SAT-Based Counterexample-Guided Abstraction Refinement

Edmund M. Clarke, *Member, IEEE*, Anubhav Gupta, and Ofer Strichman

***Abstract*—We describe new techniques for model checking in the counterexample-guided abstraction-refinement framework. The abstraction phase "hides" the logic of various variables, hence considering them as inputs. This type of abstraction may lead to "spurious" counterexamples, i.e., traces that cannot be simulated on the original (concrete) machine. We check whether a counterexample is real or spurious with a satisfiability (SAT) checker. We then use a combination of 0-1 integer linear programming and machine learning techniques for refining the abstraction based on the counterexample. The process is repeated until either a real counterexample is found or the property is verified. We have implemented these techniques on top of the model checker NuSMV and the SAT solver Chaff. Experimental results prove the viability of these new techniques.**

***Index Terms*—Abstraction, model checking, satisfiability (SAT).**

I. Introduction

WHILE state-of-the-art model checkers can verify circuits with several hundred latches, many industrial circuits are at least an order of magnitude larger. Various conservative abstraction techniques can be used to bridge this gap. Such abstraction techniques preserve all the behaviors of the concrete system but may introduce behaviors that are not present originally. Thus, if a universal property (i.e., an ACTL* property) is true in the abstract system, it will also be true in the concrete system. On the other hand, if a universal property is false in the abstract system, it may still be true in the concrete system. In this case, none of the behaviors that violate the property in the abstract system can be reproduced in the concrete system. Counterexamples corresponding to these behaviors are said to be *spurious*. When such a counterexample is found, the abstraction can be refined in order to eliminate the spurious behavior. This process is repeated until either a real counterexample is found or the abstract system satisfies the property. In the latter case, we know that the concrete system satisfies the property as well, since the abstraction is conservative.

Manuscript received April 14, 2003; revised August 27, 2003 and November 10, 2003. This work was supported in part by the Semiconductor Research Corporation (SRC) under Contract 99-TJ-684, by the National Science Foundation (NSF) under Grant CCR-9803774, by the Office of Naval Research (ONR), and by the Naval Research Laboratory (NRL) under Contract N00014-01-1-0796. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, the U.S. government, or any other entity. This paper was recommended by Associate Editor C.-J. R. Shi.

E. M. Clarke and A. Gupta are with Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: emc@cs.cmu.edu; anubhav@cs.cmu.edu).

O. Strichman is with the Technion, Haifa 32000, Israel (e-mail: ofers@ie.technion.ac.il).

Digital Object Identifier 10.1109/TCAD.2004.829807

There are many known techniques, some automatic and some manual, for generating the initial abstraction and for abstraction refinement (for example, see Rushby's suggested architecture for using information derived from a model checker to refine predicate abstractions within a theorem prover [1]). The automatic techniques are more relevant to this paper because our method is fully automatic. Our methodology is based on an iterative abstraction-refinement process. Abstraction is performed by selecting a set of latches or variables and making them *invisible*, i.e., they are treated as inputs. In each iteration, we check whether the abstract system satisfies the specification with a standard ordered binary decision diagrams (OBDD)-based symbolic model checker. If a counterexample is reported by the model checker, we try to simulate it on the concrete system with a fast Boolean satisfiability (SAT) solver. In other words, we generate and solve a SAT instance that is satisfiable if and only if the counterexample is real. If the instance is not satisfiable, we look for the *failure state*, which is the last state in the longest prefix of the counterexample that is still satisfiable. Note that this process cannot be performed with a standard circuit simulator, because the abstract counterexample does not include values for all inputs.

We use the failure state in order to refine the abstraction. The abstract system has transitions from the failure state that do not exist in the concrete system. We eliminate these transitions by refining the abstraction, i.e., by making some variables visible that were previously invisible. The problem of selecting a small set of variables to make visible is one of the main issues that we address in this paper. It is important to find a small set in order to keep the size of the abstract state space manageable. This problem can be reduced to a problem of separating two sets of states (abstraction unites concrete states, and therefore refining an abstraction is the opposite operation, i.e., separation of states). For realistic systems, generating these sets is not feasible, both explicitly and symbolically. Moreover, the minimum separation problem is known to be NP-hard [2]. We combine *directed sampling* with integer linear programming (ILP) and machine learning techniques to handle this problem. By directed sampling we mean that rather than choosing samples randomly, we guide the search for samples in a way that guarantees a full separation of the states sets without explicitly computing them.

The rest of the paper is organized as follows. In the next section, we survey related work and also discuss the main differences between this paper and an early version of it that we published in [3]. In Section III, we briefly give the technical background of abstraction and refinement in model checking. In Section IV, we describe our counterexample-guided abstraction-refinement framework. We elaborate in that section on

0278-0070/04$20.00 © 2004 IEEE

how the counterexample is being checked and how we refine the abstraction. We also describe refinement as a learning problem. In Sections V and VI, we elaborate on our separation techniques and on alternative objective functions. These techniques are combined with the directed sampling technique, which is described in Section VII. We give experimental results in Section VIII, which prove the viability of our methods compared to a state-of-the-art model checker (Cadence SMV [4]). We conclude in Section IX.

II. Related Work

The closest work to the current one that we are aware of is described in Lu's thesis [5] and is more briefly summarized in [2]. Like the current work, they also use an automatic iterative abstraction-refinement procedure that is guided by the counterexample, and they also try to eliminate the counterexample by solving the state-separation problem. But there are three main differences between the two methods. First, their abstraction is based on replacing predicates of the program with new input variables, while our abstraction is performed by making some of the variables invisible (thus, we hide the entire logic that defines these variables). As we will later show, the advantage of this approach is that computing a minimal abstraction function becomes easy. Secondly, checking whether the counterexample is real or spurious was performed in their work symbolically, using OBDDs. We do this stage with a SAT solver, which for this particular task is extremely efficient (due to the large number of solutions to the SAT instance). Thirdly, they derive the refinement symbolically. Since finding the coarsest refinement is NP-hard, they present a polynomial procedure that, in general, computes a suboptimal solution. For some well-defined cases the same procedure computes the optimal refinement. We, on the other hand, tackle this complexity by considering only samples of the states sets, which we compute explicitly. In order to maintain optimality while still only checking samples, we suggest a method for guiding the sampling process in a way that allows us to efficiently compute an optimal refinement, as if we sample the entire set of states.

The work of Das *et al.* [6] should also be mentioned in this context, since it is very similar to [2], the main difference being the refinement algorithm. Rather than computing the refinement by analyzing the abstract failure state, they combine a theorem prover with a greedy algorithm that finds a small set of previously abstracted predicates that eliminate the counterexample. They add this set of predicates as a new constraint to the abstract model.

Previous work on abstraction by making variables invisible (this technique was used under different names in the past) includes the localization reduction of Kurshan [7] and others (see, for example [8] and [9]). The localization reduction follows the typical abstraction-refinement iterative process. It starts by making all but the property variables invisible. When a spurious counterexample is identified, it refines the system by making more variables visible. The variables made visible are selected according to the variable dependency graph and information that is derived from the counterexample. The candidates in the next refinement step are those invisible variables that are adjacent on the variable dependency graph to currently visible variables. Choosing among these variables is done by extracting information from the counterexample. Another relevant work is described by Wang *et al.* in [10]. They use three-valued simulation to simulate the counterexample on the concrete model and identify the invisible variables whose values in the concrete model conflict with the counterexample. Variables are chosen from this set of invisible variables by various ranking heuristics. For example, like localization, they prefer variables that are close to the currently visible variables in the variable dependency graph.

More recent research by Chauhan *et al.* [11] follows very similar lines to ours. Like our approach, they also look for the failing state with a SAT solver. But rather than analyzing the failing state, they derive information from the SAT solver that explains why the spurious counterexample cannot be simulated beyond this state on the concrete machine. More specifically, they build an unsatisfiability proof by joining conflict graphs in the SAT solver and make visible all the variables from the failing state that correspond to vertices in this graph. As a second step, they try to minimize this set by gradually making some of these variables invisible again and check whether this makes the instance satisfiable. The success of the second phase depends on the (arbitrarily chosen) order in which they remove the variables.

This approach has both advantages and disadvantages compared to ours. The main advantage is that their refinement step consists of solving one SAT instance and analyzing the proof of unsatisfiability of this instance. We, on the other hand, look for an optimal solution and therefore solve an optimality problem that can potentially take more time. By doing so, we hope to make the model checking step faster. In general, there is less arbitrariness in our procedure compared to theirs. In practice, it is hard to compare the two methods because of this arbitrariness. For example, it is possible that due to two equally good refinements that the two tools perform, the next counterexample that they need to analyze is different (the counterexamples that model checkers produce are chosen arbitrarily from an exponential number of options) and hence changes the results of the overall procedure. For this reason, it is possible that their tool occasionally finds smaller abstract models compared to ours. We will refer to this point further when describing our experimental results.

With this technique, they were able to outperform in most cases an early version of the current research that was published in [3]. For the current paper, we added several improvements that enable us now to perform better than [11] in some of the harder cases and comparable in the easier instances, as we report in Section VIII. The two main improvements are the following. First, rather than minimizing the number of latches that we choose to make visible, we now minimize the number of inputs to the abstract model. Since existentially quantifying out inputs is a major bottleneck in model-checking, this change made a significant improvement in our results. We describe this improvement in Section V-B. Second, we improved the directed sampling scheme that was occasionally a bottleneck in our system. We define a criterion for being a "good sample" in the sense that such samples lead to faster convergence of this stage, as we describe in Section VII-B. Both of these improvements require solving a SAT instance with some

optimality criterion. We chose a pseudo-Boolean solver (PBS) [12] as an optimization engine, because it is very efficient in solving the type of instances we consider, as we will explain in Section VIII.

III. Abstraction in Model Checking

We start with a brief description of the use of abstraction in model checking (for more details refer to [13]). Consider a program with a set of variables $V = \{x_1, \ldots, x_n\}$, where each variable x_i ranges over a nonempty domain D_{x_i}. Each state s of the program assigns values to the variables in V. The set of all possible states for the program is $S = D_{x_1} \times \cdots \times D_{x_n}$. The program is modeled by a transition system $M = (S, I, R)$ where

S	set of states;
$I \subseteq S$	set of initial states;
$R \subseteq S \times S$	set of transitions.

We use the notation $I(s)$ to denote the fact that a state s is in I, and we write $R(s_1, s_2)$ if the transition between the states s_1 and s_2 is in R.

An abstraction function h for the system is given by a surjection $h : S \to \hat{S}$, which maps a concrete state in S to an abstract state in $\hat{S}$. Given a concrete state $s_i \in S$, we denote by $h(s_i)$ the abstract state to which it is mapped by h. Accordingly, we denote by $h^{-1}(\hat{s})$ the set of states s such that $h(s) = \hat{s}$.

Definition 1: The *minimal abstract transition system* $\hat{M} = (\hat{S}, \hat{I}, \hat{R})$ corresponding to a transition system $M = (S, I, R)$ and an abstraction function h is defined as follows:

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

Intuitively, minimality means that $\hat{M}$ can start in state $h(s)$ if and only if M can start in state s, and $\hat{M}$ can transition from $h(s)$ to $h(s')$ if and only if M can transition from s to s'. We refer the reader to [14] for a detailed discussion of optimal abstractions in model checking.

For simplicity, we restrict our discussion to model checking of $\mathbf{AG}p$ formulas, where p is a nontemporal propositional formula. The theory can be extended to handle any safety property because such formulas have counterexamples that are finite paths.

Definition 2: A propositional formula p *respects* an abstraction function h if for all $s \in S, h(s) \models p \Rightarrow s \models p$.

The essence of conservative abstraction is the following preservation theorem [13], which is stated without proof.

Theorem 1: Let $\hat{M}$ be an abstraction of M corresponding to the abstraction function h and p be a propositional formula that respects h. Then, $\hat{M} \models \mathbf{AG}p \Rightarrow M \models \mathbf{AG}p$.

The converse of the above theorem is not true, however. Even if the abstract model invalidates the specification, the concrete model may still satisfy the specification. In this case, the abstract counterexample generated by the model checker is *spurious*, i.e., it does not correspond to a concrete path. The abstraction function is too coarse to validate the specification, and we need to refine it.

Definition 3: Given a transition system $M = (S, I, R)$ and an abstraction function h, h' is a *refinement* of h if

1) for all $s_1, s_2 \in S, h'(s_1) = h'(s_2)$ implies $h(s_1) = h(s_2)$;
2) there exists $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$ and $h'(s_1) \neq h'(s_2)$.

IV. Abstraction Refinement

Based on the above definitions, we now describe our *counterexample-guided abstraction-refinement* procedure. Given a transition system M and a safety property φ, we have the following.

1) Generate an initial abstraction function h.
2) Model check $\hat{M}$. If $\hat{M} \models \varphi$, then $M \models \varphi$. Return TRUE.
3) If $\hat{M} \not\models \varphi$, check the counterexample on the concrete model. If the counterexample is real, $M \not\models \varphi$. Return FALSE.
4) Refine h, and go to step 2).

The above procedure is complete for finite-state systems. Since each refinement step partitions at least one abstract state, the number of loop iterations is bounded by the number of concrete states. In the next subsections, we explain in more detail how we perform each step.

A. Defining an Abstraction Function

We partition the set of variables V into two sets: the set of *visible* variables which we denote by $\mathcal{V}$ and the set of *invisible* variables which we denote by $\mathcal{I}$. Intuitively, $\mathcal{V}$ corresponds to the part of the system that is currently believed to be important for verifying the property. The abstraction function h abstracts out the irrelevant details, namely the invisible variables. The initial abstraction in step 1) and the refinement in step 4) correspond to different partitions of the set of variables. As an initial abstraction, $\mathcal{V}$ includes the variables in the property φ. In each refinement step, we move variables from $\mathcal{I}$ to $\mathcal{V}$, as we will explain in Section IV-D.

More formally, let $s(x), x \in V$ denote the value of variable x in a state s. Given a set of variables $U = \{u_1, \ldots, u_p\}, U \subseteq V, s^U$ denotes the portion of s that corresponds to the variables in U, i.e., $s^U = (s(u_1) \ldots s(u_p))$. Let $\mathcal{V} = \{v_1, \ldots, v_k\}$. The partitioning of the set of variables into visible and invisible defines our abstraction function $h : S \to \hat{S}$. The set of abstract states is $\hat{S} = D_{v_1} \times \cdots \times D_{v_k}$ and the abstraction function is simply $h(s) = s^{\mathcal{V}}$.

B. Computing the Minimal Abstraction

For an arbitrary system M and abstraction function h, it is often too expensive or impossible to construct the minimal abstraction $\hat{M}$ [13]. However, our abstraction function allows us to compute $\hat{M}$ efficiently for systems where the transition relation R is in a functional form, e.g., sequential circuits. For these systems, $\hat{M}$ can be computed directly from the program text by removing the logic that defines the invisible variables and treating them as inputs. We prove that this operation results in a minimal abstraction.

Theorem 2: The circuit obtained by replacing invisible variables with nondeterministic values represents a minimal ab-

straction of the original circuit, corresponding to the abstraction function $h(s) = s^{\mathcal{V}}$.

Proof: The proof is based on the observation that we can quantify out the next state copy of the invisible variables if the transition relation is in functional form, because in that case the value of each next state variable does not depend on other next state variables.

Consider a sequential circuit with latches $\{x_1, \ldots, x_n\}$ and inputs $\{i_1, \ldots, i_q\}$. We use the standard notation x' to denote the next-state version of x. Let $s = (x_1, \ldots, x_n), s' = (x'_1, \ldots, x'_n)$ and $i = (i_1, \ldots, i_q)$. The transition relation R for the circuit can be expressed as

$$R(s, s') = \exists i \left(\bigwedge_{j=1}^{n} x'_j = f_{x_j}(s, i) \right)$$

where f_{x_j} is the functional definition of x'_j. By definition, the minimal abstract transition relation $\hat{R}$ for the circuit is given by

$$\hat{R}(\hat{s}, \hat{s}') = \exists s \exists s' (R(s, s') \wedge h(s) = \hat{s} \wedge h(s') = \hat{s}').$$

Substituting expressions for R and the abstraction function h, and splitting s and s' into visible and invisible parts yields

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{V}} \exists s^{\mathcal{I}} \exists s'^{\mathcal{V}} \exists s'^{\mathcal{I}} \exists i \left(\bigwedge_{x_j \in \mathcal{V}} x'_j = f_{x_j}(s^{\mathcal{V}}, s^{\mathcal{I}}, i) \wedge \bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(s^{\mathcal{V}}, s^{\mathcal{I}}, i) \wedge s^{\mathcal{V}} = \hat{s} \wedge s'^{\mathcal{V}} = \hat{s}' \right).$$

We can eliminate the quantification over the visible variables using the rule: $\exists a (f(a) \wedge a = b) \equiv f(b)$. Thus, we get

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists s'^{\mathcal{I}} \exists i \left(\bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i) \wedge \bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i) \right).$$

Since the left conjunct does not depend on $s'^{\mathcal{I}}$, we can push the quantification over $s'^{\mathcal{I}}$ inside, which gives us

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists i \left(\bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i) \wedge \exists s'^{\mathcal{I}} \left(\bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i) \right) \right).$$

The quantification over $s'^{\mathcal{I}}$ evaluates to TRUE because f_{x_j} does not depend on x'_j. Thus, the expression simplifies to

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists i \left(\bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i) \right).$$

Now we are left with existential quantification over the primary inputs and the invisible variables. This proves that the minimal abstract transition system corresponding to our abstraction can be derived syntactically from the original system by replacing the invisible variables with inputs. ■

C. Checking the Counterexample

For safety properties, the counterexample generated by the model checker is a path $\langle \hat{s}_1, \hat{s}_2, \ldots \hat{s}_m \rangle$. The set of concrete paths that corresponds to this counterexample is given by

$$\psi_m = \left\{ \langle s_1 \ldots s_m \rangle \,\middle|\, I(s_1) \wedge \bigwedge_{i=1}^{m-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=1}^{m} h(s_i) = \hat{s}_i \right\}. \quad (1)$$

According to Section IV-A, $h(s_i)$ is simply a projection of s_i to the visible variables. The right-most conjunct is therefore a restriction of the visible variables in step i to their values in the counterexample.

The counterexample is spurious if and only if the set ψ_m is empty. We check for that by solving ψ_m with a SAT solver. This formula is very similar in structure to the formulas that arise in bounded model checking (BMC) [15]. However, ψ_m is easier to solve because the path is restricted to the counterexample. Most model checkers treat inputs as latches, and therefore the counterexample includes assignments to inputs. While simulating the counterexample, we also restrict the values of the (original) inputs that are part of the definition (lie on the right-hand side) of the visible variables to the value assigned to them by the counterexample, which further simplifies the formula.

If a satisfying assignment is found, we know that the counterexample corresponds to a concrete path, which means that we found a real bug. Otherwise, we try to look for the "failure" index f, i.e., the maximal index $f, f < m$, 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. Our implementation performs a binary search over the range $1 \ldots m$ in order to find the highest value f such that ψ_f is satisfiable. Thus, the number of SAT instances we solve is bounded by $\log m$.

D. Refining the Abstraction

As in Section IV-C, let f denote the failure index. Let D denote the set of all states d_f such that there exists some $\langle d_1 \ldots d_f \rangle$ in ψ_f. We call D the set of *deadend* states. Intuitively, D denotes the set of reachable concrete states corresponding to the abstract failure state. By definition, there is no concrete transition from D 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 nonempty 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 defined as follows:

$$\phi_f = \{ \langle s_f, s_{f+1} \rangle \,|\, R(s_f, s_{f+1}) \wedge h(s_f) = \hat{s}_f \wedge h(s_{f+1}) = \hat{s}_{f+1} \} \quad (2)$$

Given the definition of h, ϕ_f represents all concrete paths from step f to step $f+1$, where the visible variables in these steps are restricted to their values in the counterexample. Let B denote the set of all states b_f such that there exists some $\langle b_f, b_{f+1} \rangle$ in ϕ_f. We call B the set of *bad* states (see Fig. 1). Intuitively, B is the set of concrete states corresponding to the failure state that have a transition to a concrete state in the next abstract state of the trace.

The counterexample exists because there is an abstract transition from s_f to s_{f+1} that does not correspond to any concrete

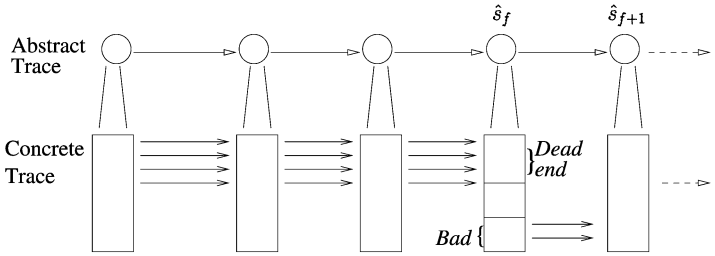


Fig. 1. Spurious counterexample corresponds to a concrete path that "breaks" in the failing state. Failing state unites concrete "deadend" and "bad" states.

transition. The transition exists because the deadend and bad states lie in the same abstract state. This suggests a mechanism to refine the abstraction. The abstraction h is refined to a new abstraction h' such that $\forall d \in D, \forall b \in B\ (h'(d) \neq h'(b))$. The new abstraction puts the deadend and bad states into separate abstract states and therefore eliminates the spurious transition from the abstract system.

E. Refinement by Separation and Learning

In the following definition, let $S = \{s_1 \dots s_m\}$ and $T = \{t_1 \dots t_n\}$ be two sets of states (binary vectors) representing assignments to a set of variables W.

Definition 4 (Separation of States With Sets of Variables): A set of variables $U = \{u_1 \dots u_k\}, U \subseteq W$ *separates* S from T if for each pair of states $(s_i, t_j), s_i \in S, t_j \in T$, there exists a variable $u_r \in U$ such that $s_i(u_r) \neq t_j(u_r)$.

Definition 5 (State Separation Problem): Given two sets of states S and T, as defined above, find a minimal set of variables $U = \{u_1 \dots u_k\}, U \subseteq W$ that separates S from T.

This definition is compatible with the intuitive definition given in [2].

Now, let D_I and B_I denote the restriction of D and B, respectively, to their invisible parts, i.e., $D_I = \{s^I \mid s \in D\}$ and $B_I = \{s^I \mid s \in B\}$. Let $H \in \mathcal{I}$ be a set of variables that separates D_I from B_I. The refinement is obtained by adding H to $\mathcal{V}$. Minimality of H is not crucial, rather it is a matter of efficiency. Smaller sets of visible variables make it easier to model check the abstract system, but can also be harder to find. In fact, it has been shown that computing the minimal separating set is NP-hard [2]. In Section V-B, we will show that in fact there is a better minimality criterion than simply the number of added variables to $\mathcal{V}$.

Lemma 1: The new abstraction function h' puts D and B in different abstract states in the abstract system.

Proof: Let $d \in D$ and $b \in B$. The refined abstraction function h' corresponds to the visible set $\mathcal{V}' = \mathcal{V} \cup H$. Since H separates D_I and B_I, there exists a $u \in H$ s.t. $d(u) \neq b(u)$. Thus, for some $u \in \mathcal{V}', d(u) \neq b(u)$. By definition, $h'(d) = (d(u_1) \dots d(u_k))$ and $h'(b) = (b(u_1) \dots b(u_k)), u_i \in \mathcal{V}'$. Thus, $h'(d) \neq h'(b)$. ∎

The naive way of separating the set of deadend states D from the set of bad states B would be to generate and separate D and B either explicitly or symbolically. Unfortunately, for systems of realistic size, this is usually not possible. For all but the simplest examples, the number of states in D and B is too large to enumerate explicitly. For systems with moderate complexity, these sets can be computed symbolically with OBDDs. Experience shows, however, that there are many systems for which this is not possible [2]. Moreover, even if it were possible to generate D and B, it would still be computationally expensive to identify the separating variables.

$$\text{Min } \sum_{i=1}^{|\mathcal{I}|} v_i$$

$$\text{subject to: } (\forall s \in S_{D_I})\ (\forall t \in S_{B_I}) \sum_{\substack{1 \le i \le |\mathcal{I}|, \\ s(v_i) \neq t(v_i)}} v_i \ge 1$$

Fig. 2. State separation with ILP.

Instead, we select *samples* from D and B and try to infer the separating variables for the entire sets from these samples. Naive sampling results in loss of data and, hence, loss of optimality. We will show in Section VII how a guided sampling scheme avoids the loss of optimality.

The idea of learning from samples has been studied extensively in the machine learning literature, and a number of learning models and algorithms have been proposed [16]. In the next section, we describe one of these algorithms and propose several techniques of our own for separating sets of samples of deadend and bad states, which we denote by S_{D_I} and S_{B_I}, respectively.

V. Separation as an ILP Problem

A. Basic 0-1 ILP Formulation

A formulation of the problem of separating S_{D_I} from S_{B_I} as a 0-1 ILP problem is depicted in Fig. 2.

The value of each Boolean variable $v_1 \dots v_{|\mathcal{I}|}$ in the ILP problem is interpreted as $v_i = 1$ if and only if v_i is in the separating set. Every constraint corresponds to a pair of states (s_i, t_j), stating that at least one of the variables that separates (distinguishes) between the two states should be selected. Thus, there are $|S_{D_I}| \times |S_{B_I}|$ constraints.

Example 1: Consider the following two pairs of states:

$$s_1 = (0,1,0,1) \quad t_1 = (1,1,1,1)$$
$$s_2 = (1,1,1,0) \quad t_2 = (0,0,0,1).$$

Let v_i correspond to the ith component of a state. Then, the corresponding ILP is

$$\text{Min } \sum_{i=1}^{4} v_i$$

subject to:

$v_1 + v_3 \ge 1$ // Separating s_1 from t_1

$v_2 \ge 1$ // Separating s_1 from t_2

$v_4 \ge 1$ // Separating s_2 from t_1

$v_1 + v_2 + v_3 + v_4 \ge 1$ // Separating s_2 from t_2.

$$\text{Min } \sum_{i=1}^{|\mathcal{F}|} v_i$$
Subject to:

$$1.\ \forall s \in S_{D_I}.\ \forall t \in S_{B_I}.\ \sum_{\substack{1 \le i \le |\mathcal{I}|,\\ s(v_i) \neq t(v_i)}} v_i \ge 1$$

$$2.\ \forall 1 \le i \le |\mathcal{I}|.\ \forall v_j \in FanIn(v_i).\ v_j - v_i \ge 0 \quad // \text{ Same as } v_i \rightarrow v_j$$

Fig. 3. 0-1 ILP formulation of the state separation problem, where the objective is to minimize the number of inputs in the abstract model.

The optimal value of the objective function in this case is 3, corresponding to one of the two optimal solutions (v_1, v_2, v_4) and (v_3, v_2, v_4).

B. Changing the Objective

The criterion of the minimum number of latches that separate a given set of samples is not necessarily the optimal one for faster model checking. Through experiments, we discovered that minimizing the number of added inputs to the abstract model predicts far better the complexity of model checking (one of the expensive stages in model checking is removing the quantifiers over all inputs). Let In denote the set of primary inputs. In order to find the set of separating variables that minimizes the number of inputs in the resulting abstract model, we derive a mapping $\mathcal{I} \rightarrow (2^{\mathcal{I} \cup In})$ from each invisible variable to the set of variables in its (first-layer) fan-in that are not yet in the model. Let $FanIn(v)$ denote this set for a variable v, and let $\mathcal{F} = \bigcup_{j=1}^{|\mathcal{I}|} FanIn(v_j)$. With ILP, we can now encode each variable $v \in \mathcal{I}$ with a new Boolean variable and add a constraint stating that if v is true, than so are all the variables in $FanIn(v)$. Minimizing over the sum of inputs gives us the desired result. In Fig. 3, we give a 0-1 ILP formulation of this problem.

Example 2: Continuing Example 1, suppose that we derive the following mapping of invisible variables to variables in their $FanIn$:

$$v_1 \rightarrow \{i_1, i_3\} \quad v_2 \rightarrow \{i_1, i_5\}$$
$$v_3 \rightarrow \{i_2\} \quad v_4 \rightarrow \{i_3, i_4\}.$$

The corresponding ILP is (here, we write the new set of constraints as propositional formulas rather than inequalities, for clarity)

$$\text{Min } \sum_{i=1}^{5} i_i$$
subject to:
$$\ldots\ldots \quad // \text{ same constraints as in Example 1}$$
$$v_1 \rightarrow i_1 \wedge i_3$$
$$v_2 \rightarrow i_1 \wedge i_5$$
$$v_3 \rightarrow i_2$$
$$v_4 \rightarrow i_3 \wedge i_4.$$

There are three possible satisfying assignments to the constraints that appeared in Example 1: $\{v_1, v_2, v_4\}$, $\{v_3, v_2, v_4\}$, and $\{v_1, v_2, v_3, v_4\}$. Only the first option minimizes the number of inputs to four.

In all our experiments, minimizing over the number of inputs turned out to be more efficient than simply minimizing the number of latches. In some cases, it enabled us to solve instances that we could not solve with the previous method in the given time and memory bounds.

VI. Separation Using Decision Tree Learning

The ILP-based separation algorithm outputs the minimal separating set. Since ILP is NP complete, we also experimented with a polynomial approximation based on decision trees learning (DTL). In this section, we formulate the basic separation problem as DTL (we could not find a natural way to formulate the objective function of Section V-B with this model). This technique is polynomial both in the number of variables and the number of samples, but does not necessarily give optimal results.

Learning with decision trees is one of the most widely used and practical methods for approximating discrete-valued functions. A DTL algorithm inputs a set of examples and outputs a tree, where each node represents a subset of the input examples. An example is described by a set of attributes and the corresponding classification. The algorithm generates a decision tree that classifies the examples. Each internal node in the tree specifies a test on some attribute, and each branch descending from that node corresponds to one of the possible values for that attribute. Each leaf in the tree corresponds to a classification.

Data is classified by a decision tree by starting at the root node of the decision tree, testing the attribute specified by this node, and then moving down the tree branch corresponding to the value of the attribute. The process is repeated for the subtree rooted at the branch until one of the leafs is reached, which is labeled with the classification.

The problem of separating S_{D_I} from S_{B_I} can be formulated as a DTL problem as follows:

- the attributes correspond to the invisible variables;
- the classifications are $+1$ and -1, corresponding to S_{D_I} and S_{B_I}, respectively;
- the examples are S_{D_I} labeled $+1$, and S_{B_I} labeled -1.

We generate a decision tree for this problem. From this tree, we extract all the variables present at the internal nodes. These variables constitute the separating set.

Lemma 2: The above algorithm outputs a separating set for S_{D_I} and S_{B_I}.

Proof: Let $d \in S_{D_I}$ and $b \in S_{B_I}$. The decision tree will classify d as $+1$ and b as -1. So, there exists a node n in the decision tree, labeled with a variable v, such that $d(v) \neq b(v)$. By construction, v lies in the output set. ■

Example 3: Going back to Example 1, the corresponding DTL problem has four attributes (v_1, v_2, v_3, v_4) and, as always, two classifications $(+1, -1)$. The set of examples E contains the following elements:

$$((0,1,0,1),+1) \quad ((1,1,1,1),-1)$$
$$((1,1,1,0),+1) \quad ((0,0,0,1),-1).$$

The tree appearing in Fig. 4 corresponds to the separating set (v_1, v_2, v_4).

A number of algorithms [16] have been developed for learning decision trees, e.g., ID3 [17] and C4.5 [18]. All these algorithms essentially perform a simple top-down greedy search through the space of possible decision trees. We implemented a simplified version of the ID3 algorithm, which is described in Fig. 5 [16]. At each recursion, the algorithm has to pick an attribute to test at the root. We need a measure of the quality of an attribute. We start with defining a quantity called *entropy* [16], which is a commonly used notion in information theory.

Definition 6: Given a set S containing $n_\oplus$ positive examples and $n_\ominus$ negative examples, the *entropy* of S is given by

$$\text{Entropy}(S) = -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus$$

where $p_\oplus = (n_\oplus)/(n_\oplus + n_\ominus)$ and $p_\ominus = (n_\ominus)/(n_\oplus + n_\ominus)$.

Intuitively, entropy characterizes the variety in a set of examples. The maximum value for entropy is one, which corresponds to a collection that has an equal number of positive and negative examples. The minimum value of entropy is zero, which corresponds to a collection with only positive or only negative examples. We can now define the quality of an attribute A by the reduction in entropy on partitioning the examples using A. This measure, called the *information gain* [16], is defined as follows.

Definition 7: The *information gain* of an attribute A with respect to a set of samples E is calculated as follows:

$$\text{Gain}(E, A) = \text{Entropy}(E) - (|E_0|/|E|) \cdot \text{Entropy}(E_0) - (|E_1|/|E|) \cdot \text{Entropy}(E_1)$$

where E_0 and E_1 are the subsets of examples having the values zero and one, respectively, for attribute A.

The BestAttribute (Examples, Attributes) procedure returns the attribute $A \in$ Attributes that has the highest Gain(Examples, A). Its complexity is $O(|\text{Examples}|\,|\text{Attributes}|)$, where $|X|$ is the number of elements in the set X.

Example 4: We illustrate the working of our algorithm with an example. Continuing with our previous example, we calculate the gains for the attributes at the top node of the decision tree.

$$\begin{aligned}
\text{Entropy}(E) &= -(2/4)\log_2(2/4) - (2/4)\log_2(2/4) = 1.00\\
\text{Gain}(E, v_1) &= 1 - (2/4) \cdot \text{Entropy}(E_{v_1=0}) - (2/4) \cdot \text{Entropy}(_{v_1=1}) = 0.00\\
\text{Gain}(E, v_2) &= 1 - (1/4) \cdot \text{Entropy}(E_{v_2=0}) - (3/4) \cdot \text{Entropy}(_{v_2=1}) = 0.31\\
\text{Gain}(E, v_3) &= 1 - (2/4) \cdot \text{Entropy}(E_{v_3=0}) - (2/4) \cdot \text{Entropy}(_{v_3=1}) = 0.00\\
\text{Gain}(E, v_4) &= 1 - (1/4) \cdot \text{Entropy}(E_{v_4=0}) - (3/4) \cdot \text{Entropy}(_{v_4=1}) = 0.31.
\end{aligned}$$

The DecTree algorithm will pick v_2 or v_4 to label the Root.

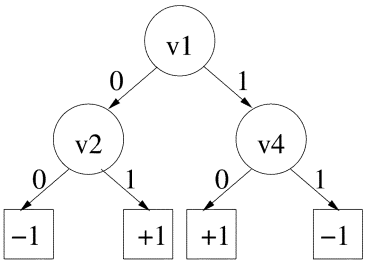


Fig. 4. Decision tree for Example 3.

VII. Directed Sampling of States

Sampling D_I and B_I does not have to be arbitrary. As we now show, it is possible to direct the search to samples that contain more information than others. Let $\delta(D_I, B_I)$ denote the minimal separating set for D_I and B_I. Finding $\delta(D_I, B_I)$ by explicitly computing D_I and B_I and separating them is computationally too expensive, because both the size of these sets and the optimal separation techniques are worst case exponential. We therefore look for samples S_{D_I} and S_{B_I} that are small enough to compute and separate, and on the other hand, maintain $\delta(S_{D_I}, S_{B_I}) = \delta(D_I, B_I)$. Finding these sets is what we refer to as directed sampling.

A. Algorithm Sample-and-Separate

We suggest an iterative algorithm for directed sampling. Let SepSet denote the current separating set. Initially, SepSet $= \emptyset$. Our algorithm iteratively adds or replaces elements in SepSet until it becomes a separating set for D_I and B_I. In each step $i > 0$, the algorithm finds samples that are not separable by SepSet that was computed in the previous iteration. Computing a new pair of deadend and bad states that are not separable by SepSet can be done by solving $\Phi(\text{SepSet})$, as

$$\Phi(\text{SepSet}) = \psi_f \wedge \phi'_f \wedge \bigwedge_{v_i \in \text{SepSet}} v_i = v'_i \qquad (3)$$

where ψ_f and ϕ_f are the formulas representing the deadend and bad states as defined in (1) and (2). The prime symbol over ϕ_f denotes the fact that we replace each variable $v \in \phi_f$ with a new variable v' (note that otherwise, by definition, the conjunction of ψ_f with ϕ_f is unsatisfiable). The right-most conjunct in the above formula guarantees that the new samples of deadend and bad states are not separable by the currently existing separating set.

Algorithm *Sample-and-Separate*, described in Fig. 6, uses formula (3) to compute the minimal separating set of D_I and B_I without explicitly computing or separating them. In each step i, it solves (3). If the formula is satisfiable, it derives from the solution the samples $d_i \in D_I$ and $b_i \in B_I$ (these are simply the assignments to the variables in ψ_f and ϕ'_f, respectively), which by definition are not separable by the current separating set SepSet. It then recomputes SepSet for the union of sets that were computed up to the current iteration. By repeating this process until no such samples exist, it guarantees that the resulting separating set separates D_I from B_I. Note that the size of SepSet can either increase or stay unchanged in each iteration.

The algorithm in Fig. 6 finds a single solution to $\Phi(\text{SepSet})$ and hence a single pair of states d_i and b_i. However, the size

$DecTree(Examples, Attributes)$

1. Create a *Root* node for the tree.
2. If all examples are classified the same, return *Root* with this classification.
3. Let $A = BestAttribute(Examples, Attributes)$. Label *Root* with attribute A.
4. For $i \in \{0, 1\}$, let $Examples_i$ be the subset of $Examples$ having value i for A.
5. For $i \in \{0, 1\}$, add an i branch to the *Root* pointing to subtree generated by $Dectree(Examples_i, Attributes - \{A\})$.
6. return *Root*.

Fig. 5. DTL algorithm.

$SepSet = \emptyset$;
$i = 0$;
repeat forever {
    If $\Phi(SepSet)$ is satisfiable, derive d_i and b_i from solution;
    else exit;
    $SepSet = \delta(\bigcup_{j=0}^{i}\{d_j\},\ \bigcup_{j=0}^{i}\{b_j\})$ //Separating all deadend from all bad states seen so far;
    $i = i + 1$; }

Fig. 6. Algorithm *Sample-and-Separate* implements directed sampling by iteratively searching for states that are not separable by the current separating set.

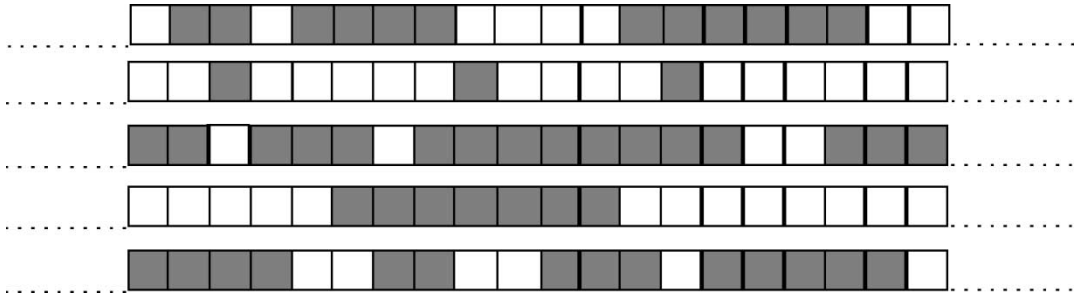


Fig. 7. Deadend states, where gray squares denote variables that separate the state from its bad state counterpart. Selecting samples with a smaller set of separating variables leads to faster convergence when looking for sets of variables that separate all pairs of samples.

of each sample can be larger. Larger samples may reduce the number of iterations but also require more time to derive and separate. The optimal number of new samples in each iteration depends on various factors, like the efficiency of the SAT solver, the separation technique, and the examined model. Our implementation lets the user control this process by adjusting two parameters: the number of samples generated in each iteration and the maximum number of iterations.

B. Bottleneck: Ordering of Samples

Although algorithm *Sample-and-Separate* enables us to completely separate the two sets of states in most cases, we found that in some large examples it can become a bottleneck. This problem is at least partially a consequence of the arbitrariness of selecting an optimal solution when there are multiple equally good possibilities. For example, in the biggest circuit we experimented with (which has 5000 latches), typically there are more than 1000 latches that can independently separate a specific sample of deadend and bad states (d_i, b_i). Our optimization engine selects one of these latches arbitrarily, say v_1. Then, a new pair of states (d_i', b_i') is sampled, and, again, more than 1000 options exist to separate the sets $(\{d_i, d_i'\}, \{b_i, b_i'\})$ with a single variable. If any one of these options was selected in the first iteration rather than v_1, it would have ruled out the sample (d_i', b_i'). Fig. 7 illuminates the general problem. It shows five deadend states, where each square represents a variable. Grayed squares denote variables that individually separate each state from its bad state counterpart (i.e., the bad state that was sampled in the same iteration of the algorithm).

The set $\{v_3, v_{10}\}$, for example, is sufficient for separating $S_{D_1} \ldots S_{D_5}$ from their counterpart bad states $S_{B_1} \ldots S_{B_5}$. Suppose that these five states constitute D_I, the full set of deadend states. Our goal is to find a small set of variables that eliminates all of them, while sampling the least number of states. Two possible scenarios are, reading from left to right:

- $(S_{D_3}, v_{20}); (S_{D_1}, v_{17}); (S_{D_4}, v_{12}); (S_{D_2}, v_3)$;
- $(S_{D_1}, v_{14}); (S_{D_4}, v_6)$;

where each pair represents a deadend state and the variable that is selected to separate it from its corresponding bad state. In the second option, the selection of v_{14} eliminates all states except S_{D_4}, hence the faster convergence. It is quite straightforward to see that giving priority to samples with a minimum number of separating variables improves the probability for fast convergence. For example, in the case of Fig. 7, choosing S_{D_2} first guarantees the elimination of S_{D_3} regardless of the separating variable that is selected. But the converse is not true. Choosing S_{D_3} first gives us only a probability of $(3/17)$ to guess a variable that eliminates S_{D_2}.

Selecting, in each iteration, the sample that has the minimum number of separating variables, corresponds to finding a satisfying assignment to $\Phi(\text{SepSet})$ of (3) that minimizes the number of pairs (v_i, v_i') that are evaluated differently. This problem can

Circuit	SMV	Samp, ILP			Samp, DTL			Eff, DTL			Eff, Opt		
	Time	Time	S	L	Time	S	L	Time	S	L	Time	S	L
*IU*30	0.7	0.1	0	1	0.1	0	1	**0.1**	0	1	0.14	0	1
*IU*35	0.6	0.1	0	1	0.1	0	1	**0.1**	0	1	**0.1**	0	1
*IU*40	1.2	6.3	3	4	0.9	5	6	0.6	2	3	**0.59**	1	3
*IU*45	37.5	6.1	3	4	1.1	5	6	**0.7**	2	3	0.85	1	3
*IU*50	23.3	19.7	13	14	9.8	13	14	24.0	4	17	**4.02**	2	9
*IU*55	-	-	-	-	2072	6	9	3.0	1	6	**0.37**	0	1
*IU*60	-	7.8	4	7	7.8	4	7	4.5	1	6	**0.48**	0	1
*IU*65	-	7.9	4	7	7.9	4	7	3.8	1	5	**0.51**	0	1
*IU*70	-	8.1	4	7	8.2	4	7	3.8	1	5	**0.47**	0	1
*IU*75	102.9	32.0	9	10	24.5	13	14	24.1	2	7	**0.32**	0	1
*IU*80	603.7	31.7	9	10	44.0	13	14	24.1	2	7	**0.37**	0	1
*IU*85	2832	33.1	9	10	44.6	13	14	25.2	2	7	**0.36**	0	1
*IU*90	-	33.0	9	10	44.6	13	14	25.4	2	7	**0.35**	0	1
Avg.	4123.2	783.5	>5.6	>7.1	174.3	7.2	8.8	10.7	1.5	5.8	0.7	0.3	1.9

Fig. 8. Model checking results for property 1. Timeout was set to 10 000 s and is included in the average results appearing in the last line.

Circuit	SMV	Samp, ILP			Samp, DTL			Eff, DTL			Depen. [6]			Eff, Opt		
	Time	Time	S	L	Time	S	L	Time	S	L	time	S	L	time	S	L
*IU*30	7.3	8.0	3	20	7.5	3	20	6.5	3	20	**1.9**	4	20	5.56	3	20
*IU*35	19.1	11.8	4	21	12.7	4	21	11.0	4	21	**10.4**	5	21	22.58	4	21
*IU*40	53.6	25.9	6	23	19.0	5	22	16.1	5	22	**13.3**	6	22	33.78	5	22
*IU*45	226.1	28.3	5	22	25.3	5	22	**22.1**	5	22	25	6	22	38.9	5	22
*IU*50	1754	160.4	13	32	85.1	10	27	15120	7	31	**32.8**	6	22	57.39	5	22
*IU*55	-	-	-	-	-	-	-	-	-	-	61.9	4	20	**58.94**	3	20
*IU*60	-	-	-	-	-	-	-	-	-	-	**65.5**	4	20	76.74	3	20
*IU*65	-	-	-	-	-	-	-	-	-	-	**67.5**	4	20	79.99	3	20
*IU*70	-	-	-	-	-	-	-	-	-	-	71.4	4	20	**69.39**	3	20
*IU*75	-	1080	21	38	586.7	16	33	130.5	5	26	**15.7**	5	21	22.59	4	21
*IU*80	-	1136	21	38	552.5	16	33	153.4	5	26	**21.1**	5	21	25.61	4	21
*IU*85	-	1162	21	38	581.2	16	33	167.7	5	26	**24.6**	5	21	27.5	4	21
*IU*90	-	965	20	37	583.3	16	33	167.1	5	26	**24.3**	5	21	27.96	4	21
Avg.	6312.3	3429.0	>12.7	>29.9	3265.6	>10.1	>27.1	4291.9	>4.9	>24.4	33.5	4.8	20.8	42.1	3.8	20.8

Fig. 9. Model checking results for property 2. Timeout was set to 10 000 s and is included in the average results appearing in the last line.

be easily formulated as an 0-1 ILP problem, or, as we will explain in the next section, as a pseudo-Boolean constraint (PBC) problem [12].

Due to the better ordering of samples, in some of the big examples we witnessed a decrease of two orders of magnitude in the number of iterations that are required for convergence of the sampling algorithm.

VIII. Experimental Results

We implemented our framework inside NuSMV [19]. We use NuSMV as a front end, for parsing SMV files and for generating abstractions. However, for actual model checking, we use Cadence SMV [4], which implements techniques like cone-of-influence reduction, cut-points, etc. We implemented a variant of the ID3 [17] algorithm to generate decision trees. We use Chaff [20] as our SAT solver. Some modifications were made to Chaff to efficiently generate multiple state samples in a single run.

To solve our 0-1 integer linear programs, we experimented with both the mixed ILP tool LP-Solve [21] and the PBC solver PBS [12]. Every 0-1 ILP problem can be modeled as a PBC problem. A PBC is a linear constraint over Boolean variables, e.g., $2b_1 + 3b_2 \leq C$ where C is an integer. Solving a 0-1 ILP minimality problem with PBC can be done by gradually decreasing a constant on the right-hand side of the objective function. While PBC can be flattened to propositional formulas, such an expansion is exponential. PBS, rather than performing this expansion, is built in the spirit of a standard Davis–Putnam SAT solver with special Boolean constraint propagation (BCP) rules for satisfying PBCs. For example, in the example above, if a partial assignment is $b_1 = 1$ and $C = 3$, it deduces that b_2 must be equal to zero. PBS accepts as input a CNF formula and a list of PBCs, one of which can be defined as an objective function. The formulation of our ILP problems, as described in Figs. 2 and 3, require solving constraints of the form $\sum v_i \geq 1$ or $v_i - v_j \geq 0$, which can be easily transformed to standard CNF clauses. PBS is therefore particularly suitable for these kinds of problems. We tuned PBS for our instances by forcing it to split first on variables in the objective function, and try first the value 0 for these variables since it is a minimality problem. This strategy turned out to be far superior to standard dynamic orderings.

Our experiments were performed on the "IU" family of circuits, which are various abstractions of an interface control circuit from SUN's Pico-Java processor, that we got from Synopsys. We also experimented with several other circuits from various other sources, as we report in Fig. 10. All experiments were performed on a 1.5-GHz Dual Athlon machine with 3-GB RAM and running Linux. No precomputed variable ordering files were used in the experiments.

		Depen. [6]			Eff. Opt		
Design	Length	Time	S	L	Time	S	L
M9	TRUE	10.2	2	38	**2.9**	1	38
M6	TRUE	44.3	4	50	**18.8**	4	50
M16	TRUE	1162	61	35	**44.7**	3	34
M17	TRUE	-	-	-	**733**	8	39
D6	20	**917**	46	89	1773	43	92
IUp1	TRUE	**3350**	13	19	-	9	41
Avg.		2580.6	>25.2	>46.2	2095.4	11.3	49.0

Fig. 10. Results for various large hardware designs, comparing our techniques with [6].

The results for the "IU" family are presented in Figs. 8 and 9. The two tables correspond to two different properties. We compared the following techniques: 1) "SMV": Cadence SMV; 2) "Samp, ILP": Random sampling, separation using LP-solve, 50 samples per refinement iteration; 3) "Samp, DTL": Random sampling, separation using DTL, 50 samples per refinement iteration; 4) "Eff, DTL": Directed sampling, separation using DTL; and 5) "Eff, Opt": Directed sampling with the optimization described in Section VII-B, minimizing the number of inputs rather than the number of latches as described in Section V-B, and solving the optimization problems with PBS. After a large set of experiments, which we do not list here, we concluded that the combination of these three optimizations is dominant over any subset of them, both for the results in this table and the tables in Figs. 9 and 10. For the results in Fig. 9, we added the results of [11] (they did not report their results for the circuits in the first table).

For each run, we measured the total running time in seconds ("Time"), the number of refinement steps ("S"), and the number of latches in the final abstraction ("L"). The original number of latches in each circuit in indicated in its name. A "—" indicates run-time longer than 10 000 s.

The experiments indicate that our technique expedites standard model checking in terms of execution time. As predicted, the number of iterations is generally reduced when either ILP or directed sampling is applied. These results are improved further with the "Opt" configuration as described above and are comparable to [11]. Our procedure, as expected, constructs either an equal or smaller abstract model compared to them (as indicated by the number of latches chosen). The method described in [11] also tries to reduce the number of latches by attempting to remove one latch at a time and checking whether the spurious counterexample can be simulated on the reduced abstract model. If the answer is no, they remove the latch. This technique does not guarantee a minimal abstract model and depends on the order in which the latches are examined. In the set of benchmarks we describe here, their method was quite successful though, as it achieved minimality or near minimality in most cases. In this set of examples, our minimality technique did not translate, in general, to faster run times compared to them because the model checking phase was not a bottleneck in either systems. We believe that the small advantage that they have in some cases is related to the fact that unlike their implementation, we do not currently use an incremental SAT solver to find the failure state. We refer the reader to the Section I, where we explained in more detail the difference between the two methods and why it is hard to compare between them.

Comparing the various configurations of our tool, it is apparent that in most cases the reduction in the number of required latches translates to a reduction in the total execution time. There were cases (see, e.g., circuit *IU*50 in Fig. 9), however, in which smaller sets of separating variables resulted in longer execution time. Such "noise" in the experimental results is typical to OBDD based techniques.

We also tried another set of examples, as summarized in Fig. 10.[1] For this set of examples, we replaced Cadence-SMV with the model checker used by [11] to check the abstract models. This model checker is built on top of NuSMV 2 and has several optimizations that Cadence-SMV does not have, like a very efficient mechanism for early quantification, as described in [22] and [23] (for the smaller examples described in the first two tables, changing the model checker did not make any notable difference).

Here, we can see that our method performs better in four cases and worse in two. As explained in Section II, we attribute the smaller number of latches that they find in the last two cases to the arbitrariness of the counterexamples that are generated by the model checker.

IX. Conclusion

We have presented an automatic counterexample guided abstraction-refinement algorithm that uses SAT, ILP, pseudo-Boolean constraint solving, and techniques from machine learning. Our algorithm outperforms standard model checking, both in terms of execution time and memory requirements. Our refinement technique is very general and can be extended to a large variety of systems. For example, following our earlier publication of this method in [3], directed sampling was adopted to model checking of software with predicate abstraction, as reported in [24].

References

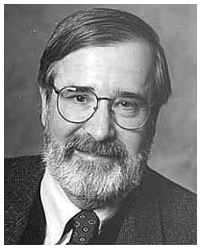
[1] J. Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. presented at *Theoretical and Practical Aspects of SPIN Model Checking: Proc. 5th and 6th Int. SPIN Workshops*. [Online]. Available: citeseer.nj.nec.com/rushby99integrated.html

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. 12th Int. Conf. Computer-Aided Verification (CAV'00)*, vol. 1855, E. Emerson and A. Sistla, Eds. New York, 2000.

[3] E. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," in *Proc. 14th Int. Conf. Computer-Aided Verification (CAV'02)*, vol. 2404, E. Brinksma and K. Larsen, Eds, Copenhagen, Denmark, July 2002, pp. 265–279.

[1]Unfortunately, we could not compare many other circuits because the model checker used by [11] is unstable.

[4] K. McMillan, *Cadence SMV*, CA: Cadence Berkeley Labs.
[5] Y. Lu, "Automatic abstraction in model checking," M.S. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 2000.
[6] S. Das and D. L. Dill, "Successive approximation of abstract transition relations," in *Proc. 16th Annu. IEEE Symp. Logic Comput. Sci., 2001*, Boston, MA, June 2001.
[7] R. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton Univ. Press, 1994.
[8] F. Balarin and A. Sangiovanni-Vinventelli, "An iterative approach to language containment," in *Proc. 5th Int. Conf. Computer-Aided Verification (CAV'94)*, vol. 697, C. Courcoubetis, Ed.. New York, 1993, pp. 29–40.
[9] J. Lind-Nielsen and H. Andersan, "Stepwise CTL model checking of state/event systems," in *Proc. 11th Int. Conf. Computer-Aided Verification (CAV'99)*, vol. 1633, N. Halbwachs and D. Peled, Eds. New York, 1999, pp. 316–327.
[10] 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 *Proc. Design Automation Conf. 2001 (DAC'01)*, 2001.
[11] 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 *Fourth Int. Conf. Formal Methods in Computer-Aided Design (FMCAD'02)*, Portland, OR, Nov. 2002.
[12] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "PBS: A backtrack-search psuedo-Boolean solver and optimizer," in *Fifth Int. Symp. Theory Applicat. Satisfiability Testing (SAT)*, 2002.
[13] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *ACM Trans. Prog. Lang. Syst.*, vol. 16, 1994, pp. 1512–1542.
[14] R. Cleaveland, P. Iyer, and D. Yankelevich, "Optimality in abstractions of model checking," in *Static Anal. Symp.*, 1995, pp. 51–63.
[15] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Workshop Tools Algorithms Construction Analy. Syst. (TACAS'99)*. New York, 1999.
[16] T. M. Mitchell, *Machine Learning*. New York: WCB/McGraw-Hill, 1997.
[17] J. Quinlan, Induction of decision trees, in *Machine Learning*, 1986.
[18] ——, *Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1993.
[19] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *Int. J. Software Tools Technology Transfer (STTT)*, vol. 2, no. 4, pp. 410–425, 2000.
[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. Design Automation Conf. 2001 (DAC'01)*, 2001.
[21] M. Berkelaar, "LPSolve, Version 2.0," Eindhoven Univ. Tech., Eindhoven, The Netherlands.
[22] P. Chauhan, E. M. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith, and D. Wang, "Non-linear quantification scheduling for efficient image computation," in *Proc. ICCAD 2001*, 2001, pp. 293–298.
[23] P. Chauhan, E. M. Clarke, S. Jha, J. Kukula, H. Veith, and D. Wang, "Using combinatorial optimization algorithms for efficient image computation," in *Proc. Correct Hardware Design and Verification Methods (CHARME)*, 2001, pp. 293–309.
[24] S. Chaki, E. Clarke, A. Groce, and O. Strichman, "Predicate abstraction with minimum predicates," in *Proc. 12th Conf. Correct Hardware Design and Verification Methods (CHARME)*, vol. 2860, D. Geist and E. Tronci, Eds., L'Aquila, Italy, Oct. 2003, pp. 19–34.



Edmund M. Clarke (M'96) received the B.A. degree in mathematics from the University of Virginia, Charlottesville, VA, in 1967, the M.A. degree in mathematics from Duke University, Durham, NC, in 1968, and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, in 1976.

After receiving his degrees, he taught in the Department of Computer Science, Duke University, for two years. In 1978, he moved to Harvard University, Cambridge, MA, where he was an Assistant Professor of Computer Science in the Division of Applied Sciences. He left Harvard in 1982 to join the faculty in the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA. He was appointed a Full Professor in 1989. In 1995, he became the first recipient of the FORE Systems Professorship, an endowed chair in the School of Computer Science. His research interests include software and hardware verification and automatic theorem proving. In 1981, he and his Ph.D. student A. Emerson first proposed the use of model checking as a verification technique for finite-state concurrent systems. His research group pioneered the use of model checking for hardware verification. Symbolic model checking using BDDs was also developed by his group.

Dr. Clarke has served on the editorial boards of *Distributed Computing and Logic* and *Computation* and is currently on the editorial board of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He is Editor-in-Chief of *Formal Methods in Systems Design*. He is on the steering committees of two international conferences, Logic in Computer Science and Computer-Aided Verification. He was a corecipient (along with R. Bryant, A. Emerson, and K. McMillan) of the ACM Kanellakis Award in 1999 for the development of symbolic model checking. For this work, he also received a Technical Excellence Award from the Semiconductor Research Corporation in 1995 and an Allen Newell Award for Excellence in Research from the Carnegie Mellon Computer Science Department in 1999. He is a Fellow of the Association for Computing Machinery and a member of Sigma Xi and Phi Beta Kappa.

Anubhav Gupta received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi, India, in 1999. Since then, he has been working toward the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, under the supervision of Edmund Clarke.

His research interests include verification of hardware and software systems.

Ofer Strichman received the B.Sc. and M.Sc. degrees from the Technion University, Israel, in operations research and systems analysis. He received the Ph.D. degree from the Weizmann Institute, Israel, where he worked on various formal verification topics under the supervision of Amir Pnueli.

After two years as a post-doctoral researcher at Carnegie Mellon University, Pittsburgh, PA, he joined the Technion University as an Assistant Professor in 2003. His research interests include formal methods, decision procedures in first order logic, SAT procedures, and selected areas in operations research.