithm on a large set of industrial circuits, and compare it against the separation-based refinement approach in [3] and the proof-based abstraction techniques in [4, 5]. The results indicate that our technique generates much smaller abstract models, leading to better overall performance and a more robust behavior on harder benchmarks.

2. Related Work

Starting with Kurshan's *localization reduction* [6], there has been a lot of work in automatically generating good abstract models [7, 8, 3, 9, 10, 4, 11, 12, 13, 14, 15, 16]. Many of these techniques follow the *Counterexample-Guided Abstraction Refinement* (CEGAR) framework. These techniques start with an initial abstract model and iteratively *add more constraints (refinement)* to eliminate *spurious* counterexamples, until the property holds on the abstract model or a counterexample is found on the concrete model. The refinement of the abstract model and the search for a concrete counterexample are guided by the abstract counterexamples produced by the model checker.

The technique in [7, 3, 9, 4, 12] uses BDDs or SAT-solvers to identify the *failure-state*, which is the last state in the longest prefix of the abstract counterexample that has a corresponding path in the concrete model. It then adds a set of constraints that eliminate the abstract transition from the failure-state by splitting it into multiple abstract states. The methods in [10, 16] are similar, except that instead of the longest prefix, they look for a minimal spurious sub-trace [10] or the longest suffix [16]. The drawback of this strategy is that it focuses its efforts on a single abstract state instead of the whole counterexample. A smaller abstract model that

* This research was sponsored by the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) and the Army Research Office (ARO).

eliminates the counterexample can be generated by splitting multiple abstract state in the counterexample. Moreover, identification of the failure-state involves building an unrolling of the concrete model, which is expensive. Our approach fixes these drawbacks by analyzing all the abstract states and by never building an unfolding of the concrete model.

The techniques presented in [13, 14] use a *game-theoretic* approach to eliminate spurious counterexamples. They analyze the abstract model to identify latches that can steer the abstract model away from the error states. The approach in [8] simulates the abstract counterexample on the concrete model using 3-valued simulation, and looks for latches that conflict with their values in the counterexample. The method in [11] finds latches that are assigned the same value in multiple counterexamples. All these approaches use a heuristic to identify a set of candidate latches to add to the abstract model, and then greedily add latches from this list until the abstract counterexamples are eliminated. These approaches provide no guarantees on the size of the abstract model. Our approach, on the other hand, computes the smallest abstract model that can prove the property.

The disadvantage of any refinement-based strategy, is that once some irrelevant constraint is added to the abstract model, it is not removed in subsequent iterations. As the model checker discovers longer abstract counterexamples, the constraints that were added to eliminate the shorter counterexamples might become redundant. Refinement-based techniques do not identify and remove these constraints from the abstract model. This drawback is present in a refinement-based strategy irrespective of the technique that is used to eliminate spurious counterexamples. The proof-based abstraction technique presented in [5, 12] tries to alleviate this problem by building a fresh abstract model at each iteration. However, the abstract model is computed from the proof of unsatisfiability produced by SAT-solvers, and the SAT-solvers internally work like a refinement engine. Our approach is not based on refinement. In our iterative loop, the abstract model generated in the next iteration is not necessarily a refinement of the previous abstract model.

Refinement minimization [8] is used by various approaches to reduce the size of the abstract model. However, it is an expensive operation, and it can only guarantee *local minimality.* There has been some work in extracting the smallest unsatisfiable subset of a set of clauses [17, 18]. In theory, these techniques can be combined with the proof-based abstraction method [5] to generate small abstract models. However, in practice, these techniques can only be applied to instances with a small number of variables and clauses, and therefore do not scale to real-world systems.

3. Abstraction in Model Checking

We start with a brief description of the use of abstraction in model (for more details refer to [2]). We restrict our attention to *safety properties*. A system is modeled by a transition system $M = (S, I, R, E)$ where S is the set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the set of transitions, and $E \subseteq S$ is the set of error states. The *size of a model* $M = (S, I, R, E)$ is defined as $|S|$. An abstraction function $h : S \to \hat{S}$ is a surjection which maps a concrete state in S to an abstract state in $\hat{S}$.

Definition 3.1. *The* abstract model $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E})$ *corresponding to a concrete model* $M = (S, I, R, E)$ *and an abstraction function* h *is defined as follows:*

1. $\hat{S} = \{\hat{s} \mid \exists s.\ s \in S \wedge h(s) = \hat{s}\}$.
2. $\hat{I} = \{\hat{s} \mid \exists s.\ I(s) \wedge h(s) = \hat{s}\}$
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) \mid \exists s_1.\ \exists s_2.\ R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$
4. $\hat{E} = \{\hat{s} \mid \exists s.\ E(s) \wedge h(s) = \hat{s}\}$

The essence of abstraction is the following preservation theorem[2], which is stated without proof.

Theorem 3.1. *Let* $\hat{M}$ *be an abstract model corresponding to* M*. Then if the error states are not reachable in* $\hat{M}$*, then the error states are also not reachable in* M*.*

The aim of an abstraction framework is to find an abstraction function h, such that the property holds on the corresponding abstract model. Abstraction techniques search over a predefined (possibly infinite) set H of abstraction functions. For example, for *localization abstraction* with *visible/invisible* latches [3], the set H consists of all functions that map a concrete state to its projection over some subset of latches. Among all $h \in H$ that prove the property, we ideally want to identify the one that corresponds to the smallest abstract model.

Note that the converse of Theorem 3.1 is not true. Even if there is a path from an initial state to an error state in the abstract model, the error states might be unreachable in the concrete model. In this case, the abstract counterexample generated by the model checker is *spurious*, i.e. it does not correspond to a concrete path. The current abstraction function is too coarse to validate the specification, and a new abstraction function is computed to eliminate this spurious behavior.

$$\begin{array}{lll}
\mathbf{init}(x) := 0; & \mathbf{init}(y) := 0; & \mathbf{init}(z) := 1; \\
\mathbf{next}(x) := x; & \mathbf{next}(y) := \neg y; & \mathbf{next}(z) := \neg x \vee \neg y; \\
\mathbf{SPEC}\ \ \mathbf{AG}z & &
\end{array}$$

Figure 1: An SMV model ($\mathcal{M}$).

Consider the SMV model $\mathcal{M}$ in Figure 1. The model has 3 boolean state elements, x, y and z. The property holds on the model. The state space of the model is $S = \mathcal{B}^3$, where $\mathcal{B} = \{0, 1\}$. Consider the abstraction function $h : \mathcal{B}^3 \to \mathcal{B}^2$ such that:

$$h(\{x, y, z\}) = \{y, z\} \tag{1}$$

This abstraction function corresponds to *localization abstraction* that removes the latch x from the abstract model, which is shown in Figure 2. Model checking of this abstract model produces a spurious counterexample:

$$\{0, 1\} \to \{1, 1\} \to \{0, 0\} \tag{2}$$

Definition 3.2. *The abstraction function* h' *is a* refinement *of* h *if* $\forall s_1, s_2 \in S$, $h'(s_1) = h'(s_2)$ *implies* $h(s_1) = h(s_2)$.

```
INPUT x;
init(y)  := 0;     init(z)  := 1;
next(y) := ¬y;    next(z) := ¬x ∨ ¬y;
SPEC  AGz
```

Figure 2: An abstract model for $\mathcal{M}$ (Figure 1).

4. Broken Traces

Definition 4.1. *Given a model $M = (S, I, R, E)$ and an abstraction function h, a* broken trace $\mathcal{T}$ *on M for h is a sequence of pairs of concrete states $\langle(s_1, t_1), \dots (s_m, t_m)\rangle$, such that*

1. *$I(s_1)$, i.e. s_1 is an initial state.*
2. *$\forall 1 \le i \le m.\ h(s_i) = h(t_i)$, i.e, s_i and t_i lie in the same abstract state.*
3. *$\forall 1 \le i < m.\ R(t_i, s_{i+1})$, i.e, $t_i \rightarrow s_{i+1}$ is a concrete transition.*
4. *$E(t_m)$, i.e. t_m is an error state.*

A broken trace $\langle(s_1, t_1), \dots (s_m, t_m)\rangle$ is said to *break* at cycle i if $s_i \ne t_i$. If a broken trace has no breaks, it corresponds to a counterexample $C = \langle s_1, \dots s_m\rangle$ on the concrete model. Figure 3 shows a broken trace on $\mathcal{M}$. This broken trace breaks at cycle 2.

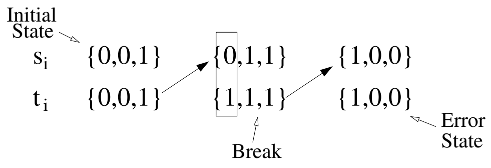

Figure 3: A broken trace on $\mathcal{M}$ (Figure 1) for the abstraction function defined in (1). A state is an assignment to $\{x, y, z\}$.

Theorem 4.1. *Given a model $M = (S, I, R, E)$ and an abstraction function h, there exists a counterexample on the abstract model corresponding to h if and only if there exists a broken trace on M for h (See Figure 4).*

Proof. (IF) Assume that $\mathcal{T}$ is a broken trace on M for h. Let $\mathcal{T} = \langle(s_1, t_1), \dots (s_m, t_m)\rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E})$ be the abstract model corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \hat{s}_2, \dots \hat{s}_m\rangle$, where $\hat{s}_i = h(s_i)$. Since $I(s_1)$, by Definition 3.1, we have $\hat{I}(\hat{s}_1)$. By Definition 4.1, $R(t_i, s_{i+1})$. Therefore, by Definition 3.1, $\hat{R}(h(t_i), h(s_{i+1}))$, i.e. $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$. Since $E(t_m)$, by Definition 3.1, we have $\hat{E}(\hat{s}_m)$. Hence, $\hat{C}$ is an counterexample on $\hat{M}$.

(ONLY IF) Assume that $\hat{C}$ is counterexample on the abstract model $\hat{M}$ corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \hat{s}_2, \dots \hat{s}_m\rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E})$. Since $\hat{I}(\hat{s}_1)$, by Definition 3.1, there exists a state s_1 such that $I(s_1)$. Since $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$, by Definition 3.1, there exist states t_i and s_{i+1} such that $h(t_i) = \hat{s}_i$, $h(s_{i+1}) = \hat{s}_{i+1}$ and $R(t_i, s_{i+1})$. Since $\hat{E}(\hat{s}_m)$, by Definition 3.1, there exists t_m such that $E(t_m)$. Thus, $\mathcal{T} = \langle(s_1, t_1), \dots (s_m, t_m)\rangle$ is a broken trace on M for h. □

Definition 4.2. *An abstraction function h, and the corresponding abstract model, are said to* eliminate *a broken trace $\langle(s_1, t_1), \dots (s_m, t_m)\rangle$ if $\exists 1 \le i \le m.\ h(s_i) \ne h(t_i)$.*

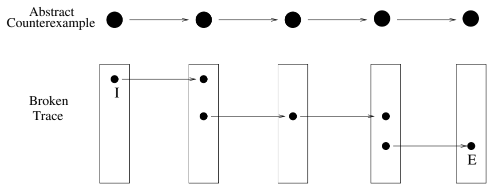

Figure 4: An abstract counterexample and a broken trace over the corresponding concrete states. This figure illustrates Theorem 4.1.

For example, the abstract counterexample in (2) corresponds to the broken trace in Figure 3. The abstraction function $h(\{x, y, z\}) = \{x\}$ eliminates the broken trace in Figure 3, because $h(\{0, 1, 1\}) = \{0\}$ and $h(\{1, 1, 1\}) = \{1\}$.

Theorem 4.1 says that the existence of a broken trace on a concrete model for an abstraction function h is a *necessary and sufficient* condition for the existence of a counterexample on the abstract model corresponding to h. This is the motivation behind our abstraction strategy. *We compute the smallest abstract model that eliminates all broken traces on the concrete model.* Theorem 4.1 implies that this is the smallest abstract model that can prove the property.

5. Learning Abstractions

The naive method of computing the abstract model by generating and eliminating all broken traces is infeasible, because the set of broken traces is too large to enumerate. Instead, we *learn* this model by generating a set of sample broken traces such that the abstract model that eliminates the broken traces in this set also eliminates all other broken traces.

For example, the broken trace samples in Figure 5 are eliminated by the abstraction function $h(\{x, y, z\}) = \{x, z\}$. This abstraction function also eliminates the broken trace in Figure 3. The property holds on the corresponding abstract model shown in Figure 6, and therefore by Theorem 4.1, h eliminates all broken traces in $\mathcal{M}$.

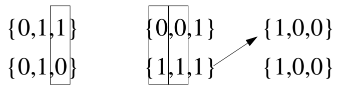

Figure 5: Sample broken traces on $\mathcal{M}$ (Figure 1). A state is an assignment to $\{x, y, z\}$.

```
INPUT y;
init(x)  := 0;     init(z)  := 1;
next(x) := x;     next(z) := ¬x ∨ ¬y;
SPEC  AGz
```

Figure 6: An abstract model that proves the property for $\mathcal{M}$ (Figure 1).

Figure 7 is a simplified view of our overall strategy. Starting with an empty set, we iteratively generate this set of samples. In each iteration of the loop, we compute an abstract model that eliminates all broken traces in the sample set, and then use the coun-

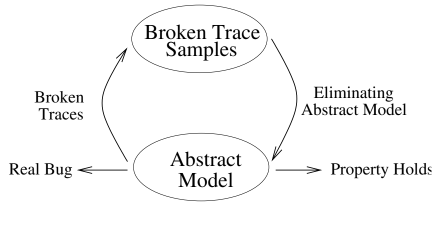

Figure 7: Learning Abstract Models.

terexample produced by model checking this abstract model to guide the search for new broken traces that are not eliminated by the current abstract model. *The abstract model generated in the next iteration from the augmented set of samples is not necessarily a refinement of the previous abstract model.* The loop terminates when: 1) No counterexample is present in the abstract model (property holds); 2) A broken trace with no breaks is generated (property does not hold).

6. The LEARNABS Algorithm

Algorithm 1 Learning Abstractions for Model Checking

```
LEARNABS ($M, \varphi$)
 1: $B = \{\}$;
 2: while (1) do
 3:     $h$ = ComputeAbstractionFunction($B$);
 4:     $\hat{M}$ = BuildAbstractModel($M, h$);
 5:     if $MC(\hat{M}, \varphi) = TRUE$ then return 'TRUE';
 6:     else Let $C$ be the counterexample produced by $MC$;
 7:     for ($n = 1; n \leq N; n = n + 1$) do
 8:         $\mathcal{T}$ = GenerateBrokenTrace($M, C$);
 9:         if $\mathcal{T}$ has no breaks then return 'FALSE';
10:         else $B = B \cup \{\mathcal{T}\}$;
```

The pseudo-code of our *Learning Abstractions*(LEARNABS) algorithm is shown in Algorithm 1. The *ComputeAbstractionFunction* function (line 3) computes an abstraction function that eliminates all broken traces in the sample set B (see Section 8). The *BuildAbstractModel* function (line 4) builds the abstract model corresponding to h (see Section 7). If the model checker proves the property on the abstract model (line 5), the algorithm returns TRUE. Otherwise, the *GenerateBrokenTrace* function (line 8) generates N broken trace samples corresponding to the abstract counterexample C (see Section 9). If a real bug is found in the process (line 9), the algorithm returns FALSE. Otherwise, the loop is repeated with the augmented set of samples (lines 10).

6.1. Termination

Since the broken trace(s) generated in a particular iteration of the loop are not eliminated by the current abstraction function h, the same abstraction function will not be computed in subsequent iterations. Thus, termination of the LEARNABS is guaranteed if the following conditions are met: 1) The *ComputeAbstractionFunction* function generates abstraction functions from a finite set H; 2) The set H contains the identity abstraction function $h(s) = s$, for which the abstract model is the same as the concrete model. These assumptions apply to many practical scenarios, including localization abstraction, and predicate abstraction [19] over a finite set of predicates.

7. Our Abstraction Functions

We used *localization abstraction* with *visible/invisible* latches [3]. We partition the set of latches into two sets: the set of *visible* latches ($\mathcal{V}$) and the set of *invisible* latches ($\mathcal{I}$). The corresponding abstraction function, denoted by $h_{\mathcal{V}}$, maps a concrete state to its projection over $\mathcal{V}$. The size of the abstract model is $2^{\mathcal{V}}$. The *Build-AbstractModel* function simply removes the logic that defines the latches in $\mathcal{I}$, and replaces these latches with inputs [3].

8. Computing the Abstraction Function

For a broken trace $\mathcal{T} = \langle s_1, t_1, \dots s_m, t_m \rangle$, the *eliminating set* $E_{\mathcal{T}}$ consists of all latches r such that for some $1 \leq i \leq m$, the states s_i and t_i differ on the value of r. The broken trace $\mathcal{T}$ is eliminated by an abstraction function $h_{\mathcal{V}}$ if $E_{\mathcal{T}} \cap \mathcal{V} \neq \{\}$. For example, the broken traces in Figure 5 are eliminated by abstraction functions corresponding to $\mathcal{V} = \{x, z\}$, $\mathcal{V} = \{y, z\}$, and $\mathcal{V} = \{x, y, z\}$. Given a set B of broken trace samples, we want to compute the smallest set $\mathcal{V}$ such that $h_{\mathcal{V}}$ eliminates all broken traces in B. This computation corresponds to the *minimum hitting-set problem*, which is an NP-complete problem. We formulate this as an Integer Linear Program(ILP), and solve it using an ILP-solver. We also implemented various polynomial-time algorithms that compute an approximation of the minimum hitting-set.

9. Generating Broken Traces

A SAT-solver implements a function SAT[F] that returns an arbitrary satisfying assignment for the boolean formula F. We enhanced the SAT-solver to implement *'SAT with hints'*. The function SATH[F,$\mathcal{H}$] takes as input a boolean formula F and a set $\mathcal{H}$ of assignments to a subset of variables in F. It returns satisfying assignment for F that agrees with the assignments in $\mathcal{H}$ on 'many' variables. This is achieved by forcing the SAT-solver to first decide on the literals corresponding to the variable assignments in $\mathcal{H}$. Thus, the satisfying assignment will disagree with $\mathcal{H}$ on a variable v only if v is forced to a different value by a conflict.

Theorem 4.1 says that if there is a counterexample on the abstract model, there exists a broken trace(s) on the concrete model for the corresponding abstraction function. The *GenerateBrokenTrace* function (Algorithm 2) generates these broken traces. Starting with a concrete initial state s_1 (line 2), it successively finds a concrete transition corresponding to each of the abstract transitions in the counterexample (line 4). The hints to SATH ensure that at cycle i, a state t_i different from s_i is picked only if s_i does not have a transition to some concrete state in $\hat{s}_{i+1}$. This helps in reducing the size of the eliminating set for the broken trace. A broken trace with a smaller eliminating set is better because it helps the sampling loop to converge faster [3]. Multiple samples are generated by randomizing the selection of assignments to the inputs.

Note that our approach does not perform BMC on the concrete model. The SAT-solver works on a single frame of the transition relation, thus it can potentially handle much larger designs. We cannot replace the SAT-solver with a circuit simulator, because the circuit outputs (latches) are constrainted to lie in the corresponding abstract state in the counterexample. A circuit-simulator, on the other hand, only permits constraints on the inputs.

Algorithm 2 Generating Broken Traces

```
GenerateBrokenTrace($M, C$)
1: Let $C = \langle \hat{s}_1, \ldots \hat{s}_m \rangle$
2: $s_1 = \text{SAT}[\, I(s_1) \wedge (h(s_1) = \hat{s}_1)\, ]$;
3: for $(i = 1; i < m; i = i + 1)$ do
4:     $(t_i, s_{i+1}) = \text{SATH}[\, R(t_i, s_{i+1}) \wedge (h(t_i) = \hat{s}_i) \wedge (h(s_{i+1}) = \hat{s}_{i+1}),\ \{t_i = s_i\}\, ]$;
5: $t_m = \text{SATH}[\, E(t_m) \wedge (h(t_m) = \hat{s}_m),\ \{t_m = s_m\}\, ]$;
6: Return $\mathcal{T} = \langle (s_1, t_1), \ldots (s_m, t_m) \rangle$;
```

10. Eliminating One Counterexample: LEARNABS vs. Splitting Failure State

Many abstraction techniques eliminate a spurious abstract counterexample as follows: they identify an abstract *failure* state [7] by a forward (or backward) simulation of the counterexample on the concrete model and then remove the abstract transition from (or to) the failure state by splitting the failure state into multiple abstract states. The drawback of this technique is that it focuses on a single abstract state. A smaller abstract model that eliminates the counterexample can be generated by splitting multiple abstract states of the counterexample. Moreover, this technique does not guarantee that the counterexample is eliminated. We illustrate these shortcomings through some examples in the context of the approach based on *separating deadend and bad states* [7, 3, 4]. We also illustrate how our approach fixes these shortcomings.

10.1. Deadend/Bad States

The set of concrete paths that corresponds to the counterexample $\langle \hat{s}_1, \hat{s}_2, \ldots \hat{s}_m \rangle$ is given by

$$\psi_m = \{\langle s_1 \ldots s_m \rangle \mid I(s_1) \wedge \bigwedge_{i=1}^{m-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=1}^{m} h(s_i) = \hat{s}_i\} \tag{3}$$

If the counterexample is spurious, ψ_m is empty. The *failure* index f, $f < m$ is the maximal index such that ψ_f is satisfiable. Given f, $\langle \hat{s}_1, \ldots \hat{s}_f \rangle$ is the longest prefix of the counterexample that corresponds to a concrete path. The abstract state $\hat{s}_f$ is called the *failure* state (see Figure 8). The states d_f, such that there exists some $\langle d_1 \ldots d_f \rangle$ in ψ_f, are called the *deadend* states. By definition, the deadend states are reachable from the initial states but have no concrete transition to $h^{-1}(\hat{s}_{f+1})$.

Since there is an abstract transition from $\hat{s}_f$ to $\hat{s}_{f+1}$, there is a non-empty set of transitions ϕ_f from $h^{-1}(\hat{s}_f)$ to $h^{-1}(\hat{s}_{f+1})$ that agree with the counterexample. The set of transitions ϕ_f is given by

$$\phi_f = \{\langle s_f, s_{f+1} \rangle \mid R(s_f, s_{f+1}) \wedge h(s_f) = \hat{s}_f \wedge h(s_{f+1}) = \hat{s}_{f+1}\} \tag{4}$$

The states b_f, such that there exists some $\langle b_f, b_{f+1} \rangle$ in ϕ_f, are called the *bad* states. By definition, bad states have a concrete transition to a state in $h^{-1}(\hat{s}_{f+1})$.

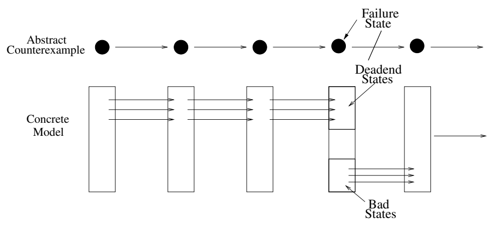

Figure 8: The spurious abstract counterexample breaks at the failure state. Deadend and bad states lie in the same abstract state.

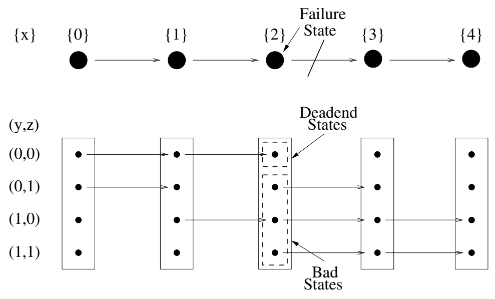

Figure 9: $\mathcal{V} = \{x\}, \mathcal{I} = \{y, z\}$. The counterexample breaks at abstract state $\{2\}$.

There is a spurious abstract transition from $\hat{s}_f$ to $\hat{s}_{f+1}$ because the deadend and bad states lie in the same abstract state. This suggests the following strategy to eliminate the counterexample: *generate a new abstract model that puts the deadend and bad states into separate abstract states.*

10.2. Drawbacks of Separating Deadend/Bad

Example: Consider the abstract counterexample in Figure 9, with $\mathcal{V} = \{x\}$ and $\mathcal{I} = \{y, z\}$. Figure 10 is an abstract model obtained by separating state set S_1 from S_2, and S_3 from S_4, where $S_1 = \{\{2, 0, 0\}\}$, $S_2 = \{\{2, 1, 0\}, \{2, 1, 1\}\}$, $S_3 = \{\{3, 0, 1\}\}$, and $S_4 = \{\{3, 1, 0\}, \{3, 1, 1\}\}$. S_1 and S_2 lie in the abstract state $\{2\}$, while S_3 and S_4 lie in the abstract state $\{3\}$. This separation can be achieved by making y visible, and the new abstract model eliminates the counterexample. Note that this abstract model does

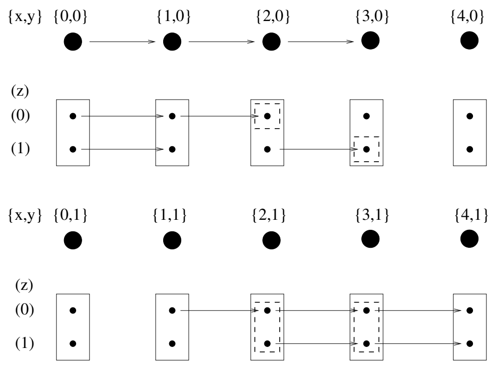

Figure 10: An abstract model that eliminates the counterexample in Figure 9 by splitting multiple abstract states.

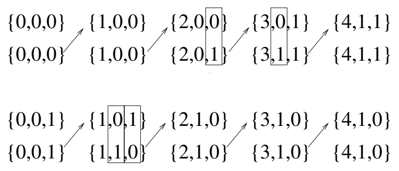

Figure 11: Broken trace samples corresponding to the counterexample in Figure 9. A state is an assignment to $\{x, y, z\}$.

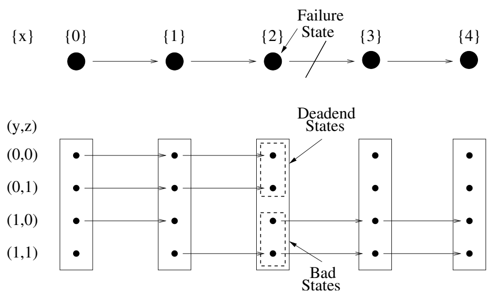

Figure 12: $\mathcal{V} = \{x\}, \mathcal{I} = \{y, z\}$. The counterexample breaks at abstract state $\{2\}$.

not separate the deadend and bad states. Separating deadend and bad states for this example would require both y and z to be visible, thereby adding unnecessary latches to the abstract model. This example illustrates that *separating deadend and bad states is not necessary for eliminating the counterexample.* Figure 11 illustrates some broken trace samples corresponding to this counterexample. All these samples are eliminated by making y visible. Thus our technique does not have this drawback.

Example: Consider the abstract counterexample in Figure 12, with $\mathcal{V} = \{x\}$ and $\mathcal{I} = \{y, z\}$. Figure 13 is an abstract model that puts the deadend and bad states into separate abstract states. The counterexample is not eliminated from the new abstract model, because the bad states are reachable in this model. This example illustrates that *separating deadend and bad states is not sufficient to eliminate the counterexample.* Figure 14 illustrates some broken traces corresponding to this counterexample. In order to eliminate these, we need to make both y and z visible, which also eliminates the

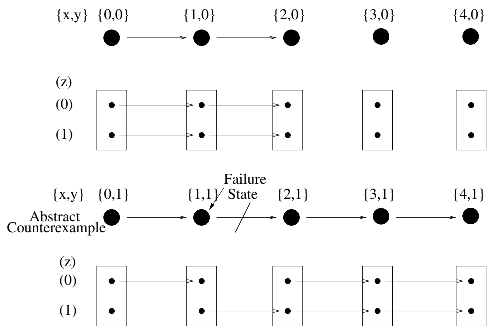

Figure 13: An abstract model obtained by separating deadend and bad states in Figure 12. The counterexample is not eliminated.

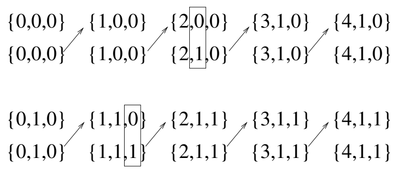

Figure 14: Broken trace samples corresponding to the counterexample in Figure 12. A state is an assignment to $\{x, y, z\}$.

counterexample. Thus our technique does not have this problem.

11. Eliminating all counterexamples: LEARNABS vs. Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) framework [6] is a common strategy for automatically generating abstract models. Starting with an initial abstraction function, this technique refines it in each iteration to eliminate one or multiple spurious counterexamples. The smallest abstract model that eliminates a set of counterexamples is not necessarily a refinement of the smallest abstract model that eliminates a subset of this set. Thus, a refinement-based strategy cannot guarantee the smallest abstract model that proves the property. The proof-based abstraction technique presented in [5, 12] tries to alleviate this problem by building a fresh abstract model at each iteration. They perform BMC on the concrete model up to the length of the counterexample. The abstract model consists of the gates that are used by the SAT-solver to prove unsatisfiability of the BMC instance. The problem with this strategy is that SAT-solvers internally work like a refinement engine, the size of the abstract model depends on the variable splitting order used by the SAT-solver. The LEARNABS algorithm is not based on refinement, and therefore it does not have these drawbacks.

Example: Consider the application of a CEGAR-based strategy to $\mathcal{M}$ (Figure 1). In the first iteration of the CEGAR loop, an abstract model is generated to eliminate counterexamples of length 1. This could produce an abstract model with $\mathcal{V} = \{y, z\}$ (see Figure 2), which is one of the smallest models that eliminates all counterexamples of length 1. In the next iteration, a counterexample of length 2 will be generated and this adds the latch x to $\mathcal{V}$. At this point, the latch y is not needed, but it ends up being part of the abstract model because CEGAR is based on refinement. Even if a fresh abstract model is generated using a SAT-solver, the abstract model will contain the latch y if the SAT-solver splits on the variables corresponding to y first. Our approach, on the other hand, guarantees that y is not included in the final abstract model (see Figure 6).

12. Experiments Results

We implemented a model checker for gate-level net-lists based on the LEARNABS algorithm. It uses zChaff [20] as the SAT-solver, and CPLEX [21] as the ILP-solver. Cadence SMV [22] is used as the BDD-based model checker for verifying the abstract models. We used 3 sets of benchmarks for our experiments: the *IU* benchmarks [3] from Synopsys, the *PJ* benchmarks derived from the PicoJava processor [5], and the *RB* benchmarks from the IBM Formal Verification Library [23]. The value of N in LEARNABS was set to 25. All experiments were performed on a 1.5GHz Athlon machine with 3GB RAM, running Linux. The following optimizations were implemented on top of LEARNABS.

12.1. Optimizations

Fine Grained Abstractions: The latch-level abstraction can be too coarse for larger circuits. We modified the *BuildAbstractModel* function (line 4) to perform BMC on the latch abstraction computed from the samples. The BMC instance is restricted with values from the counterexample. If the BMC instance is unsatisfiable, only the gates used in the proof of unsatisfiability are added to the abstract model [5]. If the BMC instance is satisfiable, it means that the counterexample has not been eliminated, and more samples are generated.

Eliminating All Counterexamples: Eliminating one counterexample in each iteration could lead to a lot of expensive model checking calls [24]. The LEARNABS loop can be modified to eliminate all counterexamples at the current depth. The modified loop performs (unrestricted) BMC on the current abstract model in the *BuildAbstractModel* function, and proceeds to the model checking step only if the BMC instance is unsatisfiable. If BMC produces a counterexample, the counterexample is used to generate more samples.

Reducing Number of Samples: It can be shown that computing the smallest abstract model is a Σ_2^P-complete problem [25]. Therefore, the LEARNABS algorithm could potentially generate an exponential number of samples before it terminates. In order to balance the time spent in computing the abstract model and in model checking, we added some simple heuristics to the *ComputeAbstractionFunction* function (line 3) to pick larger non-optimal separating sets if the sampling step takes too much time. Since the *GenerateBrokenTrace* function does not build a BMC unrolling, it might generate a large number of samples before it finds one that corresponds to a real bug. Therefore, if the number of samples at a particular depth reaches a threshold, we check for the presence of a real bug at that depth by performing BMC on the concrete model.

12.2. Results

Figure 15 shows the comparison of LEARNABS with the abstraction strategy based on separating deadend/bad states. The numbers for *Deadend/Bad* were obtained from [3] (these experiments were performed on the same machine as ours). The table shows the number of latches in the circuit *(reg)*; the counterexample length *(cex)* - *'T'* indicates that the property holds; the total running time *(time)*; the number of model checking calls *(itr)*; and the number of latches in the final abstract model *(abs)*. The LEARNABS algorithm generates smaller abstract models. For smaller benchmarks, model checking is not the bottleneck, and therefore the effort spent in generating a good abstract model does not result in a smaller overall running time. The LEARNABS algorithm performs better on the larger benchmarks.

Figure 16 compares the LEARNABS algorithm with the abstraction strategy based on the proof of unsatisfiability generated by SAT-solvers (SATPROOF) [4, 5, 12]. For both these techniques, we show results for the single-counterexample mode (indicated by 'S') that eliminates one abstract counterexample in each iteration; and the all-counterexample mode (indicated by 'A') that eliminates all counterexamples at the current depth in each iteration. The LEARNABS algorithm consistently generates smaller abstract models, and this translates to a better or similar runtime on most benchmarks. For the benchmarks with bugs, SATPROOF performs better because the BMC step is very efficient in identifying the error trace. The LEARNABS algorithm completes all the benchmarks, while SATPROOF cannot complete 4 benchmarks. On *IUP1*, the SAT-solver runs out of memory while trying to build an unrolling of the concrete model for depth 67 for SATPROOF (S), and 69 for SATPROOF (A).

circuit	reg	cex	Deadend/Bad			LEARNABS		
			time	itr	abs	time	itr	abs
IU30	30	10	6	3	20	8	10	14
IU35	35	19	23	4	21	61	70	16
IU40	40	19	34	5	22	37	35	15
IU45	45	19	39	5	22	34	31	17
IU50	50	19	57	5	22	67	22	16
IU55	55	10	59	3	20	13	10	13
IU60	60	10	77	3	20	34	9	14
IU65	65	10	80	3	20	30	10	13
IU70	70	10	69	3	20	25	10	13
IU75	75	10	23	4	21	22	11	15
IU80	80	10	26	4	21	26	10	14
IU85	85	10	28	4	21	25	9	14
IU90	90	10	28	4	21	21	10	13
IUP1	4494	T	>2hr	-	-	**1295**	18	8

Figure 15: Comparison of Deadend/Bad with LEARNABS.

13. Conclusion

We introduced the notion of broken traces and formulated abstraction as learning from samples of broken traces. We presented an iterative algorithm for abstraction-based model checking that is not based on refinement, and that computes the *smallest* abstract model that can prove the property. There are several future research directions to our work. We want to explore better heuristics for balancing the time spent in model checking and the time spent in computing a good abstraction. The current algorithm tries to map an abstract transition to a single concrete transition in the broken trace. This can be extended to multiple concrete transitions, which can be used to generate a hierarchy of abstract models. Together with predicate abstraction, our algorithm can be used for verification of high-level system descriptions and for software model checking. If the set of predicates is predefined, then the techniques presented in this paper apply directly. Otherwise, datamining techniques can be used to infer eliminating predicates from broken trace samples. The broken traces can also be used to identify which predicates are relevant at which location in the program.

References

[1] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. IBM Workshop on Logics of Programs*, 1981.

[2] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," *ACM Trans. Prog. Lang. Sys.*, vol. 16, no. 5, 1994.

[3] E. Clarke, A. Gupta, and O. Strichman, "SAT based counterexample-guided abstraction-refinement," *TCAD*, vol. 23(7), 2004.

circuit	reg	cex	SATPROOF (S)			LEARNABS (S)			SATPROOF (A)			LEARNABS (A)		
			time	itr	abs	time	itr	abs	time	itr	abs	time	itr	abs
PJ00	348	T	7	3	9	6	3	4	7	3	9	6	3	4
PJ01	321	T	6	3	3	5	2	0	6	3	3	5	2	0
PJ02	306	T	7	3	4	6	3	4	7	3	4	6	3	4
PJ03	306	T	6	3	4	5	3	4	6	3	4	6	3	4
PJ04	305	T	7	3	3	5	2	0	6	3	3	5	2	0
PJ05	105	T	9	7	34	35	8	28	9	7	34	34	8	28
PJ06	328	T	209	7	56	23	7	41	210	7	56	24	7	41
PJ07	94	T	18	11	36	13	7	32	20	10	36	14	7	32
PJ08	116	T	58	6	41	75	7	41	58	6	41	76	7	41
PJ09	71	T	8	6	31	4	6	25	8	6	31	4	6	25
PJ10	85	T	3	3	5	2	3	4	3	3	5	2	3	4
PJ11	294	T	11	5	12	5	2	0	12	5	12	5	2	0
PJ12	312	T	4	1	0	4	1	0	4	1	0	4	1	0
PJ13	420	T	10	5	13	8	3	8	10	5	13	8	3	8
PJ14	127	T	25	6	32	9	4	13	35	5	32	9	4	13
PJ15	355	T	184	5	42	14	5	23	185	5	42	15	5	23
PJ16	290	T	248	6	44	64	7	32	246	6	44	65	7	32
PJ17	212	T	2126	14	43	1869	20	29	5037	11	43	3685	14	31
PJ18	145	T	993	22	49	390	8	32	161	7	45	542	7	36
PJ19	52	T	>2hr	-	-	**18**	3	12	>2hr	-	-	**19**	3	12
RB05_1	313	31	11	10	24	141	17	13	17	6	12	137	11	13
RB09_1	168	T	1	4	9	1	3	4	1	4	9	1	3	4
RB10_1	236	T	3	5	9	1	4	3	3	5	9	1	4	3
RB10_2	236	T	3	6	11	2	4	3	3	6	11	2	4	3
RB10_3	236	T	5	7	34	2	5	5	5	7	34	1	5	5
RB10_4	236	T	8	10	23	1	4	5	6	8	22	2	4	5
RB10_5	236	T	1	4	7	2	4	4	2	4	7	2	4	4
RB10_6	236	T	3	5	7	2	4	4	2	5	7	2	4	4
RB11_2	242	T	>1hr	-	-	**128**	24	26	>1hr	-	-	**219**	14	30
RB14_1	180	T	37	7	47	3	5	15	37	7	47	3	5	15
RB14_2	180	T	>1hr	-	-	**1258**	60	37	334	11	87	179	13	38
RB15_1	270	9	2	7	8	17	9	4	2	7	8	13	9	4
RB16_1_1	1117	T	8	7	92	324	8	80	8	7	92	260	8	80
RB16_2_4	1113	5	4	5	40	61	5	32	4	5	40	48	5	32
RB26_1	608	T	1	1	0	1	1	0	1	1	0	1	1	0
RB31_2_1	111	T	>1hr	-	-	**38**	27	29	>1hr	-	-	**17**	12	26
IUP1	4494	T	*mem*	-	-	**1295**	18	8	*mem*	-	-	**151**	13	5

Figure 16: Comparison of SATPROOF and LEARNABS, in *single* (S) and *all* (A) counterexamples mode.

[4] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *FMCAD*, 2002.

[5] K. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *TACAS*, 2003.

[6] R. Kurshan, *Computer aided verification of coordinating processes*. Princeton University Press, 1994.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for model checking," *J. ACM*, 2003.

[8] D. Wang, P. H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal property verification by abstraction-refinement with formal, simulation and hybrid engines," in *DAC*, 2001.

[9] S. Barner, D. Geist, and A.Gringauze, "Symbolic localization reduction with reconstruction layering and backtracking," in *CAV*, 2002.

[10] S. Das and D. Dill, "Counterexample based predicate discovery in predicate abstraction," in *FMCAD*, 2002.

[11] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi, "Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation," in *TACAS*, 2003.

[12] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis." in *ICCAD*, 2003.

[13] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi, "Improving Ariadne's bundle by following multiple threads in abstraction refinement," in *ICCAD*, 2003.

[14] F. Y. Mang and P.-H. Ho, "Abstraction refinement by controllability and cooperativeness analysis," in *DAC*, 2004.

[15] G. D. H. Chao Wang and F. Somenzi, "Fine-grain abstraction and sequential don't cares for large scale model checking," in *ICCD*, 2004.

[16] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level predicate abstraction and refinement for verifying RTL verilog," in *DAC*, 2005.

[17] Lynce and Marques-Silva, "On computing minimum unsatisfiable cores," in *SAT*, 2004.

[18] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov, "AMUSE: a minimally-unsatisfiable subformula extractor," in *DAC*, 2004.

[19] S.Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *CAV*, 1997.

[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*, 2001.

[21] "CPLEX," www.cplex.com.

[22] K. McMillan, "Cadence SMV," Cadence Berkeley Labs, CA.

[23] E. Zarpas, "Simple yet efficient improvements of SAT based bounded model checking," in *FMCAD*, 2004.

[24] N. Amla and K. L. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *FMCAD*, 2004.

[25] C. Umans, "The minimum equivalent DNF problem and shortest implicants," in *FOCS*, 1998.