SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques*

Edmund Clarke[1], Anubhav Gupta[1], James Kukula[2], and Ofer Strichman[1]

[1] Computer Science, Carnegie Mellon University, Pittsburgh, PA
{emc,anubhav,ofers}@cs.cmu.edu
[2] Synopsys, Beaverton, OR
kukula@synopsys.com

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 can not be simulated on the original (concrete) machine. We check whether a counterexample is real or spurious with a SAT checker. We then use a combination of Integer Linear Programming (ILP) 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.

1 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 must 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

* This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), and the Naval Research Laboratory (NRL) under contract no. 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.

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.

There are many known techniques, some automatic and some manual, for generating the initial abstraction and for abstraction/refinement. The automatic techniques are more relevant to this paper, not only because our method is fully automatic, but also because of the clear practical advantage of automation. 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 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 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 can not be easily performed with a standard circuit simulator, because the abstract counter example 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 [5]. We combine *sampling* with Integer Linear Programming (ILP) and machine learning to handle this problem. Machine learning algorithms are successfully used in a wide range of problem domains like data mining and other problems where it is necessary to extract implicit information from a large database of samples[10]. These algorithms exploit ideas from a diverse set of disciplines, including information theory, statistics and complexity theory.

The closest work to the current one that we are aware of was described in [5]. 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). The advantage of our 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 sub-optimal solution. For some well defined cases the same procedure computes the optimal refinement. We, on the other hand, avoid the complexity by considering only samples of the states sets, which we compute explicitly. By doing so we also pay the price of optimality: this procedure yields a refinement step which is not necessarily optimal (i.e., we do not necessarily find the smallest number of invisible variables that should become visible in order to eliminate the counterexample). Yet we suggest a method for efficient sampling, which in most cases allows us to efficiently compute an optimal refinement.

The work of [7] should also be mentioned in this context, since it is very similar to [5], 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) include the localization reduction of Kurshan [8] and many others (see, for example [1,9]). 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 in [14]. They use 3-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 on the variable dependency graph to the currently visible variables.

The rest of the paper is organized as follows. In the next section we briefly give the technical background of abstraction and refinement in model checking. In section 3 we describe our counterexample guided abstraction/refinement framework. We elaborate in this section on how the counterexample is being checked and how we refine the abstraction. We also describe refinement as a learning problem. In sections 4 and 5 we elaborate on our separation techniques. These techniques are combined with the efficient sampling technique, which is described in section 6. We give experimental results in section 7, which proves the viability of our methods comparing to a state of the art model checker (Cadence SMV). We discuss conclusions and future work in section 8.

2 Abstraction in Model Checking

We start with a brief description of the use of abstraction in model checking (for more details refer to [6]). Consider a program with a set of variables $V = \{x_1, \ldots, x_n\}$, where each variable x_i ranges over a non-empty 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

1. S is the set of states.
2. $I \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is the 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. $\hat{S} = \{\hat{s} \mid \exists s.\ s \in S \wedge h(s) = \hat{s}\}$.
2. $\hat{I} = \{\hat{s} \mid \exists s.\ I(s) \wedge h(s) = \hat{s}\}$
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) \mid \exists s_1.\ \exists s_2.\ R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$

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'.

For simplicity, we restrict our discussion to model checking of $\mathbf{AG}p$ formulas, where p is a non-temporal 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[6], 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 $h'(s_1) \neq h'(s_2)$.*

3 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:

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.

3.1 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 sub-section 3.3.

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.* $s^U = (s(u_1)...s(u_p))$. Let $\mathcal{V} = \{v_1, \ldots, v_k\}$. The partitioning 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 $h(s) = s^{\mathcal{V}}$.

Given h, we need to compute the minimal abstraction. For an arbitrary system M and abstraction function h, it is often too expensive or impossible to construct the minimal abstraction $\hat{M}$[6]. 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.

3.2 Checking the Counterexample

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

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

According to section 3.1, $h(s_i)$ 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)[3]. 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 RHS) of the visible variables, 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, $\langle \hat{s}_1, \dots \hat{s}_f \rangle$ is the longest prefix of the counterexample that corresponds to a concrete path. Our implementation sequentially searches in the range $1..m$ for the highest value f such that ψ_f is satisfiable. For long counterexample traces, we also have an option of performing a binary search over this range, in which case the number of SAT instances we solve is bounded by $\log m$.

3.3 Refining the Abstraction

As before, let f denote the failure index. Let D denote the set of all states d_f such that there exists some $\langle d_1 ... d_f \rangle$ in ψ_f. We call D the set of *deadend* states. 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 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.

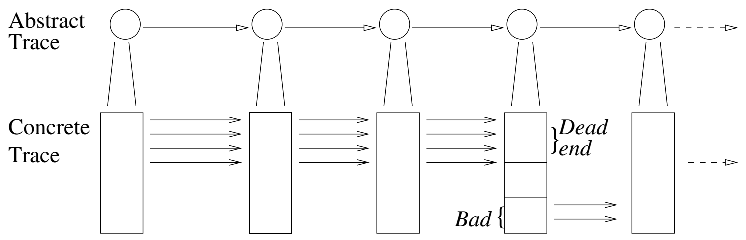


Fig. 1. A spurious counterexample corresponds to a concrete path that 'breaks' in the failing state. The failing state unites concrete 'deadend' and 'bad' states

The set of transitions ϕ_f is defined as follows:

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

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 figure 1).

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

3.4 Refinement by Separation and Learning

Let $S = \{s_1 ... s_m\}$ and $T = \{t_1 ... t_n\}$ be two sets of states (binary vectors) of size l, representing assignments to a set of variables W.

Definition 4. (The state separation problem) *Find a minimal set of variables* $U = \{u_1 ... u_k\}$, $U \subset W$, *such that for each pair of states* (s_i, t_j), $1 \le i \le m$, $1 \le j \le n$, *there exists a variable* $u_r \in U$ *such that* $s_i(u_r) \neq t_j(u_r)$.

Let D_I and B_I denote the restriction of D and B, respectively, to their invisible parts, *i.e.*, $D_I = \{s^I | s \in D\}$ and $B_I = \{s^I | s \in B\}$. 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 can be shown that computing the minimal separating set is NP-hard[5].

Lemma 1. *The new abstraction function h' separated D from B 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, $h'(d) = (d(u_1)...d(u_k))$ and $h'(b) = (b(u_1)...b(u_k))$, $u_i \in \mathcal{V}'$. Thus, $h'(d) \neq h'(b)$. $\square$

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 BDDs. However, even this is not possible for larger systems. Moreover, even if it were possible to generate D and B, it would still be computationally expensive to identify the separating variables.

Instead, we select *samples* from D and B and try to infer the separating variables for the entire sets from these samples. Of course, there is a tradeoff between the computational complexity of generating the samples, and the quality of the separating variables. Without a complete separation of D and B it can not be guaranteed that the counterexample will be eliminated. However, our algorithm is complete, because the counterexample will eventually be eliminated in subsequent refinement iterations. Our experience shows that state of the art SAT solvers like Chaff[11] can generate many samples in a short amount of time. The fact that D and B are large makes it relatively easy for SAT solvers to find satisfying assignments to equations 1 and 2 compared to typical SAT instances of similar size.

The idea of learning from samples has been studied extensively in the machine learning literature. A number of learning models and algorithms have been proposed. In the next two sections, we describe the techniques that we used to separate sets of samples of deadend and bad states, denoted by S_{D_I} and S_{B_I} respectively.

4 Separation as an Integer Linear Programming Problem

A formulation of the problem of separating S_{D_I} from S_{B_I} as an Integer Linear Programming (ILP) problem is depicted in Figure 2.

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

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

Fig. 2. State separation with integer linear programming

The value of each integer variable[1] $v_1 ... v_{|\mathcal{I}|}$ 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: $s_1 = (0,1,0,1), s_2 = (1,1,1,0)$ and $t_1 = (1,1,1,1), t_2 = (0,0,0,1)$. The corresponding ILP problem will be

$$\begin{array}{lll}
\text{Min } \sum_{i=1}^{4} v_i & & \\
\text{subject to:} & & \\
\quad v_1 + v_3 & \geq 1 & /\text{* Separating } s_1 \text{ from } t_1 */ \\
\quad v_2 & \geq 1 & /\text{* Separating } s_1 \text{ from } t_2 */ \\
\quad v_4 & \geq 1 & /\text{* Separating } s_2 \text{ from } t_1 */ \\
\quad v_1 + v_2 + v_3 + v_4 & \geq 1 & /\text{* Separating } s_2 \text{ from } t_2 */
\end{array}$$

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).

5 Separation Using Decision Tree Learning

The ILP-based separation algorithm outputs the minimal separating set. However, the algorithm has a high complexity and cannot handle a large number of variables or samples. In this section, we formulate the separation problem as a Decision Tree Learning(DTL) problem, which is polynomial both in the number of variables and the number of samples.

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 generates a decision tree that classifies them. An example is described by a set of attributes and the corresponding classification. 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 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}.
- The examples are S_{D_I} labeled $+1$, and S_{B_I} labeled -1.

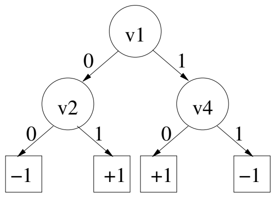
[1] Although the ILP problem is stated for integer variables, the constraints and objective function guarantees that their value will be either 0 or 1. Thus, they can be thought of as Boolean variables.

We generate a decision tree for this problem. The separating set that we output contains all the variables present at an internal nodes of the decision tree.

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. $\square$

Example 2. Going back to example 1, the corresponding DTL problem has 4 attributes (v_1, v_2, v_3, v_4) and as always, two classifications $(+1, -1)$. The set of examples is $E = \{((0,1,0,1),+1)$, $((1,1,1,0),+1)$, $((1,1,1,1),-1)$, $((0,0,0,1),-1)\}$. The following tree corresponds to the separating set (v_1, v_2, v_4).



A number of algorithms have been developed for learning decision trees, e.g. ID3[12], C4.5[13]. 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 Figure 3[10]. 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*, which is a commonly used notion in information theory. Given a set S containing $n_\oplus$ positive examples and $n_\ominus$ negative examples, the entropy of S is given by:

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

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

```
$DecTree(Examples, Attributes)$

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. 3. Decision tree learning algorithm

is 1, which corresponds to a collection that has an equal number of positive and negative examples. The minimum value of entropy is 0, 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* is defined as follows:

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

where E_0 and E_1 are the subsets of examples having the value 0 and 1, respectively, for attribute A. The $BestAttribute(Examples, Attributes)$ procedure returns the attribute $A \in Attributes$ that has the highest $Gain(Examples, A)$.

Example 3. 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.

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

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

6 Efficient 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 for 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 too computationally 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 efficient sampling.

We suggest an iterative algorithm for efficient sampling. Let $SepSet$ denote the current separating set. Initially, $SepSet = \emptyset$. In each step, the algorithm finds samples that are not separable by $SepSet$ that was computed in the previous iteration. Computing a new pair of dead-end and bad states that are not separable by $SepSet$, can be done by solving $\Phi(SepSet)$, as defined below:

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

where ψ_f and ϕ_f are the formulas representing the deadend and bad states as defined in equations 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

```
$SepSet = \emptyset$;
$i = 0$;
repeat forever {
    If $\Phi(SepSet)$ is satisfiable, derive $d_i$ and $b_i$ from solution; else
exit;
    $SepSet = \delta(\bigcup_{j=0}^{i}\{d_j\},\ \bigcup_{j=0}^{i}\{b_j\})$;
    $i = i + 1$; }
```

Fig. 4. Algorithm *Sample-and-Separate* implements efficient sampling by iteratively searching for states that are not separable by the current separating set

definition, the conjunction of ψ_f with ϕ_f is unsatisfiable). The right-most clause in the above formula guarantees that the new samples of deadend and bad states are not separable by the current separating set.

Algorithm *Sample-and-Separate*, described in Figure 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 finds samples $d_i \in D_I$ and $b_i \in B_I$ that are not separable by the current separating set *SepSet*. It then re-computes *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 Figure 6 finds a single solution to $\Phi(SepSet)$ and hence a single pair of states d_i and b_i. However, the size 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.

7 Experimental Results

We implemented our framework inside NuSMV[4]. We use NuSMV as a front-end, for parsing SMV files and for generating abstractions. However, for actual model checking, we use Cadence SMV, which implements techniques like cone-of-influence reduction, cut-points, etc. We implemented a variant of the ID3[12] algorithm to generate decision trees. We use a public domain LP solver[2] to solve our integer linear programs. We use Chaff[11] as our SAT solver. Some modifications were made to Chaff to efficiently generate multiple state samples in a single run. Our experiments were performed on the "IU" family of circuits, which are various abstractions of an interface control circuit from Synopsys. All experiments were performed on a 1.5GHz Dual Athlon machine with 3Gb RAM

Circuit	SMV		Sampling - ILP				Sampling - DTL				Eff. Samp. - DTL			
	Time	BDD	Time	BDD	S	L	Time	BDD	S	L	Time	BDD	S	L
*IU*30	0.7	116909	0.1	1731	0	1	0.1	1731	0	1	**0.1**	1731	0	1
*IU*35	0.6	149496	0.1	2357	0	1	0.1	2357	0	1	**0.1**	2357	0	1
*IU*40	1.2	225544	6.3	21249	3	4	0.9	18830	5	6	**0.6**	11028	2	3
*IU*45	37.5	2554520	6.1	17702	3	4	1.1	18847	5	6	**0.7**	10634	2	3
*IU*50	23.3	2094723	19.7	100647	13	14	**9.8**	90691	13	14	24.0	1274240	4	17
*IU*55	-	-	-	-	-	-	2072	51703825	6	9	**3.0**	64386	1	6
*IU*60	-	-	7.8	183811	4	7	7.8	183811	4	7	**4.5**	109393	1	6
*IU*65	-	-	7.9	192806	4	7	7.9	192806	4	7	**3.8**	47546	1	5
*IU*70	-	-	8.1	192806	4	7	8.2	192806	4	7	**3.8**	47546	1	5
*IU*75	102.9	7068752	32.0	142546	9	10	24.5	397620	13	14	**24.1**	550872	2	7
*IU*80	603.7	39989682	31.7	215404	9	10	44.0	341018	13	14	**24.1**	186662	2	7
*IU*85	2832	76232788	33.1	230979	9	10	44.6	443785	13	14	**25.2**	198359	2	7
*IU*90	-	-	33.0	230979	9	10	44.6	443785	13	14	**25.4**	198359	2	7

Fig. 5. Model checking results for property 1

and running Linux. No pre-computed variable ordering files were used in the experiments.
The results are presented in Figure 5 and Figure 6. The two tables correspond to two different properties. We compared the following techniques: 1) 'SMV': Cadence SMV, 2) 'Sampling-ILP': Sampling, separation using Integer Linear Programming, 50 samples per refinement iteration, 3) 'Sampling-DTL': Sampling, separation using Decision Tree Learning, 50 samples per refinement iteration, 4) 'Eff. Samp.-DTL': Efficient sampling, separation using Decision Tree Learning. For each run, we measured the total running time ('Time'), the maximum number of BDD nodes allocated ('BDD'), 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 '$-'$ symbol indicates that we ran out of memory. We could not solve Property 2 for circuits $IU55...IU70$ with any of the methods.

The experiments indicate that our technique expedites standard model checking, both in terms of execution time and required memory. As predicted, the number of iterations is generally reduced when either ILP or efficient sampling is applied. In most cases, this translates to a reduction in the total execution time. There were cases, however, when smaller sets of separating variables resulted in larger BDDs. Such 'noise' in the experimental results is typical of BDD based techniques.

8 Conclusions and Future Work

We have presented an automatic counterexample guided abstraction-refinement algorithm that uses SAT, ILP 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, in conjunction with predicate abstraction, we can apply our techniques to software model checking. There are several future research directions to our work. We are currently exploring

Circuit	SMV		Sampling - ILP				Sampling - DTL				Eff. Samp. - DTL			
	Time	BDD	Time	BDD	S	L	Time	BDD	S	L	Time	BDD	S	L
$IU30$	7.3	324268	8.0	113189	3	20	7.5	113189	3	20	**6.5**	113189	3	20
$IU35$	19.1	679224	11.8	186097	4	21	12.7	186097	4	21	**11.0**	186097	4	21
$IU40$	53.6	1100956	25.9	260299	6	23	19.0	207199	5	22	**16.1**	207199	5	22
$IU45$	226.1	6060256	28.3	411952	5	22	25.3	411952	5	22	**22.1**	411952	5	22
$IU50$	1754	25102082	160.4	2046981	13	32	**85.1**	605501	10	27	15120	3791826	7	31
$IU75$	-	-	1080	3716255	21	38	586.7	1178039	16	33	**130.5**	1050007	5	26
$IU80$	-	-	1136	3378860	21	38	552.5	1158076	16	33	**153.4**	1009030	5	26
$IU85$	-	-	1162	3493143	21	38	581.2	1272915	16	33	**167.7**	1079043	5	26
$IU90$	-	-	965	3712477	20	37	583.3	1271915	16	33	**167.1**	1079043	5	26

Fig. 6. Model checking results for property 2

criteria other than the size of the separating set for characterizing a good refinement. We also want to explore other machine learning techniques to solve the state separation problem.

References

1. F. Balarin and A. Sangiovanni-Vinventelli. An iterative approah to language containment. In C. Courcoubetis, editor, *Proc. 5th Intl. Conference on Computer Aided Verification (CAV'94)*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 29–40. Springer-Verlag, 1993. 267
2. M. Berkelaar. lpsolve, version 2.0. Eindhoven Univ. Tech., The Netherlands. 276
3. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS. Springer-Verlag, 1999. 270
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 1998. 276
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proc. 12th Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000. 266, 267, 271
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994. 268, 270
7. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA. 267
8. R. Kurshan. *Computer aided verification of coordinating processes.* Princeton University Press, 1994. 267
9. J. Lind-Nielsen and H. Andersan. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Proc. 11th Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 316–327. Springer-Verlag, 1999. 267
10. Tom M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997. 266, 274
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference 2001 (DAC'01)*, 2001. 272, 276

12. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1986. 274, 276
13. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993. 274
14. Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proc. Design Automation Conference 2001 (DAC'01)*, 2001. 267