Integrating Evolutionary Computation with Abstraction Refinement for Model Checking

Fei He, Xiaoyu Song, William N.N. Hung, Ming Gu, and Jiaguang Sun

Abstract—Model checking for large-scale systems is extremely difficult due to the state explosion problem. Creating useful abstractions for model checking task is a challenging problem, often involving many iterations of refinement. In this paper we consider techniques for model checking in the counterexample-guided abstraction refinement. The state separation problem is one popular approach in counterexample-guided abstraction refinement, and it poses the main hurdle during the refinement process. To achieve effective minimization of the separation set, we present a novel probabilistic learning approach based on the sample learning technique, evolutionary algorithm, and effective heuristics. We integrate it with the abstraction refinement framework in the VIS [1] model checker. We include experimental results on model checking to compare our new approach to recently published techniques. The benchmark results show that our approach has overall speedup of more than 56 percent against previous techniques. Our work is the first successful integration of evolutionary algorithm and abstraction refinement for model checking.

Index Terms—Formal models, verification.

1 Introduction

Formal property verification is assuming growing importance as digital designs grow in complexity and traditional validation techniques struggle to keep pace. Formal methods introduce mathematical rigor in their analysis of digital designs thereby guaranteeing exhaustive state space coverage.

Model checking is a popular formal verification technique. However, model checking for large-scale systems is extremely difficult due to the state explosion problem. Many techniques have been developed to ameliorate this problem. Abstraction is one of the most important techniques. The essence of abstraction is to eliminate the irrelevant information to reduce the system model. But coming up with a useful abstraction for the model checking task is often a challenge, especially for large systems. The abstracted system must be small enough to avoid state explosion during model checking, yet preserve the logical correctness of the property which was intended for the original large system. If the model checking found a counterexample on the abstracted system, we need to reproduce the counterexample on the original system. If the counterexample cannot be reproduced within the original system, it is a bogus counterexample. We need to create a different abstraction (a refinement) and conduct model checking again. Hence, finding a good abstraction is usually a tedious manual process. There are many types of abstraction for model checking large industrial designs [2], [3]. Nevertheless, most of the practical abstraction techniques involve expert users who are familiar with both the design under verification and model checking. Such high entrance bar is a bottleneck to widespread adoption of model checking.

As a step towards enabling automated abstraction for formal verification, counterexample-guided abstraction refinement (CEGAR) [4] has been developed to allow iterative abstraction with model checking. In CEGAR, the verification is performed in an abstract-check-refine fashion, and the refinement is guided by counterexamples. The counterexample contains the critical clues about the cause of the violation. If there exists a real path in the concrete model that simulates the counterexample, one can find a real bug, otherwise the counterexample is spurious and one has to refine the abstract model to eliminate such a spurious path.

1.1 Our Contribution

In this paper, we focus on CEGAR in verification of digital systems. We consider the strategy proposed in [5], where the abstraction is performed by making a set of latches or variables invisible. Consider an abstract counterexample $\hat{P} = \langle \mathrm{I},\ \mathrm{II},\ \mathrm{III},\ \mathrm{IV} \rangle$ shown in Fig. 1. It is spurious since there is no corresponding path in the concrete model. To eliminate the counterexample, the abstract model must be refined. That is, we need to distinguish the set of states $\{7, 8\}$ and $\{9\}$. As in [5], the set of states $\{7, 8, 9\}$ is defined as *failure* states. The sets of states $\{7, 8\}$ and $\{9\}$ are defined as *Deadend* and *Bad* sets, respectively. The *State Separation Problem* (SSP) is to effectively separate the dead-end set and bad set for the failure states.

- *F. He, M. Gu, and J. Sun are with Tsinghua National Laboratory for Information Science and Technology (TNList), and the School of Software, Tsinghua University, Beijing 100084, China and the Key Laboratory for Information System Security, Ministry of Education, Beijing, China. E-mail: {hefei, guming, sunjg}@tsinghua.edu.cn.*
- *X. Song is with the Department of Electrical & Computer Engineering, Portland State University, PO Box 751, Portland, OR 97207-0751. E-mail: song@ece.pdx.edu.*
- *W.N.N. Hung is with Synopsys, Inc., Verification Group, Mailstop: A-21, 700 East Middlefield Road, Mountain View, CA 94043. E-mail: William.Hung@synopsys.com.*

Manuscript received 14 Sept. 2008; revised 24 Apr. 2009; accepted 21 May 2009; published online 22 July 2009.
Recommended for acceptance by S. Shukla.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-09-0467.
Digital Object Identifier no. 10.1109/TC.2009.105.

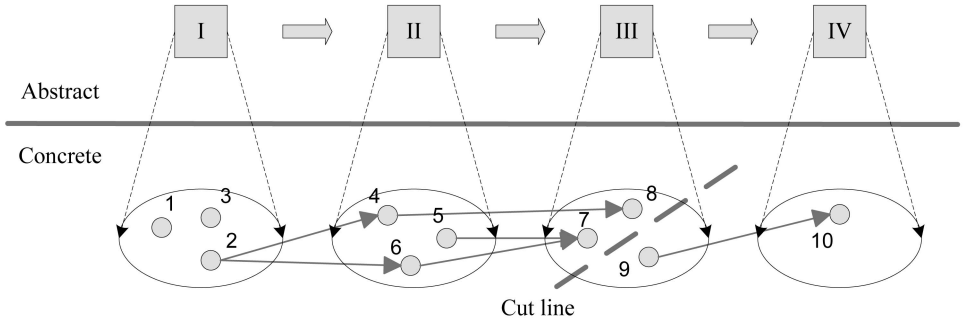


Fig. 1. A spurious counterexample.

In realistic systems, the size of failure states is usually very large. Moreover, since the state separation problem is embedded in the abstract-check-refine iteration, each time a spurious counterexample is found, a solution to the SSP needs to be provided. Thus, there is a strong demand for the effectiveness of SSP solvers in terms of time and memory.

We propose a novel probabilistic learning (PL) approach to SSP, which utilizes the sample learning technique, evolutionary algorithm (EA) and effective heuristics. We integrate our idea into the abstraction refinement framework of the VIS [1] model checker. Experimental results demonstrate the promising performance of our approach. Experiments on model checking indicate that our approach has overall speedup of 56 percent and 199 percent against recent published techniques in [5] and [6], respectively.

1.2 Prior Work

One of the pioneer works of CEGAR is Kurshan's localization reduction [7], where the abstract-check-refine paradigm was first proposed. In other approaches, such as [8], [9], the abstraction refinement schedule was determined using structural information of the model. A "goal set" of states is derived in [10] to refine successively, with respect to this goal set, the approximations made in the subformulas (of the CTL model checking), until the formula is verified or computational resources are exhausted.

There are many variations of the basic CEGAR [5], [6], [11], [12], [13], [14], [15], [16]. Most of them use a model checker (or symbolic analysis) and try to get rid of the spurious counterexample to reach a concrete counterexample or a proof of the property. Other recent methods on automatic abstraction [17], [18], [19], [20] employ the unsatisfiable core saved in the SAT solver, and the abstraction is based on the proofs provided by the SAT solver, but not on refuting the counterexamples.

In [15], the "hybridization effect" induced by the choice of projection (abstraction) is identified as the cause of the abstraction failure. A heuristic based on Hamming Distance is proposed to improve the choice of projections, hence refining the abstraction.

Predicate abstraction [21] is used in the refinement process of [13], [22], where a finite set of predicates is defined over the set of concrete state variables, such that each predicate corresponds to a fresh Boolean variable. The main disadvantage of predicate abstraction is that abstraction computation is very expensive.

The basic format of CEGAR framework has been used in commercial formal and semiformal verfication tools [14], [23], [24]. Automatic test-pattern generation and min-cut analysis are used during the counterexample analysis phase [23], [24]. Multiple Counterexample-based heuristic is also used to guide the refinement process [14].

In [11], an integer linear programming (ILP) model for the minimal state separation problem (MSSP) has been presented, and both an ILP solver and a decision tree learning (DTL) solver are employed for solving this problem. The general ILP solver attempts to enumerate the solution space to find the optimal solution for the state separation problem. However, since the minimal state separation problem is NP-hard, it is infeasible for the ILP solver to find the solution when the problem size is large. Note that we do not necessarily need the solution to be minimal. An approximate optimum may still be good enough for the refinement process; nevertheless, the resulting refined model may be slightly bigger.

In [5], an improved solver was proposed, which is based on decision tree learning. The DTL algorithm trains the decision tree based on input examples. It utilizes the well-trained decision tree to classify data. With some adjustments on the parameters, the DTL algorithm is used to solve the state separation problem, and the structure of its decision tree just gives a possible solution. Obviously, the DTL approach is an approximate method. Its solution precision relies on the number of input examples. If there are a sufficient number of examples, the solution could be guaranteed. However, if the input examples are too many, the time cost is extremely high. Thus, there is a trade-off between the solution precision and the solving cost. Furthermore, in coping with the large problem size, an efficient sampling technique has been applied to the DTL solver. Experimental results show that DTL solver with efficient sampling technique (for short, SDTL) outperforms the ordinary DTL solver [11].

In [6], an abstraction refinement method based on simultaneous analysis of all abstract counterexamples of the shortest length was presented. A multithread concretization technique was used to test all the counterexamples. To validate this method, a GRAB package has been implemented in VIS verification system [1]. That paper also presented experimental comparisons of various refinement algorithms including [5], [11], [12], and their impact on the overall performance of the CEGAR loop. The comparison [6] indicates the GRAB package is a very competitive

CEGAR implementation. Hence, we also compare our approach with the GRAB implementation in Section 5.

Some efficient heuristics for refinement variables selection were presented in [16]. To the best of our knowledge, it is the first study on the effectiveness of greedy heuristics for the state separation problem. Experimental results show the promising performance of these heuristics. This paper builds on [16] by extending these heuristics into a novel probabilistic learning approach.

An early version of our research was published in [25]. That paper presented the basic idea of using evolutionary algorithm in the state separation problem for abstraction refinement in model checking. The main drawback of that paper lies in its experimental parts. It showed only simple and trivial examples which were randomly generated. It lacked the linkage with a real model checker and did not provide any evidence to demonstrate its effectiveness in the entire abstraction refinement flow. In this paper, we chose VIS as our model checker platform and implemented the entire strategy. In addition, extended comparison tests have been performed. Now we can demonstrate the effectiveness of our method in the whole abstraction refinement flow.

1.3 Organization of this Paper

The remainder of the paper is organized as follows: In Section 2, we introduce some preliminaries. In Section 3, we formally define the problem. In Section 4, we present our probabilistic learning approach. The experimental results are reported in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

We use state transition systems to model systems. Given a nonempty set of atomic propositions AP, let $M = \langle S, S_0, R, L\rangle$ be a transition system where

- S is the set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation.
- $L : S \to 2^{AP}$ is the labeling function.

Let $V = \{v_1, v_2, \ldots v_{|V|}\}$ be the universal domain of system variables. We assume that the variables in V range over a finite set D. Atomic propositions are built from variables in V, constants in D and relation symbols, i.e., $v_1 - 2 > 0$. A valuation for V corresponds to a state in S.

As in [5], we think of V as two parts: the set of *visible* variables (denoted as V_S) and the set of *invisible* variables (denoted as V_N). Invisible variables are those that we will ignore when building the model. For example, consider a digital system with latches. The subset of latches that we are interested are considered as visible variables, while the remaining latches are regarded as invisible.

In the original (nonabstracted) model, all system variables are visible. The abstraction process is essentially selecting some variables to be invisible. Conversely, the refinement process is to convert some of the invisible variables to visible.

Let M be the original model. An abstraction function h is defined as a surjection $h : S \to \tilde{S}$. Given a concrete state $s \in S$, we denote $h(s)$ the abstract state to which it is mapped by h. Accordingly, given an abstract state $\tilde{s}$, we denote $h^{-1}(\tilde{s})$ the set of concrete states s such that $h(s) = \tilde{s}$.

Given the original model M and an abstraction function h, the abstract model $\tilde{M} = \langle \tilde{S}, \tilde{S}_0, \tilde{R}, \tilde{L}\rangle$ is defined as follows [4]:

- $\tilde{S} = \{\tilde{s} \mid \exists s \in S, h(s) = \tilde{s}\}$.
- $\tilde{S}_0 = \{\tilde{s} \mid \exists s \in S_0, h(s) = \tilde{s}\}$.
- $\tilde{R} = \{(\tilde{s_1}, \tilde{s_2}) \mid \exists s_1 \in S, s_2 \in S, R(s_1, s_2) \wedge h(s_1) = \tilde{s_1} \wedge h(s_2) = \tilde{s_2}\}$.
- $\tilde{L}(\tilde{s}) = \bigcup_{h(s)=\tilde{s}} L(s)$.

Note that in above definition we require the abstraction function to be conservative. Such a conservative translation may introduce additional behaviors into the abstract model. Consider the example shown in Fig. 1, after mapping the concrete states 7, 8, 9 to III, and 10 to IV, respectively, the additional transitions $7 \to 10, 8 \to 10$ are added implicitly to the abstract model.

A path in M is a sequence of states, $s_1, s_2, \ldots, s_m$, where $R(s_i, s_{i+1})$ holds for any $i < m$. Given an abstract path $\tilde{P} = \langle \tilde{s_1}, \tilde{s_2}, \ldots \tilde{s_m}\rangle$ in $\tilde{M}$ and a concrete path $P = \langle s_1, s_2, \ldots s_m\rangle$ in M, we define the simulation relation $\sim$ as

$$P \sim \tilde{P} \iff s_1 \in S_0 \text{ and } s_1 \in h^{-1}(\tilde{s_1}), s_2 \in h^{-1}(\tilde{s_2}), \ldots s_m \in h^{-1}(\tilde{s_m}). \quad (1)$$

In the counterexample-guided approach, if we find a counterexample $\tilde{P}$ in the abstract model, we check if there is a concrete path P in M such that $P \sim \tilde{P}$. If it is true, we have a real bug. Otherwise, the counterexample is spurious. In the case of the spurious counterexample, we need to compute the *failure index* i_F, i.e., the maximal index $i_F, i_F < m$, such that there exists a concrete path in M which simulates the i_F prefix of $\tilde{P}$. With the failure index, we define the *failure states* to be the set of concrete states $F = h^{-1}(\tilde{s_{i_F}})$ in M. Consider the example in Fig. 1, the failure index is III, and the failure states are 7, 8, and 9.

The failure states can be partitioned into three sets.

1. The set of dead-end states F_d: $s \in F_d$ if and only if
   - $s \in F$;
   - there exists a concrete path to s which simulates the i_F prefix of $\tilde{P}$.
2. The set of bad states F_b: $s \in F_b$ if and only if
   - $s \in F$;
   - there exists no concrete path to s which simulates the i_F prefix of $\tilde{P}$;
   - there exists a transition from s to some states in $h^{-1}(s_{i_F+1})$.
3. $F - F_d - F_b$.

Consider the example shown in Fig. 1, the states 7 and 8 are dead-end states, and the state 9 is a bad state.

3 State Separation Problem

A state is a group of valuations for all variables in V. We use $s_{\mapsto v}$ to denote the valuation of a state s for the variable v. Given a dead-end state $s \in F_d$ and a bad state $t \in F_b$, they cannot be distinguished in the abstract model, that is

$$\forall v \in V_S, \quad s_{\mapsto v} = t_{\mapsto v}.$$

We want s and t be separable in the refined model. Here goes the state separation problem.

Definition 1. *The SSP [5] is to find a subset Λ of the invisible variables in V_N such that*

$$\forall s \in F_d, \forall t \in F_b, \exists v \in \Lambda, s_{\mapsto v} \neq t_{\mapsto v}. \tag{2}$$

The set Λ is named as *separation set*. We usually want the separation set to be as minimal as possible so that the corresponding refined model is minimal. This problem is known as the *minimal state separation problem* (MSSP).

Consider the abstract counterexample $\hat{P} = \langle \text{I, II, III, IV} \rangle$ shown in Fig. 1. It is spurious since there is no corresponding path in the concrete model. For this instance, the failure states are 7, 8, and 9. To eliminate the counterexample, we need to make some variables visible to distinguish the sets of states $\{7, 8\}$ and $\{9\}$.

In realistic systems, the size of failure states is usually very large. Moreover, since the state separation problem is embedded in the abstract-check-refine iteration, each time a spurious counterexample is found, a solution to the SSP needs to be provided. Thus, there is a strong demand for the effectiveness of SSP solvers in terms of time and memory.

In the following, we first prove that the set covering problem is reducible to the MSSP. Then, we present a new mathematical model for MSSP.

Definition 2. *Given a pair of states $\langle s, t \rangle, s \in F_d, t \in F_b$, if there exists a variable v, such that $s_{\mapsto v} \neq t_{\mapsto v}$, we say that state pair $\langle s, t \rangle$ is* covered *by the variable v.*

With the concept of *covering*, the state separation problem can be redefined as follows:

Definition 3. *The SSP is to find a subset $\Lambda \subseteq V_N$ such that it covers all elements in $F_d \times F_b$. The MSSP is the optimization version of SSP which requires the subset Λ to be minimal.*

The set covering problem [26] is a famous NP-hard problem. We prove the NP-hardness of MSSP by reducing the set covering problem to MSSP.

Definition 4. *Given a universe $\mathcal{U}$ and a family $\mathcal{S}$ of subsets of $\mathcal{U}$, the set covering problem is to find a minimal subfamily $\mathcal{C} \subseteq \mathcal{S}$ of sets whose union is $\mathcal{U}$.*

Proposition 1. *The set covering problem is reducible to MSSP.*

Proof. Given a universe $\mathcal{U}$ and a family $\mathcal{S}$ of subsets of $\mathcal{U}$, consider $\mathcal{U}$ as $F_d \times F_b$, and define a variable v for each subset $s \in \mathcal{S}$. The set of all variables corresponds to the family $\mathcal{S}$. According to Definition 4, the set covering problem is to find a minimal set Λ of variables v such that it covers all elements in $\mathcal{U}$. Obviously, it is an MSSP. □

A failure state is not covered by any visible variable, we need only consider its invisible part when given an MSSP. In the following, we limit our discussion on invisible variables.

Assume there are n invisible variables, we denote a failure state s as a vector of length n with $s[i] = s_{\mapsto v}$, where v is the ith invisible variable in V_N.

Suppose we are given an MSSP instance with n invisible variables and m state pairs. For simplicity, we use $p_j, 1 \leq j \leq m$, to denote a state pair in $F_d \times F_b$, i.e., $F_d \times F_b = \{p_1, p_2, \ldots, p_m\}$. We define the decision variables as follows:

$$x_i = \begin{cases} 1, & \text{if } v_i \in \Lambda, \\ 0, & \text{else.} \end{cases}$$

Assume $p_j = \langle s^{p_j}, t^{p_j} \rangle$, where

$$s^{p_j} = \langle s^{p_j}[1], s^{p_j}[2], \ldots, s^{p_j}[n] \rangle$$

and

$$t^{p_j} = \langle t^{p_j}[1], t^{p_j}[2], \ldots, t^{p_j}[n] \rangle.$$

According to (2), p_j must be covered by certain variable in the separation set, i.e.

$$\exists v_i \in \Lambda, s^{p_j}[i] \neq t^{p_j}[i].$$

It is equivalent to

$$\sum_{i=1}^{n} (s^{p_j}[i] \oplus t^{p_j}[i]) \cdot x_i \geq 1, \tag{3}$$

where $\oplus$ is the *exclusive or* operator, and x_i the decision variable of v_i.

Let $A = \{a_{ij}\}_{m \times n}$ be a coefficient matrix where

$$a_{ij} = s^{p_j}[i] \oplus t^{p_j}[i], \quad \text{for } 1 \leq i \leq n,\ 1 \leq j \leq m.$$

Obviously, a_{ij} equals 1 if and only if the state pair p_j is covered by the variable v_i. Then the MSSP can be formulated as

$$\begin{aligned} &\min \sum_{i=1}^{n} x_i, \text{ where} \\ &\sum_{i=1}^{n} a_{ij} x_i \geq 1, \quad j = 1, \ldots, m, \end{aligned} \tag{4}$$

$$x_i = \{0, 1\}, \quad i = 1, \ldots, n, \tag{5}$$

where (4) and (5) characterize the feasible solutions.

4 Our Approach

During verification, the solutions of SSP do not need to be exactly the minimum. Thus, it is possible to use some approximate method to solve this problem. In [5], a decision tree learning solver is proposed. In this paper, we present a novel learning approach based on the sample learning technique, evolutionary algorithm, and efficient heuristics. Experimental results show the better performance of our approach.

4.1 Sample Learning Technique

In practice, the number of failure states of SSP is very large. It is not easy to determine the separation set for large-scale systems. In [11], an idea of inferring the separation set by learning from some selected samples, instead of the entire set, was introduced. More formally, given a set of dead-end states F_d and a set of bad states F_b, they looked for samples $S_{F_d} \subseteq F_d$ and $S_{F_b} \subseteq F_b$, such that the minimal separation set computed on S_{F_d} and S_{F_b} equals to that computed on F_d and F_b.

We follow the basic idea in [11] and present a new implementation for this technique. The main procedure of our Sample Learning Approach (SLA) is shown in Algorithm 1. The method avoids the complexity of SSP by considering only samples of the set of state pairs. This algorithm is iterative. By adjusting the parameters *MAX_ITER* and *MAX_SAM*, we set the maximal number of iterations and the maximal number of samples picked in every iteration. A sample here is a pair of states $\langle s, t\rangle \in F_d \times F_b$. The algorithm picks *MAX_SAM* samples in every iteration, among which only those that are not covered by the present separation set (we call them efficient samples) are added into the set *SAMPLE*. The *REQ_SIZE* is a preassigned parameter. When there are enough efficient samples generated, the set of samples will be renewed, and then the separation set is computed.

Note that we use the covering concept to judge the validity of the given samples. The samples that are already covered by the present separation set will be directly discarded. Given an appropriate value to the *REQ_SIZE*, many samples will be discarded directly according to their coverage to the present separation set, and thus the number of SSP solver invocations will be greatly reduced.

Let $A_j = \langle a_{0j}, a_{1j}, \ldots, a_{nj}\rangle$ be the coefficient vector corresponding to p_j. According to (3), it is not difficult to determine the coverage of p_j to the present separation set Λ. It is equivalent to testing true value of the following formula:

$$\sum_{i=1}^{n} a_{ij} \cdot x_i \geq 1. \tag{6}$$

The way we check the validity of samples is different from that in [11] where a SAT-based method is used. Our procedure is based on testing of (6), which is easy to be performed and can always be accomplished in a constant time. Reversely, the validity checking procedure in [11] is SAT-based which may become slower when the size of separation set increases.

Algorithm 1. Outline of the sample learning algorithm

```
1: Λ := φ
2: SAMPLE := φ
3: NEWSAMPLE := φ
4: for i := 1 to MAX_ITER do
5:   for j := 1 to MAX_SAM do
6:     pick the next sample ⟨s,t⟩ from F_d × F_b
7:     if ⟨s,t⟩ cannot be covered by Λ then
8:       NEWSAMPLE := NEWSAMPLE ∪⟨s,t⟩
9:     end if
10:  end for
11:  if sizeof(NEWSAMPLE) ≥ REQ_SIZE then
12:    SAMPLE := SAMPLE ∪ NEWSAMPLE
13:    call solver to compute Λ based on SAMPLE
14:    NEWSAMPLE := φ
15:  end if
16: end for
```

4.2 Probabilistic Evolutionary Algorithm

EA [27] is a powerful search and optimization paradigm. It utilizes the principles of natural evolution and "survival of the fittest." Starting with a set of initial solutions, evolutionary algorithms explore the solution space through simulated evolution. In each generation, the solutions are evaluated by their fitness. The more suitable they are, the more chances they have to survive and be reproduced.

EA is a *population*-based algorithm, which manipulates multiple solutions at the same time. Each solution is usually encoded as a string of symbols or numbers, called a *chromosome*. The number of chromosomes in the population is a predetermined integer and is called *population size*. EA uses evolutionary operators to evolve the population of candidate solutions. The underlying operators are that of *crossover*, *mutation*, and *selection*. In the beginning of each generation, all chromosomes are updated by the crossover and mutation operators. And then a proportion of the existing population is selected to breed a new generation. This process is repeated until a solution with sufficient quality is found or a previously defined generation limit is reached. The basic procedure of EA is shown in Algorithm 2.

Algorithm 2. Evolutionary algorithm

```
1: Generate an initial population
2: while not (terminal condition) do
3:   Update the chromosomes by crossover and mutation
     operations
4: Evaluate the fitness of each chromosome
5: Select chromosomes to form a new population
6: end while
```

There are many studies on applying evolutionary algorithms to the set covering problem [28], [29], [30], [31]. The experimental results listed in the above literatures show good performance of applying EA to the set covering problem. However, we cannot apply EA directly to the SSP due to the huge number of failure states. SSP is essentially a special case of set covering problem, where the number of constraints is much more than the number of variables. By applying the sample learning technique, we avoid the complexity of such huge number of constraints. We use EA as the central solver embedded in the learning structure which is used to compute the separation set.

We reinforce the basic EA in a way such that problem-specific knowledge is incorporated. We observe the following properties in SSP, which derive the effective heuristics:

1. For a state pair, there may be multiple variables that can cover it.
2. If the variables in a separation set cover all state pairs in $F_d \times F_b$, the corresponding solution is already a feasible solution.

In order to get a feasible solution more quickly, an effective strategy is to assign larger probabilities to the variables which cover more state pairs. Denote $EV(v)$ the number of state pairs covered by variable v. Based on the statistical analysis on the sets of states F_d and F_b, the $EV(v)$ values for all variables can be evaluated easily in advance of the execution of our algorithm. More clearly, for each state pair $\langle s, t\rangle$ in $F_d \times F_b$, we increase $EV(v_i)$ by 1 when $s[i] \neq t[i], 1 \leq i \leq n$.

4.2.1 Probabilistic Initialization

We use a n-bit binary string as the chromosome structure where n is the number of invisible variables. A value of 1 for the ith bit implies that the variable v_i is selected into the separation set.

We generate *pop_size* chromosomes to initialize the population. To obtain a random chromosome, the involved method acts as in Algorithm 3.

Algorithm 3. Initialize a chromosome
1: randomly generate an integer e $(0 \leq e \leq n)$, and use it as the size of the separation set.
2: randomly select e variables into the separation set.
3: the probability of each variable to be selected is proportional to the number of state pairs it covers.

4.2.2 Probabilistic Mutation

Mutation operator acts on one chromosome and results in a new candidate solution. Let P_m be the probability of mutation. The number of chromosomes undergoing the mutation operation is most likely $P_m * pop\_size$.

We adopt the two-point mutation approach. For a traditional two-point mutation, it randomly selects two points r_1 and r_2 in the chromosome, and then replaces the value of every character between sites r_1 and r_2 with a random value (0 or 1).

In our probabilistic two-point mutation, the mutation sites r_1 and r_2 are selected similarly; however, the value of each character between sites r_1 and r_2 are heuristically replaced, as shown in Algorithm 4.

Algorithm 4. Mutate a chromosome
1: randomly generate an integer e $(0 \leq e < r_2 - r_1)$.
2: randomly select e genes between sites r_1 and r_2 into the separation set.
3: the probability of each gene between sites r_1 and r_2 to be chosen is proportional to the number of state pairs it covers.

4.2.3 Probabilistic Crossover

Crossover operator acts on two selected chromosomes (called *parents*) and results in one or two new candidate solutions (called *children*). Let P_c be the probability of crossover. The number of chromosomes undergoing the crossover operation is most likely $P_c * pop\_size$.

We adopt the uniform crossover operator, which is claimed to have more recombination potential to combine smaller building blocks into larger ones [32], [33]. The uniform crossover usually works by first generating a *crossover mask* and then creating a child solution. The child solution is created by inheriting bits from parents. The mask bits decide from which parent the corresponding bits of the child can inherit.

Let P_1 and P_2 be the two parent chromosomes whose encoding strings are $P_1[1], \ldots, P_1[n]$ and $P_2[1], \ldots, P_2[n]$, respectively. Let M be the mask string $M[1], \ldots, M[n]$. The child string C is created as

$$C[i] := \begin{cases} P_1[i], & \text{if } M[i] = 0; \\ P_2[i], & \text{if } M[i] = 1. \end{cases}$$

We follow the probabilistic crossover operator defined in [28]. Empirical studies show that this crossover operator is suitable for the set covering problem. Probabilistic crossover is derived from the standard uniform crossover. For the probabilistic crossover operator, the probability of a parent to be chosen for passing its variable to the offspring is proportional to its fitness value. Formally, given parents P_1 and P_2 the crossover mask M is generated as

$$M[i] = \begin{cases} 0, \text{ with probability } p = \dfrac{fitness(P_2)}{fitness(P_1) + fitness(P_2)}, \\ 1, \text{ with probability } 1 - p, \end{cases}$$

where the $fitness$ function returns the objective value for a solution.

4.2.4 Solution Improvement

When applying evolutionary operators to the chromosomes, the resulting solutions are no longer guaranteed to be feasible. We implemented two strategies to deal with infeasible solutions.

The first strategy is to apply penalty function to deteriorate the optimality of an infeasible solution by adding a penalty cost to its objective function. In our approach, after the penalty function is applied, the optimization model becomes

$$\text{Minimize} \sum_{i=1}^{n} x_i + \sum_{j=1}^{m} f\left(\sum_{i=1}^{n} a_{ij} x_j \geq 1\right),$$

where $f(\cdot)$ is the penalty function for unsatisfying the constraints (4). The penalty function has a strong influence on the performance of the whole algorithm. In our approach, we implement a simple and efficient penalty function as follows:

$$f(x) = \begin{cases} 0, & \text{if } x \text{ is true}, \\ \text{BIGVALUE}, & \text{otherwise}. \end{cases}$$

The second strategy is to apply a heuristic operator to transform the infeasible solution into feasible solution. We implemented the heuristic feasibility operator proposed in [28] with minor modifications. By applying this heuristic operator, not only can the infeasible solutions be transformed into feasible solutions, but also the feasible solutions can be improved by eliminating the redundant variables. The basic idea behind the operator is the greedy heuristic. Algorithm 5 gives the framework of the operator.

Algorithm 5. Heuristic feasibility operator
1: for each A_j, compute the number of variables that are in the separation set and can cover this row, i.e.,

$$n_j = \sum_{i=1}^{n} a_{ij} x_i, \quad \text{for} \quad 1 \leq j \leq m.$$

2: **while** $(\exists j \in [1, m], n_j = 0)$ **do**
3: find the best variable v^* which is not in the separation set and can cover maximal number of uncovered rows, i.e.,

$$\max_{i=1}^{n} \left\{ \sum_{j=1}^{m} (n_i = 0) \wedge (x_j = 0) \wedge (a_{ij} = 1) \right\}.$$

4: add v^* into the solution and renew n_j for each A_j.
5: eliminate the redundant variables, i.e., the variables satisfying:

$$\forall j \in [1, m], (a_{ij} = 1) \wedge (x_i = 1) \rightarrow n_j \geq 2.$$

6: **end while**

TABLE 1
Our Solver versus EA

Benchmark	PL			EA	
	time	|SepSet|	eff_ratio	time	|SepSet|
ran_k10_m150_n120	0.062	8	2.38E-02	0.063	10
ran_k20_m150_n120	0.203	8	4.37E-02	1.796	11
ran_k30_m150_n120	0.688	6	4.21E-02	5.953	11
ran_k40_m150_n120	0.453	7	4.25E-02	4.859	11
ran_k50_m150_n120	2.313	6	4.25E-02	65.281	11
ran_k20_m500_n300	0.875	11	8.87E-03	132.875	15
ran_k30_m500_n300	2.125	8	1.02E-02	198.390	14
ran_k40_m500_n300	2.594	8	1.05E-02	225.219	14
ran_k50_m500_n300	6.438	8	1.25E-02	872.532	14
ran_k60_m500_n300	6.391	8	1.23E-02	323.297	13
ran_k30_m150_n200	0.750	7	3.38E-02	16.875	12
ran_k30_m500_n1000	6.640	10	5.06E-03	1937.75	15
ran_k30_m3000_n4000	129.141	14	4.46E-04	N/P	N/P
ran_k30_m5000_n4000	203.516	14	2.71E-04	N/P	N/P
ran_k40_m200_n250	1.953	7	2.17E-02	34.141	12
ran_k40_m1000_n2000	37.187	11	2.16E-03	N/P	N/P
ran_k40_m2000_n5000	156.343	12	6.85E-04	N/P	N/P
ran_k40_m8000_n7000	913.172	14	1.95E-04	N/P	N/P
ran_k50_m200_n300	3.656	7	2.44E-02	42.953	13
ran_k50_m1000_n2000	88.500	10	2.31E-03	N/P	N/P
ran_k50_m2000_n5000	336.297	12	8.12E-04	N/P	N/P
ran_k50_m8000_n7000	1507.328	13	2.57E-04	N/P	N/P
ran_k60_m200_n300	1.875	7	2.10E-02	40.359	12
ran_k60_m1000_n2000	77.375	10	2.53E-03	N/P	N/P
ran_k60_m2000_n5000	332.672	11	9.39E-04	N/P	N/P

TABLE 2
Our Solver versus SDTL

Benchmark	λ^a	σ^b	PL		SDTL	
			time	|SepSet|	time	|SepSet|
ran_k10_m150_n120	18	500	0.141	7	15.954	10
ran_k20_m150_n120	32	500	0.422	7	30.594	20
ran_k30_m150_n120	31	500	0.594	6	35.891	28
ran_k40_m150_n120	31	500	0.656	7	51.813	31
ran_k50_m150_n120	31	500	2.953	6	53.312	38
ran_k20_m500_n300	54	500	0.515	7	1197.562	20
ran_k30_m500_n300	62	500	1.281	7	1836.203	30
ran_k40_m500_n300	63	500	1.454	7	2664.453	40
ran_k50_m500_n300	76	500	3.907	7	3476.797	46
ran_k60_m500_n300	74	500	2.89	7	4360.484	55
ran_k30_m150_n200	41	500	0.719	6	78.485	29
ran_k30_m500_n1000	102	500	1.985	8	4028.937	30
ran_k30_m3000_n4000	214	500	3.813	8	N/P	N/P
ran_k30_m5000_n4000	217	500	3.922	8	N/P	N/P
ran_k40_m200_n250	44	500	1.735	7	470.078	36
ran_k40_m1000_n2000	173	500	4.703	8	N/P	N/P
ran_k40_m2000_n5000	274	500	5.734	8	N/P	N/P
ran_k40_m8000_n7000	438	500	8.859	8	N/P	N/P
ran_k50_m200_n300	59	500	1.891	7	1026.672	43
ran_k50_m1000_n2000	185	500	8.672	7	N/P	N/P
ran_k50_m2000_n5000	325	500	12.687	8	N/P	N/P
ran_k50_m8000_n7000	576	500	23.515	9	N/P	N/P
ran_k60_m200_n300	51	500	1.125	6	1595.594	47
ran_k60_m1000_n2000	203	500	7.578	8	N/P	N/P
ran_k60_m2000_n5000	376	500	11.625	8	N/P	N/P

[a] λ: the number of samplings
[b] σ: the number of samples picked in each sampling

5 Experimental Results

We implemented a PL solver based on our approach. Our solver is written in C++ language, and computes the separation set. To validate our approach, we compared our solver to the basic evolutionary algorithm solver and the previous published sampling decision tree learning (SDTL) solver [5], [11]. The results are listed in Sections 5.1 and 5.2. We compared how fast and how efficient do these solvers compute the separation set when given a set of dead-end states and a set of bad states. The benchmarks in Sections 5.2 and 5.1 are created using a random generator.

Note CEGAR is an iterative framework, which involves the state separation problem in each of its iteration. We want to validate that our approach is not only valuable to the state separation problem, but also can be used to improve the whole model checking. To validate this idea, we implemented our algorithm and the SDTL algorithm in the VIS [1] verification system. We compared the performance of VIS model checking with our probabilistic learning algorithm, with the SDTL algorithm, and with the original GRAB [6] algorithm, respectively. The benchmarks in Section 5.3 came from the VIS distribution [1], [34].

All experiments were run on a PC with Intel T2300 (1.66 GHz) CPU and 512 M RAM.

5.1 Our Solver versus EA

This experiment compares the performance of our solver to the basic evolutionary algorithm. The efficiency of a solver is evaluated by its runtime, and the solution quality is evaluated by the size of its separation set.

The results are listed in Table 1. *Benchmark* is the name of the tested benchmark. The benchmark's name implies relevant parameters. For example, name "$ran\_k30\_m500\_n300$" indicates that the number of invisible variables is 30, the number of dead-end states is 500, and the number of bad states is 300, respectively. The $time$ column lists the runtime in seconds, and the |SepSet| column gives the size of the resulting separation set. To guarantee termination, we impose a limit of two hours on the running time. We denote by "N/P" in the *time* column the examples that could not be solved in this time limit.

Since there is no sampling technique applied, all state pairs are involved in the running of the basic EA. To make a fair comparison, we make the sampling in our solver to be performed successively, such that all state pairs in $F_d \times F_b$ can be sampled. In comparison with the two solvers, we observed that our solver can always find a better solution in shorter runtime. The EA solver quickly blows up as the problem size increases. Such phenomena show that the sample learning technique and probabilistic operators can greatly improve the performance of an evolutionary algorithm.

We also sum up the efficient samples encountered in all iterations, and compare it to the total number of samples $\|F_d \times F_b\|$, the ratios are listed in the *eff_ratio* column. When we consider the columns *eff_ratio*, we can observe that for all benchmarks, the number of efficient samples is much less than the total number of samples. In all cases, the ratios of efficient samples to all samples are in the magnitudes of 10^{-2} to 10^{-4}. Such phenomena showed the power of our sample learning technique. With this technique, after an initial separation set is obtained, most of the samples will be discarded directly because they are already covered by the current separation set.

5.2 Our Solver versus SDTL

This experiment compares the performance of our PL solver to the SDTL solver [5], [11].

All results are listed in Table 2. The λ and σ columns give the number of samplings, and the number of samples

TABLE 3
Benchmarks' Information

Benchmark	Model			Prop
	#inputs	#outputs	#latches	#invs
4-arbit	0(16)	0(13)	19	2
8-arbit	0(38)	0(31)	43	2
arbiter	0(13)	3(13)	16	1
bakery	3(3)	0(0)	16	1
bpb	9(9)	4(4)	36	1
bufferAlloc	6(6)	5(5)	27	1
coherence	6(38)	0(24)	25	5
cups	2(2)	1(1)	14	1
dcnew	0(15)	0(10)	12	1
eisenberg	3(3)	0(0)	18	3
gigamax	0(44)	0(40)	10	1
good_bakery	3(28)	0(11)	36	3
heap	3(12)	6(15)	22	1
huff	5(47)	0(61)	37	2
luckySevenONE	6(6)	0(0)	30	1
needham	15(83)	0(60)	54	3
palu	10(21)	4(8)	37	1
rether	2(2)	0(0)	43	1
simple8	8(45)	0(29)	15	1
vMiim	41(102)	28(56)	83	1

picked in each sampling, respectively. Note that the solution of SSP need not be exactly the minimum. Thus, it is possible for us not to enumerate all state pairs in $F_b \times F_b$. The product $\lambda \times \sigma$ gives the total number of samples in all sampling iterations. In this experiment, the value of σ is fixed to 500, while the value of λ is associated to the number of efficient samples found in the former experiment. Note that if no information on efficient samples is given, the value of λ can be set to a predefined value too.

The results in Table 2 are arranged into six groups. In the first two groups, we let the numbers of dead-end states and bad states be fixed, and let the number of invisible variables increase, we observed that all solvers' runtimes increase in most cases. In the last four groups, we fixed the number of invisible variables, and let the number of dead-end states and bad states increase. We observed that the SDTL solver quickly blows up, whereas our solver still works well. Even for the benchmarks that are solvable by both the solvers, the runtime of our solver are two to four orders of magnitude smaller than that of the SDTL solver. Regarding the separation set size, the separation set found by our solver is 76 percent smaller than that by the SDTL solver on average.

5.3 Model Checking Experiments

To validate the merit of our approach to the whole model checking, we implemented our PL algorithm and the SDTL algorithm in the VIS [1] verification system. We choose the GRAB package [6], which came with VIS as a built-in package, to be our base verification framework for comparison, because it is originally derived from CEGAR and is reported to be a better abstraction refinement

TABLE 4
Model Checking Experiments

Benchmark	Prop	Result	PL			SDTL			GRAB	Ratios	
			ref_iter	image	time	ref_iter	image	time		SDTL/PL	GRAB/PL
4-arbit	P1	Y	3	33	0.81	6	103	1.44	0.89	1.78	1.10
4-arbit	P2	Y	5	120	1.33	5	182	1.42	1.05	1.07	0.79
8-arbit	P1	Y	10	233	7.32	15	346	11.36	18.01	1.55	2.46
8-arbit	P2	Y	8	376	5.53	15	697	12	11.49	2.17	2.08
arbiter	P1	Y	6	54	0.55	4	34	0.49	0.27	0.89	0.49
bakery	P1	N	7	490	8.78	6	430	7.5	15.88	0.85	1.81
bpb	P1	N	5	97	47.65	6	116	36.18	25.81	0.76	0.54
bufferAlloc	P1	Y	6	59	7.45	14	111	14.59	13.45	1.96	1.81
coherence	P1	N	4	40	2.12	5	57	2.5	0.51	1.18	0.24
coherence	P2	Y	4	98	2.13	7	162	4.01	1.48	1.88	0.69
coherence	P3	Y	4	166	2.02	5	238	3.26	2.48	1.61	1.23
coherence	P4	Y	3	223	1.75	6	338	3.6	1.89	2.06	1.08
coherence	P5	N	3	258	2.04	6	409	3.36	0.56	1.65	0.27
cups	P1	N	1	19	0.92	1	19	0.94	2.08	1.02	2.26
dcnew	P1	N	4	45	0.56	5	53	0.86	0.29	1.54	0.52
eisenberg	P1	Y	5	350	16.17	4	262	10.42	25.66	0.64	1.59
eisenberg	P2	Y	4	648	12.21	4	549	13.78	26.05	1.13	2.13
eisenberg	P3	Y	6	1024	10.87	7	1035	22.09	36.69	2.03	3.38
gigamax	P1	Y	3	49	0.69	2	33	0.54	0.26	0.78	0.38
good_bakery	P1	Y	3	246	5.78	5	391	10.8	60.19	1.87	10.41
good_bakery	P2	Y	5	570	11.01	7	924	13.79	66.35	1.25	6.03
good_bakery	P3	Y	4	881	10.24	7	1469	14.54	61.23	1.42	5.98
heap	P1	Y	9	260	10.03	13	291	11.19	12.78	1.12	1.27
huff	P1	Y	9	89	5.44	11	114	5.94	2.54	1.09	0.47
huff	P2	Y	7	150	3.49	11	195	5.14	2.52	1.47	0.72
luckySevenONE	P1	N	3	436	8.11	3	436	8.1	78.62	1.00	9.69
needham	P1	Y	11	157	45.74	19	276	45.8	48.28	1.00	1.06
needham	P2	N	5	228	8.47	9	402	17.74	N/P	2.09	N/P
needham	P3	N	6	282	15.58	10	483	12.81	N/P	0.82	N/P
palu	P1	N	4	36	161.14	8	55	344.6	738.31	2.14	4.58
rether	P1	Y	13	361	13.35	20	806	30.9	30.38	2.31	2.28
simple8	P1	Y	6	92	1.23	7	80	1.2	1.62	0.98	1.32
vMiim	P1	Y	4	51	1.57	9	83	3.08	2.31	1.96	1.47
sum			180	8221	432.08	262	11179	675.97	1289.93	1.56	2.99

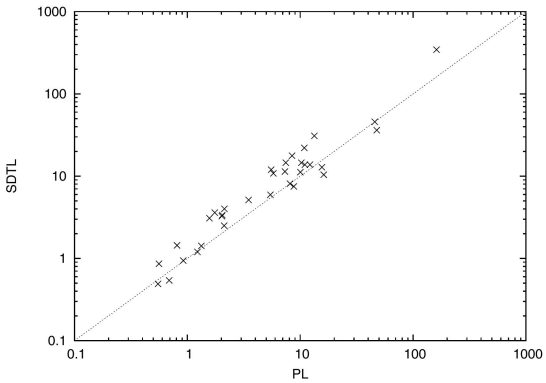


Fig. 2. Time speedup: SDTL versus PL.

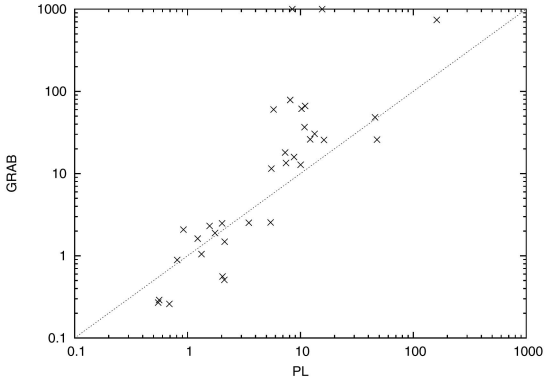


Fig. 3. Time speedup: GRAB versus PL.

technique. We compared the performances of VIS model checking (using GRAB method) with above three kinds of refinement algorithms enabled, respectively. For simplicity, we use the algorithm name to denote the corresponding run of model checking involving this algorithm, i.e., PL, SDTL, and GRAB.

All benchmarks tested in this experiment came from the VIS Verification Benchmarks [34]. Information about these benchmarks are listed in Table 3. Column *#inputs* lists the number of inputs, where the first integer gives the number of primary inputs, and the second gives the number of total inputs with all hierarchies. In the same way, column *#outputs* lists the number of outputs. Column *#latches* gives the number of latches in the model. Column *#invs* lists the number of invariants to be validated on the model.

The results are listed in Table 4, where *Prop* column lists the properties no., and *Result* column gives results for verifying the property on the model, where value "Y" means property passed and "N" means property failed. Note that all the three solvers report same result for each property on each model. The runtime (in seconds) for PL, SDTL, and GRAB is listed in corresponding *time* columns. In general, by looking at the *sum* row of Table 4, we can see that PL outperforms both SDTL and GRAB in runtime. Moreover, notice that GRAB blows up in BDD and times out for two properties on the Needham design, while PL and SDTL work well for all properties on all designs.

In Table 4, we also reported the refinement iterations and image (including postimage and preimage) computations involving in runs of model checking. Note that the abstraction refinement framework of GRAB is a bit different from ours, which involves multiple refinements within one iteration, we compare only SDTL to PL with these measures. By considering together the refinement iterations, image computations, and runtime, we can find that the refinement iterations have significant impact to others. With the same benchmark, more refinement iterations usually results in more image computations, hence longer runtime.

To further analyze the data in Table 4, we compared the runtime of SDTL and GRAB to that of PL, and then plotted them in Figs. 2 and 3, respectively. Among the 33 cases, there are 24 cases where our solver runs faster than SDTL, and 23 cases where our solver runs faster than GRAB. For the 10 cases where our solver lost to GRAB, most of them are simple which can be solved in 10 seconds. For most complex cases, our solver beats GRAB. Note that the last two columns in Table 4 give the runtime ratios of SDTL and GRAB to that of PL. Consider all benchmarks, our PL solver has total speedup of 1.56 (i.e., 56 percent) and 2.99 (i.e., 199 percent) against SDTL and GRAB, respectively.

6 Conclusion

In this paper, we investigated techniques for counter-example-guided abstraction refinement in model checking. The state separation problem is one popular approach in counterexample-guided abstraction refinement, and it poses the main hurdle during the refinement process. We presented a novel probabilistic learning approach to solve this problem, and integrated it with the abstraction refinement framework in the VIS model checker. Experimental results showed the efficiency and power of our approach. Model checking experiments on the VIS Verification Benchmarks indicate that our approach has overall 56 percent speedup against SDTL and 199 percent speedup against GRAB. Our work is the first successful integration of evolutionary computing and abstraction refinement for model checking.

Acknowledgments

This work was supported in part by the Chinese National 973 Plan under grant No. 2004CB719400, the National Science Foundation of China under grant Nos. 60553002, 60635020, 60903030 and 90718039.

References

[1] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa, "VIS: A System for Verification and Synthesis," *Proc. Eighth Int'l Conf. Computer-Aided Verification (CAV)*, R. Alur and T.A. Henzinger, eds., vol. 1102, pp. 428-432, 1996.

[2] W.N.N. Hung and N. Narasimhan, "Reference Model Based RTL Verification: An Integrated Approach," *Proc. IEEE Int'l High Level Design Validation and Test Workshop (HLDVT)*, pp. 9-13, Nov. 2004.

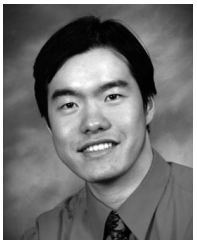
[3] R.H. Beers, R. Ghughal, and M.D. Aagaard, "Applications of Hierarchical Verification in Model Checking," *Proc. Formal Methods in Computer-Aided Design*, Nov. 2000.

[4] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counter-example-Guided Abstraction Refinement," *Proc. Computer-Aided Verification*, pp. 154-169, 2000.

[5] E.M. Clarke, A. Gupta, and O. Strichman, "SAT Based Counterexample-Guided Abstraction-Refinement," *IEEE Trans. Computer Aided Design,* vol. 23, no. 7, pp. 1113-1123, July 2004.
[6] C. Wang, B. Li, H. Jin, G.D. Hachtel, and F. Somenzi, "Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement," *IEEE Trans. Computer Aided Design,* vol. 25, no. 11, pp. 2297-2316, Nov. 2006.
[7] R.P. Kurshan, *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.
[8] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, "Tearing Based Abstraction for CTL Model Checking," *Proc. Int'l Conf. Computer-Aided Design,* pp. 76-81, Nov. 1996.
[9] J. Lind-Nielsen and H.R. Andersen, "Stepwise CTL Model Checking of State/Event Systems," *Proc. Computer-Aided Verification,* pp. 316-327, 1999.
[10] A. Pardo and G.D. Hachtel, "Incremental CTL Model Checking Using BDD Subsetting," *Proc. Design Automation Conf.,* pp. 457-462, 1998.
[11] E.M. Clarke, A. Gupta, J.H. Kukula, and O. Strichman, "SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques," *Proc. Computer-Aided Verification (CAV),* E. Brinksma and K.G. Larsen, eds., pp. 265-279, 2002.
[12] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT Based Conflict Analysis," *Proc. Formal Methods in Computer-Aided Design (FMCAD),* 2002.
[13] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," *Proc. Symp. Principles of Programming Languages,* pp. 58-70, 2002.
[14] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M.Y. Vardi, "Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* pp. 176-191, 2003.
[15] S.G. Govindaraju and D.L. Dill, "Counterexample-Guided Choice of Projections in Approximate Symbolic Model Checking," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* pp. 115-119, 2000.
[16] F. He, X. Song, M. Gu, and J. Sun, "Effective Heuristics for Counterexample-Guided Abstraction Refinement," *Proc. 17th ACM Great Lakes Symp. Very Large-Scale Integration (GLSVLSI '07),* H. Zhou and E. Macii, eds., pp. 393-398, 2007.
[17] K.L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* pp. 2-17, 2003.
[18] A. Gupta and O. Strichman, "Abstraction Refinement for Bounded Model Checking," *Proc. Computer-Aided Verification,* K. Etessami and S.K. Rajamani, eds., pp. 112-124, 2005.
[19] A. Gupta, M.K. Ganai, Z. Yang, and P. Ashar, "Iterative Abstraction Using SAT-Based BMC with Proof Analysis," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* pp. 416-423, 2003.
[20] C. Wang, H. Jin, G.D. Hachtel, and F. Somenzi, "Refining the SAT Decision Ordering for Bounded Model Checking," *Proc. Design Automation Conf. (DAC),* pp. 535-538, 2004.
[21] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," *Proc. Computer-Aided Verification,* pp. 72-83, 1997.
[22] H. Jain, D. Kroenig, N. Sharygina, and E.M. Clarke, "Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog," *Proc. Design Automation Conf.,* pp. 445-450, 2005.
[23] P.-H. Ho, T. Shiple, K. Harer, J.H. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart Simulation Using Collaborative Formal Simulation Engines," *Proc. Int'l Conf. Computer-Aided Design,* pp. 120-126, 2000.
[24] D. Wang, P.-H. Ho, J. Long, J.H. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines," *Proc. Design Automation Conf.,* pp. 35-40, 2001.
[25] F. He, X. Song, M. Gu, and J. Sun, "A Probabilistic Learning Approach for Counterexample Guided Abstraction Refinement," *Proc. Fourth Int'l Symp. Automated Technology for Verification and Analysis (ATVA '06),* S. Graf and W. Zhang, eds., pp. 29-50, Oct. 2006.
[26] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. MIT Press/McGraw-Hill, 2001.
[27] D. Dumitrescu, B. Lazzerini, L. Jain, and A. Dumitrescu, *Evolutionary Computation,* L. Jain, ed. CRC Press, 2000.
[28] J. Beasley and P. Chu, "A Genetic Algorithm for the Set Covering Problem," *European J. Operational Research,* vol. 94, pp. 392-404, 1996.
[29] S. Sen, "Minimal Cost Set Covering Using Probabilistic Methods," *Proc. 1993 ACM/SIGAPP Symp. Applied Computing,* pp. 157-164, 1993.
[30] U. Aickelin, "An Indirect Genetic Algorithm for Set Covering Problems," *J. Operational Research Soc.,* vol. 53, no. 10, pp. 1118-1126, 2002.
[31] E. Marchiori and A. Steenbeek, "An Evolutionary Algorithm for Large Scale Set Covering Problems with Application to Airline Crew Scheduling," *Proc. EvoWorkshops,* pp. 367-381, 2000.
[32] G. Syswerda, "Uniform Crossover in Genetic Algorithms," *Proc. Third Int'l Conf. Genetic Algorithms,* pp. 2-9, 1989.
[33] W.M. Spears and K.A. De Jong, "On the Virtues of Parameterized Uniform Crossover," *Proc. Fourth Int'l Conf. Genetic Algorithms,* R. Belew and L. Booker, eds., pp. 230-236, http://citeseer.ist.psu.edu/spears91virtues.html, 1991.
[34] VIS Verification Benchmarks, ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.0.tar.gz, 2008.

Fei He received the BS degree in computer science from the National University of Defence Technology, Changsha, China, in 2002. He received the MS and PhD degrees in computer science from Tsinghua University, Beijing, China, in 2004 and 2008, respectively. He is currently a lecturer in the School of Software at Tsinghua University. His current research interests include model checking, automata theory, and their applications in embedded systems.

Xiaoyu Song received the PhD degree from the University of Pisa, Italy, 1991. From 1992 to 1999, he was on the faculty at the University of Montreal, Canada. In 1998, he worked as a senior technical staff in Cadence, San Jose. In 1999, he joined the faculty at Portland State University. He is currently a professor in the Department of Electrical & Computer Engineering at Portland State University, Oregon. His current research interests include formal methods, design automation, embedded system design, and emerging technologies. He has been awarded as the Intel Faculty Fellow during 2000-2005. He served as an associate editor of the *IEEE Transactions on Circuits and Systems* and the *IEEE Transactions on VLSI Systems*.



William N.N. Hung received the BS and MS degrees in electrical and computer engineering from the University of Texas at Austin in 1994 and 1997, respectively. He received the PhD degree in electrical and computer engineering from Portland State University, Oregon, in 2002. He worked at Intel as a senior engineer from 1997 to 2004 in Hillsboro, Oregon. From 2004 to 2007, he worked at Synplicity as a senior staff engineer/director in Sunnyvale, California. Since November 2007, he has been working at Synopsys as a senior staff R&D engineer/senior R&D manager in Mountain View, California. His research interests include logic synthesis, physical design, formal methods, combinatorial optimization, nanotechnology, and quantum computing. He served as a session chair for the Design Automation Conference (DAC) and the IEEE World Congress on Computational Intelligence (WCCI), and as publications chair for the Formal Methods in Computer-Aided Design (FMCAD). He served as vice chair for the Quantum Computing Task Force of the Emergent Technologies Technical Committee for the IEEE Computational Intelligence Society. He also served in the Technical Program Committees of the Design Automation and Test in Europe (DATE), the IEEE Congress on Evolutionary Computation (CEC), and the IEEE International Computer Software and Applications Conference.

Ming Gu received the BS degree in computer science from the National University of Defence Technology, Changsha, China, in 1984, and the MS degree in computer science from the Chinese Academy of Science at Shengyang in 1986. Since 1993, she has been working as a lecturer/associate professor/researcher in Tsinghua University. She is also serving as the vice dean of the School of Software at Tsinghua University. Her research interests include formal methods, middleware technology, and distributed applications.

Jiaguang Sun received the BS degree in automation science from Tsinghua University in 1970. He is currently a professor in Tsinghua University. He is dedicated in teaching and R&D activities in computer graphics, computer-aided design, formal verification of software, software engineering, and system architecture. He is currently the director of the School of Information Science & Technology and the School of Software in Tsinghua University. He is also the director of the National Laboratory for Information Science & Technology. He has been a member of the Chinese Academy of Engineering since 1999.