Detecting Spurious Counterexamples Efficiently in Abstract Model Checking ♠

Cong Tian and Zhenhua Duan ♠

ICTT and ISN Laboratory, Xidian University, Xi’an, 710071, P.R. China

Abstract

Abstraction is one of the most important strategies for dealing with the state space explosion problem in model checking. In the abstract model, the state space is largely reduced, however, a counterexample found in such a model may not be a real counterexample in the concrete model. Accordingly, the abstract model needs to be further refined. How to check whether or not a reported counterexample is spurious is a key problem in the abstraction-refinement loop. In this paper, a formal definition for spurious path is given. Based on it, efficient algorithms for detecting spurious counterexamples are proposed.

Key words: model checking, formal verification, abstraction, refinement, algorithm.

1. Introduction

Model checking is an important approach for the verification of hardware, software, multi-agent systems, communication protocols, embedded systems and so forth. The term model checking was coined by Clarke and Emerson [1], as well as Sifakis and Queille [2], independently. The earlier model checking algorithms explicitly enumerated the reachable states of the system in order to check the correctness of a given specification. This restricted the capacity of model checkers to systems with a few million states. Since the number of states can grow exponentially in the number of variables, early implementations were only able to handle

✩This research is supported by the NSFC Grant No. 61003078, 91018010, 61133001 and 60910004, 973 Program Grant No. 2010CB328102, SRFDP Grant No. 200807010012 and ISN Lab Grant No. ISN1102001.

*Corresponding author. E-mail address: zhenhua_duan@126.com
small designs and did not scale to examples with industrial complexity. To combat this, kinds of methods, such as abstraction, partial order reduction, OBDD, symmetry and bound technique are applied to model checking to reduce the state space for efficient verification. Thanks to these efforts, model checking has been one of the most successful verification approaches which is widely adopted in industrial community.

Among the techniques for reducing the state space, abstraction is certainly the most important one. Abstraction technique preserves all the behaviors of the concrete system but may introduce behaviors that are not present originally. Thus, if a property (i.e. a temporal logic formula) is satisfied in the abstract model, it will still be satisfied in the concrete model. However, if a property is unsatisfiable in the abstract model, it may still be satisfied in the concrete model, and none of the behaviors that violate the property in the abstract model can be reproduced in the concrete model. In this case, the counterexample is said to be spurious. Thus, when a spurious counterexample is found, the abstraction should be refined in order to eliminate the spurious behaviors. This process is repeated until either a real counterexample is found or the abstract model satisfies the property.

In the abstraction-refinement loop, how to check whether or not a reported counterexample is spurious is a key problem. In [3], algorithm SplitPath is presented for checking whether or not a counterexample is spurious, and a SAT solver is employed to implement it [4, 10]. In SplitPath, whether or not a counterexample is spurious can be checked by detecting the first failure state in the counterexample. If a failure state is found, the counterexample is spurious, otherwise, the counterexample is a real one. However, whether or not a state, say is, is a failure state relies on the prefix of the counterexample s0, s1, ..., si. This brings in a polynomial number of unwinding of the loop in an infinite counterexample [3, 15].

In this paper, based on a formal definition of failure states, spurious paths are re-analyzed, and a new approach for checking spurious counterexamples is proposed. Within this approach, whether or not a counterexample is spurious still depends on the existence of failure states in the counterexample. Instead of the prefix, to checking whether or not a state is a failure state is only up to is’s pre- and post- states in the counterexample. Based on this, for an infinite counterexample, the polynomial number of unwinding of the loop can be avoided. Further, the algorithm can be easily improved by detecting the heaviest failure state such that a number of model checking iterations can be saved in the whole abstract-refinement loop. In addition, the algorithm can be naturally parallelled.

The rest parts of the paper are organized as follows. The next section briefly presents the preliminaries in abstraction-refinement. In section 3, why spurious
counterexamples occur is analyzed intuitively and algorithm \textsc{SplitPath} is briefly presented. In section 4, a formal definition of spurious counterexamples is given with respect to the formal definition of failure states. Further, in section 5, efficient algorithms for checking whether or not a counterexample in the abstract model is spurious are presented. Finally, conclusions are drawn in section 6.

2. Abstraction and Refinement

There are many techniques for obtaining the abstract models \cite{6, 8, 12}. We follow the counterexample guided abstraction and refinement method proposed by Clarke, etc, where abstraction is performed by selecting a set of variables which are insensitive to the desired property to be invisible \cite{4}. We use $h : S \rightarrow \hat{S}$ to denote an abstract function, where $S$ is the set of all states in the original model, and $\hat{S}$ the set of all states in the abstract model. For clearance, $s, s_1, s_2, \ldots$ are usually used to denote the states in the original model, and $\hat{s}, \hat{s}_1, \hat{s}_2, \ldots$ indicate the states in the abstract model. Further, for a state $\hat{s}$ in the abstract model, $h^{-1}(\hat{s})$ is used to denote the set of origins of $\hat{s}$ in the original model.

The abstraction-refinement loop is depicted in Fig. 1. Initially, the abstract model $M'$ is obtained by the abstract function $h$. Then a model checker is employed to check whether or not the abstract model satisfies the desired property. If no errors are found, the model is correct. Otherwise, a counterexample is reported and rechecked by a checker which is used to check whether or not a counterexample is spurious. If the counterexample is not spurious, it will be a real counterexample that violates the system; otherwise, the counterexample is spurious, and a refining tool is used to refine the abstract model \cite{3, 4, 5, 7, 9, 13}. Subsequently, the refined abstract model is checked with the model checker again until either a real counterexample is found or the model is checked to be correct. In this paper, we concentrate on the how to check whether or not a counterexample is spurious.
3. Spurious Paths

To check a spurious counterexample efficiently, we first show why spurious paths occur intuitively with an example. Then we briefly present the basic idea of algorithm SplitPath which is used in [3, 15] for checking whether or not a counterexample is spurious.

3.1. Why Spurious Paths?

Abstraction technique preserves all the behaviors of the concrete system but may introduce behaviors that are not present originally. Therefore, when implementing the model checker with the abstract model, some reported counterexamples will not be real counterexamples that violate the desired property. This is intuitively illustrated by the traffic lights controller example [3].

Example 1. For the traffic light controller in Fig. 2(1), by making variable color invisible, an abstract model can be obtained as shown in Fig. 2(2). We want to prove □◊(state = stop) (any time, the state of the light will be stop sometimes in the future). By implementing model checking with the abstract model, a counterexample, \( s_1, s_2, s_3, s_2, \ldots \) will be reported. However, in the concrete model, such a behavior cannot be found. So, this is not a real counterexample. □

3.2. Detecting Spurious Counterexample with SplitPath

In [3], algorithms SplitPath is presented for checking whether or not a finite counterexample is spurious. In SplitPath, as illustrated in Fig. 3, initially, the set, \( M_0 \), of starting states falling into \( h^- (s_0) \),

\[
M_0 = I \cap h^- (s_0)
\]
is computed. Then for the image of the states in $I \cap h^{-}(\hat{s}_0)$, i.e. $R(I \cap h^{-}(\hat{s}_0))$, the set of states falling into $h^{-}(\hat{s}_1)$, 

$$M_1 = M_0 \cap h^{-}(\hat{s}_1) = R(I \cap h^{-}(\hat{s}_1)) \cap h^{-}(\hat{s}_2)$$

is computed. Generally, for any $i \geq 1$, 

$$M_i = R(M_{i-1}) \cap h^{-}(\hat{s}_i)$$

$$= R(R(M_{i-2}) \cap h^{-}(\hat{s}_{i-1})) \cap h^{-}(\hat{s}_i)$$

$$= R(R(R(M_{i-1}) \cap h^{-}(\hat{s}_{i-1})) \cap h^{-}(\hat{s}_{i-1})) \cap h^{-}(\hat{s}_i)$$

$$= ...$$

$$= R(R(...(I \cap h^{-}(\hat{s}_1)) \cap ... \cap h^{-}(\hat{s}_{i-1})) \cap h^{-}(\hat{s}_i))$$

is computed recursively. For some state $\hat{s}_k$, $k \geq 1$, if $M_k = \emptyset$, $\hat{s}_{k-1}$ is a failure state. Note that if $M_0 = \emptyset$, $\hat{s}_0$ is a failure state. To check whether or not a finite counterexample is spurious, $M_0$, $M_1$, $M_2$, ... are computed in turn until the first state $\hat{s}_k$ where $M_k = \emptyset$ is found, or the last state in the counterexample is reached.

For infinite counterexamples, it is more complicated to be dealt with since the last state in the counterexample can never be reached. Thus, a polynomial number of unwinding of the loop in the counterexample is needed [3]. That is an infinite counterexample can be reduced to a finite counterexample by unwinding the loop for a polynomial number of times. Accordingly, SplitPath can be used again to check whether or not this infinite counterexample is spurious.

4. Failure States and Spurious Counterexamples

In [4, 5], a spurious counterexample is informally defined by: a counterexample in the abstract model which does not exist in the concrete model. In this section, we give a formal definition for spurious counterexamples based on the formal definition of failure states.
To this end, $In_{\hat{s}_i}^0$, $In_{\hat{s}_i}^1$, ..., $In_{\hat{s}_i}^n$ and $In_{\hat{s}_i}$ are defined first:

\[
In_{\hat{s}_i}^0 = \{ s \mid s \in h^-(\hat{s}_i), s' \in h^-(s_{\hat{s}_i-1}) \text{ and } (s', s) \in R \}
\]

\[
In_{\hat{s}_i}^1 = \{ s \mid s \in h^-(\hat{s}_i), s' \in In_{\hat{s}_i}^0 \text{ and } (s', s) \in R \}
\]

\[
... \quad ...
\]

\[
In_{\hat{s}_i}^n = \{ s \mid s \in h^-(\hat{s}_i), s' \in In_{\hat{s}_i}^{n-1} \text{ and } (s', s) \in R \}
\]

\[
... \quad ...
\]

\[
In_{\hat{s}_i} = \bigcup_{i=0}^{\infty} In_{\hat{s}_i}^i
\]

Clearly, $In_{\hat{s}_i}^0$ denotes the set of states in $h^-(\hat{s}_i)$ with inputting edges from the states in $h^-(s_{\hat{s}_i-1})$, and $In_{\hat{s}_i}^1$ stands for the set of states in $h^-(\hat{s}_i)$ with inputting edges from the states in $In_{\hat{s}_i}^0$, and $In_{\hat{s}_i}^2$ means the set of states in $h^-(\hat{s}_i)$ with inputting edges from the states in $In_{\hat{s}_i}^1$, and so on. Thus, $In_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i)$ that are reachable from some state in $h^-(s_{\hat{s}_i-1})$ as illustrated in the lower gray part in Fig. 4. Note that there must exist a natural number $n$, such that $\bigcup_{i=0}^{n+1} In_{\hat{s}_i}^i = \bigcup_{i=0}^{n} In_{\hat{s}_i}^i$ since $h^-(\hat{s}_i)$ is finite. Note that for state $\hat{s}_0$,

\[
In_{\hat{s}_0}^0 = \{ s \mid s \in (h^-(\hat{s}_0) \cap I) \}
\]

\[
In_{\hat{s}_0}^1 = \{ s \mid s \in h^-(\hat{s}_0), s' \in In_{\hat{s}_0}^0 \text{ and } (s', s) \in R \}
\]

\[
... \quad ...
\]

That is only $In_{\hat{s}_0}^0$ is defined differently since there are no pre states.

Similarly, $Out_{\hat{s}_i}^0$, $Out_{\hat{s}_i}^1$, ..., $Out_{\hat{s}_i}^n$ and $Out_{\hat{s}_i}$ can also be defined.

\[
Out_{\hat{s}_i}^0 = \{ s \mid s \in h^-(\hat{s}_i), s' \in h^-(s_{\hat{s}_i+1}) \text{ and } (s, s') \in R \}
\]

\[
Out_{\hat{s}_i}^1 = \{ s \mid s \in h^-(\hat{s}_i), s' \in Out_{\hat{s}_i}^0 \text{ and } (s, s') \in R \}
\]

\[
... \quad ...
\]
\[
\text{Out}^n_{\hat{s}_i} = \{ s \mid s \in h^-(\hat{s}_i), s' \in \text{Out}^{n-1}_{\hat{s}_i} \text{ and } (s, s') \in R \}
\]

\[
\text{Out}_{\hat{s}_i} = \bigcup_{i=0}^{\infty} \text{Out}^i_{\hat{s}_i}
\]

Where \( \text{Out}^0_{\hat{s}_i} \) denotes the set of states in \( h^-(\hat{s}_i) \) with outputting edges to the states in \( h^-(s_{i+1}) \), and \( \text{Out}^1_{\hat{s}_i} \) stands for the set of states in \( h^-(\hat{s}_i) \) with outputting edges to the states in \( \text{Out}^0_{\hat{s}_i} \), and \( \text{Out}^2_{\hat{s}_i} \) means the set of states in \( h^-(\hat{s}_i) \) with outputting edges to the states in \( \text{Out}^1_{\hat{s}_i} \), and so on. Thus, \( \text{Out}_{\hat{s}_i} \) denotes the set of states in \( h^-(\hat{s}_i) \) from which some state in \( h^-(s_{i+1}) \) are reachable as depicted in the higher gray part in Fig. 4. Similar to \( \text{In}_{\hat{s}_i} \), there must exist a natural number \( n \), such that

\[
\bigcup_{i=0}^{n+1} \text{Out}^i_{\hat{s}_i} = \bigcup_{i=0}^{n} \text{Out}^i_{\hat{s}_i}.
\]

Note that for the last state \( \hat{s}_n \) in a finite counterexample,

\[
\text{Out}^0_{\hat{s}_n} = \{ s \mid s \in h^-(\hat{s}_n) \cap F \}
\]

\[
\text{Out}^1_{\hat{s}_n} = \{ s \mid s \in h^-(\hat{s}_n), s' \in \text{Out}^0_{\hat{s}_n}, \text{ and } (s, s') \in R \}
\]

where \( F \) is the set of states without any successors in the original model.

Accordingly, a failure state can be defined as follows.

Definition 1. (Failure States) A state \( \hat{s}_i \) in a counterexample \( \hat{\Pi} \) is a failure state if, and only if \( \text{In}_{\hat{s}_i} \cap \text{Out}_{\hat{s}_i} = \emptyset \).

Further, given a failure state \( \hat{s}_i \) in a counterexample \( \hat{\Pi} \), the set of the origins of \( \hat{s}_i, h^-(\hat{s}_i) \), is separated into three sets, \( D = \text{In}_{\hat{s}_i} \) (the set of dead states), \( B = \text{Out}_{\hat{s}_i} \) (the set of bad states) and \( I = h^-(\hat{s}_i) \setminus (D \cup B) \) (the set of the isolated states).

Definition 2. (Spurious Counterexamples) A counterexample \( \hat{\Pi} \) in an abstract model \( \hat{K} \) is spurious if there exists at least one failure state \( \hat{s}_i \) in \( \hat{\Pi} \).

Example 2. Fig. 5 shows a spurious counterexample where state \( \hat{2} \) is a failure state.

In the set, \( h^-(\hat{2}) = \{7, 8, 9\} \), of the origins of state \( \hat{2} \), 9 is a dead state, 7 is a bad state, and 8 is an isolated state.

5. Algorithms for Detecting Spurious Counterexamples

Based on the formal definition of spurious counterexample, new algorithms for checking whether or not a counterexample is spurious are presented in this section.
5.1. Algorithm by Detecting the First Failure State

Algorithm CheckSpurious-I takes a counterexample as input and outputs the first failure state in the counterexample. Note that a counterexample may be a finite path \(<s_0, s_1, ..., s_n>\), \(n \geq 0\), or an infinite path \(<s_0, s_1, ..., (s_i, ..., s_j)\omega>\), \(0 \leq i \leq j\), with a loop suffix (a suffix produced by a loop). For the finite one, it can be checked directly; while for an infinite one, we need only to check its Complete Finite Prefix (CFP) \(<s_0, s_1, ..., s_i, ..., s_j>\) since whether or not a state \(s_i\) is a failure state only relies on its pre and post states. It is pointed out that in the CFP \(<s_0, s_1, ..., s_i, ..., s_j>\) of an infinite counterexample,

\[
\text{Out}^0_{\hat{s}_j} = \{ s \mid s \in h^{-}(\hat{s}_j), s' \in h^{-}(\hat{s}_j) \text{ and } (s, s') \in R \}
\]

\[
\text{Out}^1_{\hat{s}_j} = \{ s \mid s \in h^{-}(\hat{s}_j), s' \in \text{Out}^0_{\hat{s}_j} \text{ and } (s, s') \in R \}
\]

... since the post state of \(\hat{s}_j\) is \(\hat{s}_i\).

\textbf{Algorithm 1 : CheckSpurious-I(\(\hat{\Pi}\))}

\textbf{Input:} a counterexample \(\hat{\Pi} = <\hat{s}_0, \hat{s}_1, ..., \hat{s}_n>\) in the abstract model \(\hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})\), and the original model \(K = (S, S_0, R, L)\)

\textbf{Output:} a failure state \(s_f\)

1: \textbf{Initialization:} int \(i = 0\);
2: \textbf{while} \(i \leq n\) \textbf{do}
3: \hspace{1em} if \(\text{In}_{\hat{s}_i} \cap \text{Out}_{\hat{s}_i} \neq \emptyset\), \(i = i + 1\);
4: \hspace{1em} else return \(s_f = \hat{s}_i\); break;
5: \textbf{end while}
6: \textbf{if} \(i == n + 1\), return \(\hat{\Pi}\) is a real counterexample;
Algorithm Analyzing. In algorithm CheckSPURIOUS-I, to check whether or not a state \( \hat{s}_i \) is a failure state only relies on \( \hat{s}_i \)'s pre and post states, \( \hat{s}_{i-1} \) and \( \hat{s}_{i+1} \); while in algorithm SplitPath, to check state \( \hat{s}_i \) is up to the prefix, \( \hat{s}_0, ..., \hat{s}_{i-1} \), of \( \hat{s}_i \). Based on this, to check a periodic infinite counterexample, several repetitions of the periodic parts are needed in SplitPath. In contrast, this can be easily done by checking the complete finite prefix \( \langle \hat{s}_1, \hat{s}_2, ..., \hat{s}_i, ..., \hat{s}_j \rangle \) in algorithm CheckSPURIOUS-I. Thus, the polynomial number of unwinding of the loop can be avoided. That is for infinite counterexamples, the finite prefix to be checked will be polynomial shorter than the one in algorithm SplitPath.

5.2. Algorithm by Detecting the Heaviest Failure State

In algorithm SplitPath and CheckSPURIOUS-I, always, the first failure state is detected. Then further refinement will be done based on the analysis of this failure state. Possibly, several failure states may occur in one counterexample, so which

Algorithm 2: CheckSPURIOUS-II(\( \hat{\Pi} \))

**Input:** a counterexample \( \hat{\Pi} = \langle \hat{s}_0, \hat{s}_1, ..., \hat{s}_n \rangle \) in the abstract model \( \hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L}) \), and the original model \( K = (S, S_0, R, L) \)

**Output:** the heaviest failure state \( s_f \)

1: Sorting: the heavier the earlier (stored in array int \( w[n+1] \));
2: Initialization: int \( i = 0 \);
3: while \( i \leq n \) do
4:   if \( In_{s_{w[i]}} \cap Out_{s_{w[i]}} \neq \emptyset \), \( i = i + 1 \);
5:   else return \( s_f = s_{w[i]} \); break;
6: end while
7: if \( i == n + 1 \), return \( \hat{\Pi} \) is a real counterexample;

one is chosen to be refined is not considered in SplitPath. Obviously, if a failure state shared by more paths is refined, a number of model checking iterations will be saved in the whole abstract-refinement loop. With this consideration, we will check the states which is shared by more paths first. To do so, for an abstract state \( \hat{s} \) as illustrated in Fig.6, \( EIn(\hat{s}) \) and \( EOut(\hat{s}) \) are defined. \( EIn(\hat{s}) \) equals to the number of edges connecting to the states in \( h^- (\hat{s}) \) from the states outside of \( h^- (\hat{s}) \); and \( EOut(\hat{s}) \) is the number of edges connecting to the states out of \( h^- (\hat{s}) \) from the states in \( h^- (\hat{s}) \). Accordingly, \( EIn(\hat{s}) \times EOut(\hat{s}) \) is the number of the paths where \( \hat{s} \) occurs. For convenience, we call \( EIn(\hat{s}) \times EOut(\hat{s}) \) the weight of the abstract state \( \hat{s} \). Based on this, algorithm CheckSPURIOUS-II is given for detecting the heaviest
failure state in a counterexample. In \textsc{CheckSpurious-II}, an array \( w[i] \) is used to store the indexes of the states in the counterexample by the heavier the earlier.

5.3. Parallel Algorithms

Considering whether or not a state \( \hat{s}_i \) is a failure state only relies on the pre- and post- states, \( \hat{s}_{i-1} \) and \( \hat{s}_{i+1} \), of \( \hat{s}_i \), the algorithm can be naturally paralleled as presented in algorithm \textsc{CheckSpurious-III} and \textsc{CheckSpurious-IV}.

In \textsc{CheckSpurious-III}, anytime, if a failure state is detected by a processor, all the processors will be stop and the failure state is returned. Otherwise, if no failure states are reported, the counterexample is a real one. That is the algorithm always reports the first detected failure state obtained by the processors. Note that a boolean array \( c[n] \) is used to indicate whether or not a state in the counterexample is a failure one. Initially, for all \( 0 \leq i \leq n \), \( c[i] \) is undefined (\( c[i] = \perp \)). \( c[i] == \text{true} \) means state \( \hat{s}_i \) is not a failure state.

\begin{algorithm}
\caption{\textsc{CheckSpurious-III}(\( \hat{\Pi} \))}
\begin{algorithmic}
\Input a counterexample \( \hat{\Pi} = \langle \hat{s}_0, \hat{s}_1, ..., \hat{s}_n \rangle \) in the abstract model \( \hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L}) \), and the original model \( K = (S, S_0, R, L) \) in shared memory
\Output a failure state \( \hat{s}_f \)
\State \textbf{Initialization}: \textit{bool} \( c[n+1] = \{ \perp, ..., \perp \} \);
\For {\( k = 0 \) to \( n \) do in parallel}
\State \textbf{if} \( \text{In}_{\hat{s}_i} \cap \text{Out}_{\hat{s}_i} \neq \emptyset \), \( c[k] = \text{true} \);
\State \textbf{else} return \( \hat{s}_f = \hat{s}_k \); stop all processors;
\EndFor
\State \textbf{if} for all \( 0 \leq i \leq n \), \( c[i] == \text{true} \), return \( \hat{\Pi} \) is a real counterexample;
\end{algorithmic}
\end{algorithm}

In \textsc{CheckSpurious-IV}, the weight of the states are considered, and always the heaviest failure state is found.
Algorithm 4: \textsc{CheckSpurious-IV}($\hat{\Pi}$)

\begin{algorithmic}
  \Require a counterexample $\hat{\Pi} = < \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n >$ in the abstract model $\hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$, and the original model $K = (S, S_0, R, L)$ in shared memory
  \State $n$: the number of processors
  \State $k$: processor id
  \Ensure a failure state $s_f$
  \State 1: \textbf{Sorting}: the heavier state first (stored in array int $w[n+1]$);
  \State 2: \For {$k = 0$ to $n$} do in parallel \Do
  \State 3: \textbf{if} $\text{In}_{\hat{s}_k} \cap \text{Out}_{\hat{s}_k} \neq \emptyset$, $c[k] = \text{true}$;
  \State 4: \textbf{else} return $c[k] = \text{false}$;
  \State 5: \EndDo
  \State 6: \textbf{if} for all $1 \leq i \leq n$, $c[i] = \text{true}$, return $\hat{\Pi}$ is a real counterexample;
  \State 7: \textbf{else} return $s_f = s_i$ such that $c[i] == \text{false}$, and for any state $\hat{s}_j$, if the weight of $\hat{s}_j$ is heavier than $\hat{s}_i$, $c[j] == \text{true}$;
\end{algorithmic}

6. Conclusion

Based on a formal definition of spurious paths, a novel approach for detecting spurious counterexamples are presented in this paper. In the new approach, whether or not a state $\hat{s}_i$ is a failure state only relies on $\hat{s}_i$’s pre- and post- states in the counterexample. So, for infinite counterexample, the polynomial number of unwinding of the loop can be avoided. Further, the algorithm can be easily improved by detecting the heaviest failure state such that a number of model checking iterations can be saved in the whole abstract-refinement loop. Also, the algorithm can be naturally parallelled.

The presented algorithms are useful in improving the abstract based model checking, especially the counterexample guided abstraction refinement model checking. In the near future, the proposed algorithm will be implemented and integrated into the tool CEGAR. Further, some case studies will be conducted to evaluate the algorithms.

References

[1] E.M.Clarke and E.A.Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, LNCS 131, Springer, 1981.
[2] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Proceedings of the 5th international symposium on programming, pp.337-350, 1981.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample guided abstraction refinement, in Proc. 12th Int. Conf. Computer-Aided Verification (CAV00), vol. 1855, E. Emerson and A. Sistla, Eds. New York, 2000.

[4] Clarke, E.M., Gupta, A., Strichman, O. SAT Based Counterexample-Guided Abstraction-Refinement. IEEE Trans. Computer Aided Design, vol.23, no. 7, pp. 1113-1123, July 2004.

[5] Fei He, Xiaoyu Song, William N. N. Hung, Ming Gu, Jiaguang Sun. Integrating Evolutionary Computation with Abstraction Refinement for Model Checking. IEEE Trans. Computers 59(1): 116-126 (2010)

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

[7] Fei He, Xiaoyu Song, William N. N. Hung, Ming Gu, Jiaguang Sun. Integrating Evolutionary Computation with Abstraction Refinement for Model Checking. IEEE Trans. Computers 59(1): 116-126 (2010)

[8] R.P. Kurshan. Computer Aided Verification of Coordinating Processes. Princeton Univ. Press, 1994.

[9] C. Wang, B. Li, H. Jin, G.D. Hachtel, 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.

[10] 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.

[11] 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.

[12] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proc. Symp. Principles of Programming Languages, pp. 58-70, 2002.

[13] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, 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.

[14] S.G. Govindaraju, D.L. Dill. Counterexample-Guided Choice of Projections in Approximate Symbolic Model Checking. Proc. Intl Conf. Computer-Aided Design (ICCAD), pp. 115-119, 2000.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. Technical Report CMU-CS-00-103, Computer Science, Carnegie Mellon University, 2000.