Advance Access publication on October 11, 2007 doi:10.1093/comjnl/bxm085

Heuristic-Guided Abstraction Refinement

FEI HE[1,2,*], XIAOYU SONG[3], MING GU[2] AND JIAGUANG SUN[2]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[2]*School of Software, Tsinghua University, Beijing, China*
[3]*Department of ECE, Portland State University, Oregon, USA*
**Corresponding author: hef02@mails.tsinghua.edu.cn*

Model checking has been considered as a promising approach to establish the correctness of systems. Counterexample-guided abstraction refinement is a key strategy for model checking in verification of large-scale systems. State separation problem poses the main hurdle during the refinement. We present two fast heuristics to solve this problem. We prove the effectiveness of our heuristics by both theoretical analysis and experimental results. Experimental results show the promising performance of our approach.

Keywords: verification, model checking, abstraction, heuristics

Received 2 August 2006; revised 1 September 2007

1. INTRODUCTION

Model checking has been considered as a promising approach to establish the correctness of systems. While model checking has achieved significant success in verification of control-dominated systems, it still suffers from the state explosion problem when applied to large-scale systems. Abstraction is particularly useful for ameliorating state explosion.

One of the prior model abstraction and reduction works is Kurshan's localization reduction [1] in verifying the properties specified in omega-regular automata in which an abstract-check-refine paradigm was proposed. His method builds an abstract model, then checks the model, and if a counterexample is found, it refines the model and repeats the same iteration.

Recently, many abstraction strategies using abstract-check-refine paradigm have been proposed. They can be categorized into two approaches. The first approach [2–9] makes use of the counterexample, which is named as counterexample-guided abstraction refinement (CEGAR). In this method, the generated counterexample is used to test the original model. If the counterexample is a real path in the original model, then a real counterexample is found; otherwise the path is *spurious*, then the abstract model should be refined to eliminate such spurious paths. The second approach [6, 10–12] employs the unsatisfiable core saved in the SAT solver, and can rule out all counterexamples up to a given length.

In this paper, we focus on CEGAR in hardware verification. We follow the method proposed in [3], where the abstraction is performed by making a set of latches or variables invisible. In case the counterexample is spurious, we need to refine the abstract model. State separation problem poses the main hurdle during the refinement.

We propose the fast heuristic approaches to solve the state separation problem. We prove the effectiveness of our heuristics by both theoretical analysis and experimental results. Experimental results show the promising performance of our approach.

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 heuristic-guided approach. The experimental results are reported in Section 5. Finally, Section 6 concludes the paper.

2. PRELIMINARIES

Let $V = \{v_1, v_2, \ldots, v_{|V|}\}$ be the universal set of system variables. We assume the variables in V range over a finite set D. A valuation for V corresponds to a state in S. As in [3], 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 the latches in which we are interested is considered as visible variables, whereas the remaining latches are regarded as invisible.

In the original (non-abstracted) model, all system variables are visible. The abstraction process is essentially equivalent to

selecting and setting some of the visible variables as invisible. Oppositely, the refinement process is to make some of the invisible variables as visible.

We use transition systems to model systems. Given a set of atomic propositions AP, the original model M can be defined as a four-tuple $M = \langle S, S_0, R, L\rangle$, where S is the set of states, $S_0 \subseteq S$ the set of initial states, $R \subseteq S \times S$ the transition relation and $L : S \to 2^{AP}$ the labeling function. Similarly, the abstract model can be defined as $\tilde{M} = \langle \tilde{S}, \tilde{S}_0, \tilde{R}, \tilde{L}\rangle$.

Let h be a map function from S to $\tilde{S}$. Notice that we need our abstraction to be conservative, that is:

$$\tilde{R} = \{(\tilde{s_1}, \tilde{s_2}) | \exists s_1, s_2, R(s_1, s_2) \wedge \tilde{s_1} = h(s_1) \wedge \tilde{s_2} = h(s_2)\}.$$

Such conservative translation may introduce additional behaviors into the abstract model. Consider the example shown in Fig. 1, after mapping the concrete states 7–9 to III, and 10 to IV, respectively, the additional transition $7 \to 10$, $8 \to 10$ are added implicitly to the abstract model.

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: $P \sim \tilde{P}$ if and only if $s_1 \in S_0$ and $s_1 \in h^{-1}(\tilde{s}_1)$, $s_2 \in h^{-1}(\tilde{s}_2)$, $\ldots$, $s_m \in h^{-1}(\tilde{s}_m)$.

In the counterexample-guided approach, if we find a counterexample $\tilde{P}$ in the abstract model, then we check if there is a concrete path P in M such that $P \sim \tilde{P}$. If it is true, we find 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 group of concrete states $F = \{s | s \in h^{-1}(\tilde{s}_{i_F})\}$ in M. Consider the example in Fig. 1, the failure index is III, and the failure states are 7–9.

The failure states can be partitioned into three sets: set of dead-end states F_d, set of bad states F_b and set of remaining states $F - F_d - F_b$ [2]. Given a failure state s, it belongs to F_d if and only if there exists a concrete path to s that simulates the i_F prefix of $\tilde{P}$. Oppositely, given a failure state s, it belongs to F_b if and only if: (i) there exists no concrete path to s that simulates the i_F prefix of $\tilde{P}$; (ii) there exists a transition from s to some states in $h^{-1}(s_{i_F} + 1)$. Consider the example shown in Fig. 1, the state 7 is a dead-end state, and the state 9 is a bad state.

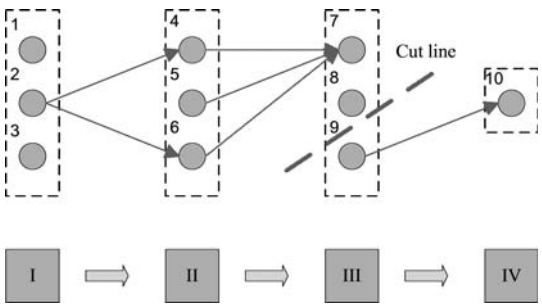

FIGURE 1. An spurious counterexample.

3. STATE SEPARATION PROBLEM

DEFINITION 3.1 *The state separation problem* (SSP) [3] *is to find a subset* Λ *of the invisible variables in* V_N *such that*

$$\forall s_i \in F_d, \forall t_j \in F_b, \exists v_r \in \Lambda, s_i(v_r) \neq t_j(v_r). \quad (1)$$

The set Λ is named as *separation set*. Usually, we 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).

Let $T(x)$ be the boolean function such that

$$T(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0. \end{cases}$$

Define the characteristic function of Λ of V_N as:

$$I_\Lambda(v) = \begin{cases} 1, & \text{if } v \in \Lambda, \\ 0, & \text{else.} \end{cases}$$

Assume $|V_N| = n$. We use the bit vector

$$I_\Lambda = \langle I_\Lambda(v_1), I_\Lambda(v_2), \ldots, I_\Lambda(v_n)\rangle$$

to denote a possible solution of the state separation problem. Given a possible solution, if it satisfies the constraint (1), then it is a feasible solution.

According to Definition 3.1, the state separation problem can be formulated as:

$$\max \Phi(I_\Lambda), \text{ where}$$

$$\Phi(I_\Lambda) = \sum_{\substack{s \in F_d \\ t \in F_b}} T\left(\sum_{\substack{1 \le i \le n \\ s(v_i) \neq t(v_i)}} I_\Lambda(v_i) \right). \quad (2)$$

Obviously, the maximal value of $\Phi(I_\Lambda)$ equals $|F_d| \times |F_b|$, and only when $\Phi(I_\Lambda)$ gets this maximal value, the solution I_Λ is feasible.

In [2], the MSSP has been proved to be NP-hard. In [3], an integer linear programming (ILP) model for the MSSP has been presented, and both an ILP solver and a decision-tree learning (DTL) solver are employed to solve 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 MSSP is NP-hard, it is very

time-consuming 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 [3], an improved solver was proposed, which is based on the DTL. The DTL algorithm trains the decision tree based on the 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. In [8], an alternative implementation of DTL for computing the separation set has been present. This paper also present a comprehensive set of experimental comparisons of various refinement algorithms, and their impact on the overall performance of the CEGAR loop.

4. HEURISTIC-GUIDED SEARCH

We consider heuristic-guided search in this section. By using the greedy heuristics, the search process can quickly be guided to the optimal solution in a more direct way than the DTL approach does.

4.1. Main framework of our algorithm

Algorithm 1 is the main framework of our approach. In our method, the separation set Λ is initialized to be empty. The function *decide_next_var*() determines the next variable to be added. Every time a variable is added, the set Λ will be tested for its satisfiability of the constraint (1). If the check succeeds, which means the present Λ is already a feasible solution to the state separation problem, the algorithm terminates; otherwise the loop continues again.

Algorithm 1 Main framework

```
Λ := φ
while NOT (Λ satisfy (1)) do
  v = decide_next_var();
  Λ := Λ ∪ {v};
  V_N := V_N − {v};
end while
return Λ
```

Function *decide_next_var*() is the key of our algorithm. We present two heuristics to implement it.

4.2. Heuristic 1

Considering the formula (2) of the state separation problem, a natural idea for the variable selection is to choose a variable which can increase the value of $\Phi(I_\Lambda)$ the most. However, it is too time-consuming to determine such a variable. Instead, we consider the following heuristic.

Heuristic 1. Choose a variable with the largest $EV(v)$, where $EV(v)$ is the number of states pairs $\langle s,t\rangle$ such that:

$$\forall s \in F_d, \forall t \in F_b, s(v) \neq t(v). \tag{3}$$

Obviously, based on the statistic analysis on the sets of states F_d and F_b, we can compute the $EV(v)$ values for all variables in advance before the execution of our algorithm. And then the function *decide_next_var*() can easily be implemented by selecting the variable with the greatest $EV(v)$ value from V_N.

In what follows, we give some important observations on Heuristic 1 in regard to the effectiveness of the method. First of all, we can easily observe that the Heuristic 1 does guide the search in the direction that increases the value of

$$\tilde{\Phi} = \sum_{\substack{1\le i\le n \\ I_\Lambda(v_i)=1}} EV(v_i).$$

According to (3), there is

$$EV(v_i) = \sum_{\substack{s\in F_d \\ t\in F_b}} T(s(v_i) \neq t(v_i)).$$

Therefore,

$$\begin{aligned}\tilde{\Phi} &= \sum_{\substack{1\le i\le n \\ I_\Lambda(v_i)=1}} \sum_{\substack{s\in F_d \\ t\in F_b}} T(s(v_i) \neq t(v_i)) \\ &= \sum_{\substack{s\in F_d \\ t\in F_b}} \sum_{\substack{1\le i\le n \\ I_\Lambda(v_i)=1}} T(s(v_i) \neq t(v_i)) \\ &= \sum_{\substack{s\in F_d \\ t\in F_b}} \sum_{\substack{1\le i\le n \\ s(v_i)\neq t(v_i)}} T(I_\Lambda(v_i) = 1) \\ &= \sum_{\substack{s\in F_d \\ t\in F_b}} \sum_{\substack{1\le i\le n \\ s(v_i)\neq t(v_i)}} I_\Lambda(v_i).\end{aligned}$$

Comparing $\tilde{\Phi}$ to the formula (2), we have $\tilde{\Phi} \geq \Phi$, and on the other hand we can deduce following proposition.

Proposition 1. *When $\tilde{\Phi}$ reaches its maximal value, Φ must also get its maximal value.*

Proof. By contridiction, assume Φ does not get its maximal value, then there must exist $s \in F_d$, $t \in F_b$ and $i \in [1, n]$, such that $v_i = 0$ and $s(v_i) \neq t(v_i)$. Then $\tilde{\Phi}$ does not get its maximal value either, because at least we can set $v_i = 1$ to increase $\tilde{\Phi}$ by 1. This is contradictory to the assumption. Thus, the proposition follows. □

From the above proposition, we can conclude the Heuristic 1 does guide the search to the feasible solutions of the state separation problem.

4.3. Heuristic 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 $s(v) \neq t(v)$, we say that the state pair $\langle s,t\rangle$ is *covered* by the variable v. With the covering definition, we observe following facts.

(i) For a state pair, there may be multiple variables that can cover it.
(ii) If the variables in a separation set can cover all of the state pairs in $F_d \times F_b$, then the corresponding solution is already a feasible solution.
(iii) Given a separation set, the uncovered state pairs are those that cannot be covered by any of the variables in it.

HEURISTIC 2. Choose the variable that can cover the most of uncovered state pairs $\langle s,t\rangle$ related to the current separation set.

To utilize the Heuristic 2, we need to implement a data structure for storing the uncovered state pairs related to the current separation set. Let Π be such a structure. It is initialized to be the Cartesian product of $F_d \times F_b$. The variables are evaluated by its coverage to the set Π. When choosing the next variable to be added into the separation set, the variable with the highest evaluation value will be selected. Then we renew the set Π by eliminating the elements that are covered by this variable, and then update the scores for remaining variables according to the renewed Π.

Comparing to Heuristic 1, Heuristic 2 is more aggressive. It is a dynamic decision heuristic, and can take search history into consideration. The experimental results demonstrate that Heuristic 2 is more likely to find a better solution. We also show the effectiveness of the heuristic 2 by theoretical analysis. Following proposition shows that the solution found by Heuristic 2 is very close to the global optimum.

PROPOSITION 2. *The approximation ratio of heuristic* 2 *is*

$$1 + \ln\left(\max_{v_i \in V_N} |v_i|\right).$$

Proof. Every variable in V_N covers a subset of elements in $F_d \times F_b$. Consider $F_d \times F_b$ as the universal set, and the variable in V_N as its subset, the SSP is try to find a collection of variables in V_N such to cover all elements in $F_d \times F_b$. Thus, the SSP is equivalent to the set cover problem. Heuristic 2 is essentially a kind of natural greedy algorithms. It has been proved that the natural greedy algorithm for the set cover problem lead to the approximation ratio of $1 + \ln(\max_{v_i} \in V_N|v_i|)$ [13]. Thus, the proposition follows. □

4.4. Computation of EV(v)

The computation of EV(v) for Heuristic 2 is implemented in the function *incre_satisy*(v*) as shown in Algorithm 2, where v* is the variable to be added into the separation set. The returned value can be 0 or 1, which represents whether the new-formed separation set satisfies equation (1) or not. The evaluation values will continually be updated during the repeated invocations of this function.

Assume all state pairs and their coverage statuses are stored in the structure *data*. The value of $EV(v)$ for each variable is initialized to be the total number of state pairs that can be covered by it (as same as the evaluation value for Heuristic 1). The coverage status of each state pair is initialized as false.

Note that for each state pair, there may be multiple variables can cover it. And according to the definition of Heuristic 2, each state pair can contribute to the evaluation value of one variable. Assume the variable v* is going to be added into the separation set, and there is an uncovered state pair r that can be covered by v*. First, we need to change the coverage status of r to be true. Second, after the status of r becomes covered, it cannot be considered any more by other variables in the computation of their evaluation values. So, we need to decrease all those $EV(v)$ values by 1.

Algorithm 2 Algorithm of $incre\_satisfy(v^*)$

```
result := 1;
for (i = 0; i < data.size; i++) do
  if data[i].status=true then
    continue; {already covered}
  else if data[i] can be covered by variable v* then
    data[i].status:=true; {just cover it}
    for (j = 0; j < num_var; j++) do
      if data[i] has ever been considered in the
      computation of EV(v_j) then
        EV(v_j)--;
      end if
    end for
    EV(v*) := 0; {set score to 0 to avoid being
    selected again}
  else
    result:=0;
  end if
end for
return result;
```

4.5. Efficient sampling technique

In practice, the number of failure states usually is very large. It is not possible for determining the separation set for large scale systems. In [3], an idea of inferring the separation set from some selected samples instead of the entire sets was introduced. Such an idea can also be incorporated in our approach.

As shown in Algorithm 3, we give the frame of the efficient sampling technique according to [3] with minor modification. By adjusting the parameters MAX_ITER and MAX_SAM, we set the number of iterations and the maximal number of samples generated in every iteration. A sample here is a pair of states $\langle s,t\rangle \in F_d \times F_b$.

Definition 4.1. *A sample is* efficient *if and only if it is not covered by the present separation set.*

The algorithm randomly generates MAX_SAM samples in every iteration, among which only the efficient samples can be added into the set SAMPLE. The separation set is then computed by the renewed set of samples.

Algorithm 3 Efficient sampling technique

```
Λ := φ
SAMPLE := φ
for i := 1 to MAX_ITER do
  for j := 1 to MAX_SAM do
    randomly pick ⟨s, t⟩ from F_d × F_b
    if ⟨s, t⟩ cannot be covered by Λ then
      SAMPLE := SAMPLE ∪⟨s, t⟩
    end if
  end for
  compute the Λ by some solver based on SAMPLE
end for
```

5. EXPERIMENTAL RESULTS

We have implemented two heuristic solvers: *heu1* and *heu2*, which are based on Heuristics 1 and 2, respectively. We use some randomly generated benchmarks to test our solvers. All experiments have been run on a PC with Intel® Celeron® 2.4 GHz CPU and 512 MB RAM.

The first experiment compares the CPU run-time and the separation set size between our solvers and the latest published DTL solver [3]. This experiment is focused on the algorithm performance comparison, and thus no sampling technique is applied in it. The results are shown in Table 1. *Benchmark* is the name of the tested benchmark. The benchmark's name implies some parameters. For example, the name '*ran_k*20_*m*500_*n*300' indicates that the number of invisible variables is 20, 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 $|\Lambda|$ column gives the size of the resulting separation set. We evaluate the efficiency of an algorithm by its runtime, and the solution quality by its

TABLE 1. Our solver versus DTL solver without efficient sampling technique

			Heuristic solver			
	DTL		heu1		heu2	
Benchmark	Time	$\|\Lambda\|$[a]	Time	$\|\Lambda\|$	Time	$\|\Lambda\|$
ran_k20_m150_n120	36.235	20	0.016	20	0.016	12
ran_k30_m150_n120	31.750	28	0.015	13	0.016	11
ran_k40_m150_n120	42.594	32	0.016	14	0.031	11
ran_k50_m150_n120	49.718	33	0.015	14	0.047	11
ran_k60_m150_n120	64.766	41	0.031	16	0.047	11
ran_k20_m500_n300	1220.843	20	0.078	19	0.140	16
ran_k30_m500_n300	1778.438	30	0.140	18	0.203	15
ran_k40_m500_n300	2828.907	40	0.156	21	0.312	14
ran_k50_m500_n300	3465.563	48	0.172	19	0.343	14
ran_k60_m500_n300	4918.453	58	0.203	18	0.375	14
ran_k30_m150_n200	84.390	29	0.015	18	0.031	13
ran_k30_m500_n1000	timeout		0.297	18	0.594	16
ran_k30_m3000_n4000	timeout		9.781	23	18.469	21
ran_k30_m5000_n4000	timeout		15.359	26	31.109	25
ran_k30_m10000_n50000	timeout		359.046	30	1045.266	27
ran_k40_m200_n250	356.500	36	0.047	17	0.093	12
ran_k40_m1000_n2000	timeout		2.281	23	4.250	18
ran_k40_m2000_n5000	timeout		12.750	28	21.047	21
ran_k40_m8000_n7000	timeout		58.828	26	119.125	23
ran_k50_m200_n300	880.625	42	0.062	21	0.125	13
ran_k50_m1000_n2000	timeout		2.672	21	5.094	17
ran_k50_m2000_n5000	timeout		15.593	24	25.937	20
ran_k50_m8000_n7000	timeout		95.375	29	147.094	22
ran_k50_m10000_n15000	timeout		194.406	30	392.843	24

[a]$|\Lambda|$ is the size of the resulting separation set

value of $|\Lambda|$. For the runtime of our solvers, the time spent on computing $EV(v)$ values is also included. In order to force termination, we imposed a limit of 3 h on the running time. We denote by 'timeout' in the *time* column the examples that could not be solved in this time limit.

The results in Table 1 are arranged into five groups. In the former two groups, we let the number of dead-end states and bad states be fixed, and let the number of invisible variables increase, we observed that all the solvers' run times increase slowly. In the latter three groups, we fixed the number of invisible variables, and let the number of dead-end states and bad states increase, we observed the DTL solver quickly blows up, whereas all our heuristic solvers still work well. Even for the benchmarks that are solvable by the DTL solver, the run times of our solvers are three to four orders of magnitude smaller than those of the DTL solver. Regarding the separation set size, our algorithm performs better than the DTL solver too. Consider our *heu2* solver, for example, the separation set found by it is smaller 60% than that by the DTL solver on average. Such phenomenons can be explained as follows. Since our solvers are based on the greedy strategy, they can find the feasible solution in a much more direct way than a machine-learning one, then it is understandable that our heuristic solvers ran much faster than the DTL solver. As the better solution quality obtained by our solvers, it indicates that the greedy heuristic is more suitable for this problem.

In comparison with our two heuristic solvers, we can observe that the solver *heu2* can always find the better solution. Although the runtime of *heu2* is longer, but it is approximately linear to that of *heu1*. In the following experiments, we use *heu2* only.

In the second experiment, we use *heu2* to test the efficient sampling technique. The experimental results are listed in Table 2. These results will be compared to those in Table 1 to show the effect of the sampling technique. To make the comparison as fair as possible, we let all state pairs in $F_d \times F_b$ be sampled. We sum up the efficient samples encountered in all iterations, and compare it the total number of samples (that is the size of the $F_d \times F_b$), the ratios are listed in the *eff_ratio* column.

Comparing these results to those listed in *heu2* column of Table 1, we observed that the runtime increases. This is easy to understand, since here the solver need to be invoked many times (see Algorithm 3). When we compare the Λ columns, we can find that the solutions' quality has greatly been improved (45% on average). This phenomenon shows that the effective sampling technique can greatly improve the performance of our solver. Additionally, when we consider the column *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 the efficient samples to all samples are in the magnitudes of 10^{-2} to 10^{-6}. This phenomenon indicates that it is not necessary to explore all the samples in $F_d \times F_b$ completely.

TABLE 2. Experiment with complete sampling technique using *heu2*

Benchmark	Time	$\|\Lambda\|$	eff_ratio
ran_k20_m150_n120	0.109	10	9.39×10^{-3}
ran_k30_m150_n120	0.125	7	1.58×10^{-2}
ran_k40_m150_n120	0.266	8	1.79×10^{-2}
ran_k50_m150_n120	0.359	7	2.18×10^{-2}
ran_k60_m150_n120	0.438	6	2.33×10^{-2}
ran_k20_m500_n300	0.672	15	1.51×10^{-3}
ran_k30_m500_n300	1.094	9	3.93×10^{-3}
ran_k40_m500_n300	2.187	10	5.03×10^{-3}
ran_k50_m500_n300	2.859	8	6.08×10^{-3}
ran_k60_m500_n300	3.609	8	6.88×10^{-3}
ran_k30_m150_n200	0.234	8	1.08×10^{-2}
ran_k30_m500_n1000	3.438	11	1.56×10^{-3}
ran_k30_m3000_n4000	86.172	16	1.60×10^{-4}
ran_k30_m5000_n4000	168.734	15	8.98×10^{-5}
ran_k30_m10000_n50000	4635.719	24	4.78×10^{-6}
ran_k40_m200_n250	0.75	7	1.01×10^{-2}
ran_k40_m1000_n2000	26.984	11	9.85×10^{-4}
ran_k40_m2000_n5000	131.141	14	3.56×10^{-4}
ran_k40_m8000_n7000	709.078	15	9.86×10^{-5}
ran_k50_m200_n300	1.171	8	1.03×10^{-2}
ran_k50_m1000_n2000	35.734	10	1.32×10^{-3}
ran_k50_m2000_n5000	169.75	13	4.81×10^{-4}
ran_k50_m8000_n7000	919.062	10	1.62×10^{-4}
ran_k50_m10000_n15000	2424.828	14	8.58×10^{-5}

In the third experiment, we compare the performance of our solver to DTL solver within the efficient sampling framework. The samples are picked randomly from the data file. Table 3 lists the experimental results. From Table 3, we observed that the DTL solver cannot get a solution in the time limit (3 h) for most of the benchmarks. Even for the benchmarks that DTL solver are available, the runtimes of our solver are about three orders of magnitudes less than those of DTL solver, and the separation sets obtained by our solver are also 70% smaller on average than those by DTL solver. Such results demonstrate the efficiency of our solver.

The fourth experiment is conducted on some real examples that come from the ITC' 99 benchemarks. The tool NuSMV [14] is selected as the back-end for performing model checking. The initial abstract models are obtained using the cone of influence technique. In the case of the spurious counterexamples, the models will be refined. We implemented a tool for generating the dead-end states and bad states from the failure paths. The early results we have gotten are listed in Table 4. Owing to implementation reasons, we only considered the first found counterexample. In Table 4, column $|V|$ lists the number of variables, column m the number of dead-end

TABLE 3. Our solver versus DTL solver with efficient sampling technique

			DTL		heu2	
Benchmark	λ[a]	σ[b]	Time	$\|\Lambda\|$	Time	$\|\Lambda\|$
ran_k30_m3000_n4000	240	50	141.39	30	0.468	8
ran_k30_m5000_n4000	400	50	timeout		0.719	8
ran_k30_m10000_n50000	1000	50	414.656	30	6.078	9
ran_k40_m1000_n2000	40	50	64.516	26	0.078	5
ran_k40_m2000_n5000	200	50	2476.86	31	0.547	7
ran_k40_m8000_n7000	1120	50	timeout		1.657	8
ran_k40_m30000_n80000	4800	50	timeout		21.985	9
ran_k50_m1000_n2000	40	50	78.625	23	0.094	5
ran_k50_m2000_n5000	200	50	1886.532	29	0.531	6
ran_k50_m8000_n7000	1120	50	timeout		3.047	8
ran_k50_m10000_n15000	3000	50	timeout		5.172	9
ran_k50_m30000_n80000	4800	50	timeout		22.579	9
ran_k60_m1000_n2000	40	50	59.406	25	0.125	5
ran_k60_m2000_n5000	200	50	1293.75	34	0.703	6
ran_k60_m8000_n10000	1600	50	timeout		5.11	7
ran_k60_m20000_n20000	4000	50	timeout		7.921	9
ran_k60_m30000_n120000	7200	50	timeout		42.969	9

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

TABLE 4. Experiments on ITC'99 benchmarks

				DTL		heu2	
Circuit	$\|V\|$[a]	m[b]	n[c]	Time	$\|\Lambda\|$	Time	$\|\Lambda\|$
B08	30	256	34	0.191	1	0.050	1
B11	38	64	130	0.240	1	0.110	1
B04	77	256	154	2.374	1	1.923	1

[a]$|V|$, the number of system variables
[b]m, the number of dead-end states
[c]n, the number of bad states

states and n the number of bad states. For comparison reason, only the runtimes for solving the state separation problem are accumulated. From Table 4, we observed that our solver outperforms the DTL solver in all cases. For each case, note the sizes of separation sets are small, such that the runtimes are both short.

6. CONCLUSION

We considered the SSP in this paper. A heuristic-guided approach was presented for solving this problem. An efficient sampling technique has been applied. Experimental results show the promising performance of our approach.

ACKNOWLEDGEMENTS

We gratefully acknowledge the work of the anonymous reviewers who provided helpful comments leading to significant improvements of the presentation, and Yu Xiao and Li Li for their helps on the experiments of ITC'99 benchmarks.

FUNDING

Chinese National 973 Plan (2004CB719406); National Natural Science Foundation of China (60553002, 60635020).

REFERENCES

[1] Kurshan, R.P. (1994) *Compter Aided Verification of coordinating Processes*. Princeton University Press, NJ, USA.
[2] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2000) Counterexample-guided abstraction refinement. In Emerson, E.A. and Sistla, A.P. (eds.), *CAV*, Chicago, IL, USA, July 15–19. Lecture Notes in Computer Science, **1855**, 154–169. Springer.
[3] Clarke, E., Gupta, A. and Strichman, O. (2004) SAT based counterexample-guided abstraction-refinement. *IEEE Trans. Comput. Aided Des.*, **23**, 1113–1123.
[4] Henzinger, T.A., Jhala, R., Majumdar, R. and Sutre, G. (2002) Lazy abstraction. *Symp. Principles of Programming Languages*, Portland, USA, January 16–18, pp. 58–70. ACM Press.
[5] Glusman, M., Kamhi, G., Mador-Haim, S., Fraer, R. and Vardi M.Y. (2003) Multiple-counterexample guided iterative abstraction refinement: an industrial evaluation. In Garavel, H. and Hatcliff, J. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, USA, April 7–11, pp. 176–191. Lecture Notes in Computer Science, **2619**. Springer.
[6] Gupta, A. and Strichman, O. (2005) Abstraction refinement for bounded model checking. In Etessami, K. and Rajamani, S.K. (eds.), *CAV*, Edinburgh, Scotland, UK, July 6–10. Lecture Notes in Computer Science, **3576**, pp. 112–124. Springer.
[7] Govindaraju, S.G. and Dill, D.L. (2000) Counterexample-guided choice of projections in approximate symbolic model checking. In Sentovich, E. (ed.), *ICCAD*, San Jose, CA, USA, , November 5–9, pp. 115–119. IEEE.
[8] Wang, C., Li, B., Jin, H., Hachtel, G.D. and Somenzi, F. (2003) Improving ariadne's bundle by following multiple threads in abstraction refinement. *Int. Conf. on Computer-Aided Design (ICCAD'03)*, Los Alamitos, CA, USA, November 9–13, pp. 408–415. IEEE Computer Society/ACM.
[9] Li, B., Wang, C. and Somenzi, F. (2003) A satisfiability-based approach to abstraction refinement in model checking. *Electronic Notes in Theoretical Computer Science*, **89**, pp. 608–622.
[10] McMillan, K.L. and Amla, N. (2003) Automatic abstraction without counterexamples. In Garavel, H. and Hatcliff, J. (eds.), *Tools and Algorithms for the construction and Analysis*

of Systems, Warsaw, Poland, USA, April 7–11, pp. 2–17. Lecture Notes in Computer Science, **2619**. Springer.

[11] Gupta, A., Ganai, M.K., Yang, Z. and Ashar, P. (2003) Iterative abstraction using SAT-based BMC with proof analysis. *Int. Conf. on Computer-Aided Design (ICCAD'03)*, Los Alamitos, CA, USA, November 9–13, pp. 416–423. IEEE Computer Society/ACM.

[12] Wang, C., Jin, H., Hachtel, G.D. and Somenzi, F. (2004) Refining the SAT decision ordering for bounded model checking. In Malik, S., Fix, L. and Kahng, A.B. (eds.), *DAC*, San Diego, CA, USA, June 7–11, pp. 535–538. ACM.

[13] Paschos, V.T. (1997) A survey of approximately optimal solutions to some covering and packing problems. *ACM Comput. Surv.*, **29**, 171–208.

[14] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A. (2002) NuSMV Version 2: an OpenSource Tool for Symbolic Model Checking. *Proc. Int. Conf. Computer-Aided Verification* (*CAV 2002*), Copenhagen, Denmark, July. LNCS, 2404. Springer.

[15] Garavel, H. and Hatcliff, J. (eds.) (2003) *Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, USA, April 7–11. Lecture Notes in Computer Science, **2619**. Springer.

[16] (2003) *International Conference on Computer-Aided Design* (*ICCAD'03*), Los Alamitos, CA, USA, November 9–13. IEEE Computer Society/ACM.