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
{ctian, zhhduan}@mail.xidian.edu.cn

***Abstract*—Abstraction is one of the most important strategies for dealing with the state space explosion problem in model checking. With an abstract model, the state space is largely reduced, however, a counterexample found in such a model that does not satisfy the desired property may not exist in the concrete model. Therefore, how to check whether a reported counterexample is spurious is a key problem in the abstraction-refinement loop. Particularly, there are often thousands of millions of states in systems of industrial scale, how to check spurious counterexamples in these systems practically is a significant problem. In this paper, by re-analyzing spurious counterexamples, a new formal definition of spurious path is given. Based on it, efficient algorithms for detecting spurious counterexamples are presented.**

By the new algorithms, when dealing with infinite counterexamples, the finite prefix to be analyzed will be polynomially shorter than the one dealt by the existing algorithm. Moreover, in practical terms, the new algorithms can naturally be parallelized that makes multi-core processors contributes more in spurious counterexample checking. In addition, by the new algorithms, the state resulting in a spurious path (*false state*) that is hidden shallower will be reported earlier. Hence, as long as a *false state* is detected, lots of iterations for detecting all the *false states* will be avoided. Experimental results show that the new algorithms perform well along with the growth of system scale.

***Index Terms*—model checking, formal verification, abstraction, refinement, parallel algorithm.**

I. Introduction

Model checking is an important approach to improve the reliability of hardware, software, multi-agent systems, communication protocols, embedded systems and so forth. The term model checking was coined by Clarke and Emerson [7], as well as Sifakis and Queille [19], independently. The earlier model checking algorithms explicitly enumerated the reachable states of the system in order to check the correctness of the system. This restricted the capacity of model checkers to systems with a few million states. Since the number of states can grow exponentially with the number of variables, early implementations were only able to handle small designs and did not scale to examples with industrial complexity. To combat this, various methods, such as abstraction [9], [10], [11], [12], [13], partial order reduction [3], [4], ROBDD [5], [6] and bounded model checking [8], etc. techniques 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 one of the most important ones which has been widely used in software model checking. In several software model checkers, SLAM [25], [26] and BLAST [27] for instance, Counter-Example Guided Abstraction Refinement (CEGAR) [9], [10], [11], [12], [13] based abstract model checking has been well implemented. Abstraction technique preserves all the behaviors of a 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 certainly 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 (called abstraction-refinement loop) until either a real counterexample is found or the abstract model satisfies the property.

In the abstraction-refinement loop, how to check whether a reported counterexample is spurious is a key problem. In [13], Algorithm SplitPath is presented for checking whether a counterexample is spurious, and a SAT solver is employed to implement it [10], [11]. In SplitPath, whether a counterexample is spurious can be checked by detecting *failure states* in the counterexample. If a *failure state* is found, the counterexample is spurious, otherwise, the counterexample is a real one. However, whether a state, say $\hat{s}_i$, is a *failure state* relies on the whole prefix of the counterexample $\langle \hat{s}_0, \hat{s}_1, \cdots, \hat{s}_i \rangle$. Therefore, to check a counterexample $\Pi = \langle \hat{s}_0, \hat{s}_1, \cdots, \hat{s}_n \rangle$, each state in Π should be checked sequentially. Moreover, to check a counterexample with infinite length, a polynomial number of unwinding of the loop in the infinite path is required [12], [13]. For systems with a small state space, the polynomial number of unwinding of the loop is tolerable. However, for systems with large state space, i.e. a common software system, the polynomial growth of the number of the states to be checked might lead to the exhaustion of memory. Therefore, effective algorithms for checking spurious counterexamples are significant in making abstract model checking to be practical.

In this paper, based on the definition of *false states*, spurious paths are re-analyzed, and a new approach for checking spurious counterexamples is proposed. With this approach, whether a counterexample is spurious depends on the existence of *false states* in the counterexample. There are several merits

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

of the new approach. First, instead of the prefix, to check whether a state $\hat{s}_i$ is *false* is only up to $\hat{s}_i$'s previous and successor states in the counterexample. Based on this, for an infinite counterexample, the polynomial number of unwinding of the loop can be avoided. Second, the algorithm can easily be improved by detecting *the heaviest false state* such that a number of model checking iterations can be saved in the whole abstract-refinement loop. Thirdly, the algorithm can naturally be parallelled. This will largely improve the efficiency for checking spurious counterexamples with thousands of millions of states in big examples with industrial scale. Finally, the algorithms analyze each state in the counterexample gradually by considering its previous and successor states. Thus, some *false states* that are hidden shallower will be detected easier. This will be useful in practise since in the whole abstract-refinement loop, anytime, one *false state* is enough for the refinement of the counterexample. We have developed a counterexample checker containing both the new proposed algorithms and the existing one. The tool is implemented on randomly generated models in different scales. Experimental results show that the new algorithms perform well along with the growth of system scale.

The rest part of the paper is 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 SPLITPATH is explained briefly. In Section 4, a new formal definition of spurious counterexample is given with respect to the definition of *false states*. Further, in Section 5, algorithms for checking whether a counterexample in the abstract model is spurious are presented. Experimental results are given in Section 6. Finally, conclusions are drawn in Section 7.

II. RELATED WORK

We focus on the Counter-Example Guided Abstraction Refinement, CEGAR, framework which was first proposed by Kurshan [18]. Based on the basic CEGAR, some variations were given [11], [21], [10], [1], [16], [15], [14] in the past years. Most of them use a model checker and try to get rid of spurious counterexamples to achieve a concrete counterexample or a proof of the desired property. Recently, CEGAR was also improved for the purpose of abstract model checking of shared-variable concurrent programs [28], [29].

The closest works to ours are those where the abstract models are obtained by making some of the variables invisible. To the best of our knowledge, this abstraction method was first proposed by Clarke, etc. [11], [10]. With their approach, abstraction is performed by selecting a set of variables (or latches in circuits) to be invisible. In each iteration, a standard Ordered Binary Decision Diagram (OBDD)-based symbolic model checker is used to check whether or not the abstract model satisfies the desired property which is described by a formula in temporal logic. If a counterexample is reported by the model checker, it is simulated with the concrete system by a SAT solver. It tells us that the model is satisfiable if the counterexample is a real one, otherwise, the counterexample is a spurious one and a *failure state* is found which is the last state in the longest prefix of the counterexample that is still satisfiable. Subsequently, the *failure state* is used to refine the abstraction by making some invisible variables visible. In the method given by Clarke, etc., ALGORITHM SPLITPATH is used to check whether a counterexample is spurious, and a SAT solver is employed to implement it [10], [11]. SPLITPATH is carried out by detecting *failure states* in the counterexample. If a *failure state* is found, the counterexample is spurious, otherwise, the counterexample is a real one. With this method, to check a counterexample with infinite length, a polynomial number of unwinding of the loop in the infinite path is required [12], [13].

III. ABSTRACTION AND REFINEMENT LOOP

As usual, a Kripke structure [2] is used to model a system. Let $V = \{v_1, \cdots, v_n\}$ ranging over a finite domain $D \cup \{\perp\}$ be the set of variables involved in a system. For any $v_i \in V$, $1 \le i \le n$, a set of the valuations of v_i is defined by $\Sigma_{v_i} = \{v_i = d \mid d \in D \cup \{\perp\}\}$ where $v_i = \perp$ means v_i is undefined. Further, the set Σ of all the possible states of the system is defined by $\Sigma = \Sigma_{v_1} \times \cdots \times \Sigma_{v_n}$ for each $v_i \in V$. Let AP be the set of propositions. A Kripke structure over AP is a tuple $K = (S, I, R, L)$, where $S \subseteq \Sigma$ is the set of states (i.e. a state in S is a valuation of variables in V), $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, $L : S \to 2^{AP}$ is the labeling function. For convenience, $s(v)$ is employed to denote the value of v at state s. A path in a Kripke structure is a sequence of states, $\Pi = \langle s_1, s_2, \cdots \rangle$, where $s_1 \in S_0$ and $(s_i, s_{i+1}) \in R$ for any $i \ge 1$. For convenience, we use $R(s)$ to denote the set of direct successors of a state $s \in S$, $R(S')$ the set of direct successors of all states in S'. More generally, $R^i(s)$ means the set of states reachable from s after i times of transitions, and $R^i(S')$ the set of states reachable from all states in S' after i times of transitions.

There are several techniques for obtaining the abstract models [16], [18], [20]. We follow the counterexample guided abstraction and refinement method proposed by Clarke, et al. where abstraction is performed by selecting a set of variables which are insensitive to the desired property to be invisible [11]. Following the idea given in [11], we separate V into two parts V_V and V_I such that $V = V_V \cup V_I$. V_V stands for the set of visible variables while V_I denotes the set of invisible variables. Invisible variables are those we do not care about and will be ignored when building the abstract model. In an original model $K = (S, S_0, R, L)$, all variables are visible ($V_V = V$, $V_I = \emptyset$). To obtain an abstract model $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$, some variables, e.g. $V_X \subseteq V$, are selected to be invisible ($V_V = V \setminus V_X$, $V_I = V_X$). Thus, the set of all possible states in the abstract model will be: $\hat{\Sigma} = \Sigma_{v_1} \times \cdots \times \Sigma_{v_k}$, where $k = |V_V|$, and for each $1 \le i \le k$, $v_i \in V_V$. For a state $s \in S$ and a state $\hat{s} \in \hat{S}$, $\hat{s}$ is called the mapping of s in the abstract model by selecting V_V as the set of visible variables iff $s(v) = \hat{s}(v)$ for all $v \in V_V$. Formally, $\hat{s} = h(s, V_V)$ is used to denote that $\hat{s}$ is the mapping of s in the abstract model by selecting V_V as the set of visible variables.

Inversely, s is called the origin of $\hat{s}$, and the set of origins of $\hat{s}$ is denoted by $h^{-}(\hat{s}, V_V)$.

Therefore, given an original model $K = (S, S_0, R, L)$ and a selected set of visible variables V_V, an abstract model $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ can be obtained by Algorithm ABSTRACT as shown below.

Algorithm 1 : ABSTRACT(K, V_V)

Input: an original model $K = (S, S_0, R, L)$ and a set of selected visible variables V_V

Output: an abstract model $\hat{K}$=($\hat{S}$,$\hat{S_0}$,$\hat{R}$,$\hat{L}$)

1: $\hat{S} = \{\hat{s} \in \hat{\Sigma} \mid$ there exists $s \in S$ such that $h(s, V_V) = \hat{s}\}$;
2: $\hat{S_0} = \{\hat{s} \in \hat{S} \mid$ there exists $s \in S_0$ such that $h(s, V_V) = \hat{s}\}$;
3: $\hat{R} = \{(\hat{s_1}, \hat{s_2}) \mid \hat{s_1}, \hat{s_2} \in \hat{S}$, and there exist $s_1, s_2 \in S$ such that $h(s_1, V_V) = \hat{s_1}, h(s_2, V_V) = \hat{s_2}$ and $(s_1, s_2) \in R\}$;
4: $\hat{L}(\hat{s}) = \bigcup_{s \in S, h(s, V_V) = \hat{s}} L(s)$;
5: return $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$;

Example 1: As illustrated in Fig. 1, the concrete model is a Kripke structure with four states. Initially, the system

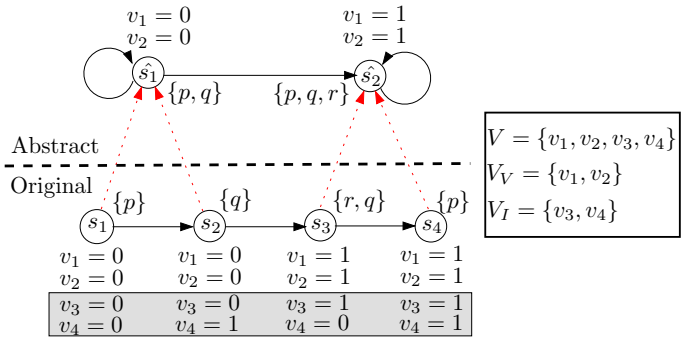


Fig. 1. Abstraction function

has four variables v_1, v_2, v_3 and v_4. Suppose that v_3 and v_4 are selected to be invisible. By ALGORITHM ABSTRACT, an abstract model with two states is obtained. In the abstract model, $\hat{s_1}$ is the mapping of s_1 and s_2, while $\hat{s_2}$ is the mapping of s_3 and s_4. $(\hat{s_1}, \hat{s_2}) \in \hat{R}$ since $(s_2, s_3) \in R$, and $(\hat{s_1}, \hat{s_1})$, $(\hat{s_2}, \hat{s_2}) \in \hat{R}$ because of (s_1, s_2) and $(s_3, s_4) \in R$. □

After the abstract model is obtained, a model checker is utilized to check whether the abstract model can satisfy the desired property. If the property is satisfied, the (original) system can satisfy the property. Nevertheless, if the 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 spurious. Thus, when a spurious counterexample is found, the abstraction should be refined in order to eliminate the spurious behaviors. This process is repeated (abstraction-refinement loop) until either a real counterexample is found or the abstract model satisfies the property.

The abstraction-refinement loop is depicted in Fig.2. Initially, the abstract model M' is obtained by the abstrac-

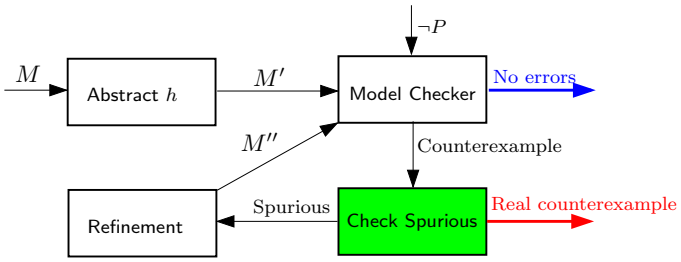


Fig. 2. Abstraction refinement loop

t algorithm. 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 spurious checker which is used to check whether a counterexample is spurious. If the counterexample is not spurious, it will be a real counterexample that violates the system property; otherwise, the counterexample is spurious, and a refining tool is used to refine the abstract model [11], [13], [15], [17], [21]. Subsequently, the refined abstract model is checked by 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 a counterexample is spurious (the green part in Fig.2). Several methods about how to refine an abstract model can be found in [13], [17], [23], [24].

IV. SPURIOUS PATHS

To check a spurious counterexample efficiently, we first show why spurious paths occur intuitively by an example. Then we briefly present the basic idea of ALGORITHM SPLITPATH which is used in [12], [13] for checking whether a counterexample is spurious.

A. Why Spurious Paths?

Abstraction technique preserves all the behaviors of the concrete system but may introduce behaviors that are not present originally. Therefore, when checking the abstract model using a model checker, some reported counterexamples might not be real counterexamples that violate the desired property. This problem can intuitively be illustrated by the traffic lights controller example [13].

Example 2: For the traffic light controller system in the l.h.s of Fig.3 involving variables *color* and *state*, by making variable *color* to be invisible, an abstract model can be obtained as shown in the r.h.s of Fig.3. We want to prove $\Box\Diamond(state = stop)$ (any time, the state of the light will be *stop* sometimes in the future). By implementing a model checker on the abstract model, a counterexample, $\langle \hat{s_1}, \hat{s_2}, \hat{s_2}, \hat{s_2}, \cdots \rangle$ will be reported. However, in the concrete model, such a behavior cannot be found. So, this is not a real counterexample. □

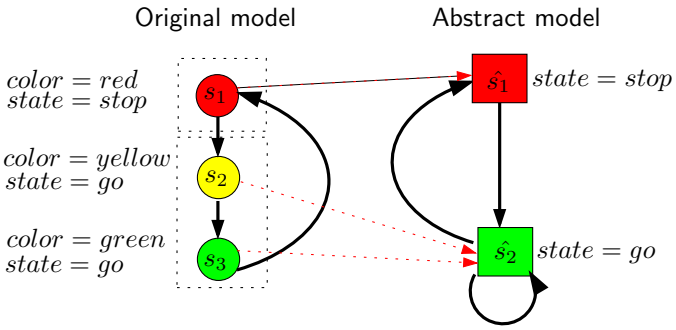


Fig. 3. Traffic Light Controller

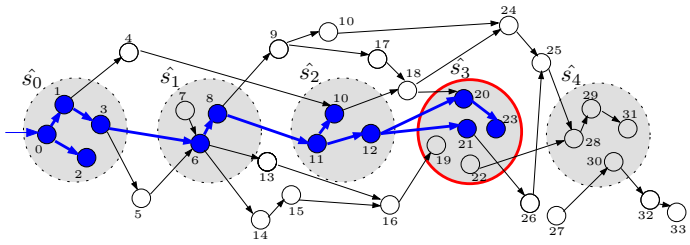


Fig. 4. Algorithm SplitPath

B. Exposition of Algorithm SplitPath

In [13], Algorithm SplitPath is presented for checking whether a finite counterexample is spurious. We present it formally below.

Let $\Pi = \langle \hat{s}_0, \hat{s}_1, \cdots, \hat{s}_n \rangle$ be a finite counterexample in an abstract model. The basic idea of Algorithm SplitPath is to compute the set of reachable states M_i in each $h^-(\hat{s}_i, V_V)$ $(0 \le i \le n)$ from I under the following two conditions:

(1) for each i $(1 \le i \le n)$, any state $s \in M_i$ is reachable from each M_k $(0 \le k \le i-1)$; and
(2) for each i $(1 \le i \le n)$, any state $s \in M_i$, and $(s', s) \in R$, it has either $s' \in M_i$ or $s' \in M_{i-1}$.

Formally, M_0^k, $k \ge 0$, can be computed by:

$$\begin{array}{lcl} M_0^0 & = & I \cap h^-(\hat{s}_0, V_V) \\ M_0^1 & = & R(M_0^0) \cap h^-(\hat{s}_0, V_V) \\ M_0^2 & = & R(M_0^1) \cap h^-(\hat{s}_0, V_V) \\ & \cdots & \\ M_0^k & = & R(M_0^{k-1}) \cap h^-(s_0, \hat{V_V}) \end{array}$$

Then, we have $M_0 = \bigcup_{k=0}^{\infty} M_0^k$. Similarly, for each $1 \le i \le n$, M_i^k $(k \ge 0)$ can be computed by:

$$\begin{array}{lcl} M_i^0 & = & R(M_{i-1}) \cap h^-(\hat{s}_i, V_V) \\ M_i^1 & = & R(M_i^0) \cap h^-(\hat{s}_i, V_V) \\ M_i^2 & = & R(M_i^1) \cap h^-(\hat{s}_i, V_V) \\ & \cdots & \\ M_i^k & = & R(M_i^{k-1}) \cap h^-(\hat{s}_i, V_V) \end{array}$$

Accordingly, $M_i = \bigcup_{k=0}^{\infty} M_i^k$.

Note that there must exist a natural number m, such that $\bigcup_{k=0}^{m+1} M_i^k = \bigcup_{k=0}^{m} M_i^k$ since $h^-(\hat{s}_i, V_V)$ is finite. Intuitively, each state in M_i is reachable from I, M_0, $\cdots$, M_{i-1}; and cannot pass through any state outside of M_0, $\cdots$, M_{i-1}, and M_i.

For some state $\hat{s}_k$, $k \ge 1$, if $M_k = \emptyset$, $\hat{s_{k-1}}$ is called a *failure state*. To check whether a finite counterexample is spurious, M_0, M_1, M_2, $\cdots$ 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. The following example illustrates how Algorithm SplitPath works.

Example 3: Fig.3 depicts a Kripke structure and a counterexample, $\langle \hat{s}_0, \hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4 \rangle$, in the abstract model. In this counterexample, $I = \{0\}$.

$$\begin{array}{lcl} M_0^0 & = & I \cap h^-(\hat{s}_0, V_V) = \{0\} \\ M_0^1 & = & R(M_0^0) \cap h^-(\hat{s}_0, V_V) = \{1, 2\} \\ M_0^2 & = & R(M_0^1) \cap h^-(\hat{s}_0, V_V) = \{3\} \\ M_0^3 & = & R(M_0^2) \cap h^-(\hat{s}_0, V_V) = \emptyset \end{array}$$

Thus, $M_0 = M_0^0 \cup M_0^1 \cup M_0^2 \cup M_0^3 = \{0, 1, 2, 3\}$.

$$\begin{array}{lcl} M_1^0 & = & R(M_0) \cap h^-(\hat{s}_1, V_V) = \{6\} \\ M_1^1 & = & R(M_1^0) \cap h^-(\hat{s}_1, V_V) = \{8\} \\ M_1^2 & = & R(M_1^1) \cap h^-(\hat{s}_1, V_V) = \emptyset \end{array}$$

It has $M_1 = M_1^0 \cup M_1^1 \cup M_1^2 = \{6, 8\}$. Further,

$$\begin{array}{lcl} M_2^0 & = & R(M_1) \cap h^-(\hat{s}_2, V_V) = \{11\} \\ M_2^1 & = & R(M_2^0) \cap h^-(\hat{s}_2, V_V) = \{10, 12\} \\ M_2^2 & = & R(M_2^1) \cap h^-(\hat{s}_2, V_V) = \emptyset \end{array}$$

So, $M_2 = M_2^0 \cup M_2^1 \cup M_2^2 = \{10, 11, 12\}$.

$$\begin{array}{lcl} M_3^0 & = & R(M_2) \cap h^-(\hat{s}_3, V_V) = \{20, 21\} \\ M_3^1 & = & R(M_3^0) \cap h^-(\hat{s}_3, V_V) = \{23\} \\ M_3^2 & = & R(M_3^1) \cap h^-(\hat{s}_3, V_V) = \emptyset \end{array}$$

Hence, $M_3 = M_3^0 \cup M_3^1 \cup M_3^2 = \{20, 21, 23\}$.

$$M_4^0 = R(M_3) \cap h^-(\hat{s}_4, V_V) = \emptyset$$

So, $M_4 = M_4^0 = \emptyset$. Therefore, M_4 is an empty set and $\hat{s}_3$ is a *failure state*. In Fig.3, M_0, M_1, M_2, M_3, and M_4 are the set of blue nodes in $\hat{s}_0$, $\hat{s}_1$, $\hat{s}_2$, $\hat{s}_3$, and $\hat{s}_4$, respectively. The *failure state* is depicted in red circle. □

For infinite counterexamples, it is more complicated to dealing 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 [13]. That is, an infinite counterexample can be reduced to a finite one by unwinding the loop for a polynomial number of times. Accordingly, SplitPath can be used again to check whether this infinite counterexample is spurious.

In a counterexample, there may exist more than one states that make the path to be spurious. In fact, by Algorithm SplitPath, always, the only *failure* state (if it exists) is detected. However, in abstraction-refinement loop, the elimination of any state that makes the counterexample to be spurious will be enough for the refinement of the abstract model. Therefore, it is unnecessary to detect only the *failure state* in the counterexample for the refinement.

C. Algorithm Analysis

For a finite counterexample, ALGORITHM SPLITPATH is linear in the size[1] of the counterexample. Nevertheless, for an infinite counterexample, ALGORITHM SPLITPATH is polynomial in the size[2] of the counterexample since an infinite counterexample is reduced to a finite one by unwinding the loop for polynomial number of times. Moreover, in the verification of systems with industrial scale, it is possible that a counterexample contains thousands of millions of states. So how to make ALGORITHM SPLITPATH more practical is significant in abstract model checking.

V. YET ANOTHER DEFINITION OF SPURIOUS COUNTEREXAMPLES

In [11], [17], a spurious counterexample is described by: a counterexample in the abstract model which does not exist in the concrete model. By the analysis in the previous section, it can be preciously defined by: a counterexample with one *failure state*. In this section, we redefine spurious counterexamples from another aspect.

For convenience, $In^0_{\hat{s}_i}$, $In^1_{\hat{s}_i}$, $\cdots$, and $In^n_{\hat{s}_i}$ are defined:

$$\begin{aligned}
In^0_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in h^-(\hat{s_{i-1}}, V_V) \text{ and } (s', s) \in R\}\\
In^1_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in In^0_{\hat{s}_i} \text{ and } (s', s) \in R\}\\
&\cdots\\
In^n_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in In^{n-1}_{\hat{s}_i} \text{ and } (s', s) \in R\}
\end{aligned}$$

Then we have $In_{\hat{s}_i} = \bigcup_{i=0}^{\infty} In^i_{\hat{s}_i}$. Here $In^0_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ with inputting edges from the states in $h^-(\hat{s_{i-1}}, V_V)$, and $In^1_{\hat{s}_i}$ stands for the set of states in $h^-(\hat{s}_i, V_V)$ with inputting edges from the states in $In^0_{\hat{s}_i}$, and $In^2_{\hat{s}_i}$ means the set of states in $h^-(\hat{s}_i, V_V)$ with inputting edges from the states in $In^1_{\hat{s}_i}$, and so on. Thus, $In_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ that are reachable from some state in $h^-(\hat{s_{i-1}}, V_V)$ as illustrated in the lower irregular shape in Fig.5. Note that there must exist a natural number n, such that $\bigcup_{i=0}^{n+1} In^i_{\hat{s}_i} = \bigcup_{i=0}^{n} In^i_{\hat{s}_i}$ since $h^-(\hat{s}_i, V_V)$ is finite. Particularly, for state $\hat{s_0}$,

$$\begin{aligned}
In^0_{\hat{s}_0} &= \{s \mid s \in (h^-(\hat{s_0}, V_V) \cap I)\}\\
In^1_{\hat{s}_0} &= \{s \mid s \in h^-(\hat{s_0}, V_V), s' \in In^0_{\hat{s}_0} \text{ and } (s', s) \in R\}\\
&\cdots\\
In^n_{\hat{s}_0} &= \{s \mid s \in h^-(\hat{s_0}, V_V), s' \in In^{n-1}_{\hat{s}_0} \text{ and } (s', s) \in R\}
\end{aligned}$$

That is only $In^0_{\hat{s}_0}$ is defined differently since $\hat{s_0}$ has no previous states.

Symmetrically, $Out^0_{\hat{s}_i}$, $Out^1_{\hat{s}_i}$, $\cdots$, and $Out^n_{\hat{s}_i}$ are also defined.

[1]The size of a finite counterexample can be measured by the length of the counterexample as well as the number of states in the original model that are involved in the counterexample.

[2]The size of an infinite counterexample is measured by the number of individual states in the counterexample as well as the number of states in the original model that are involved in the counterexample.

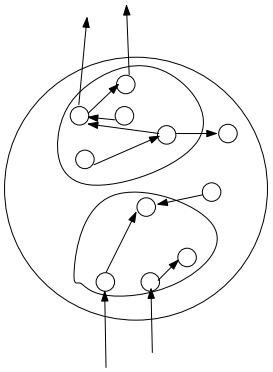


Fig. 5. $In_{\hat{s}_i}$ and $Out_{\hat{s}_i}$

$$\begin{aligned}
Out^0_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in h^-(\hat{s_{i+1}}, V_V) \text{ and } (s, s') \in R\}\\
Out^1_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in Out^0_{\hat{s}_i} \text{ and } (s, s') \in R\}\\
&\cdots\\
Out^n_{\hat{s}_i} &= \{s \mid s \in h^-(\hat{s}_i, V_V), s' \in Out^{n-1}_{\hat{s}_i} \text{ and } (s, s') \in R\}
\end{aligned}$$

Thus, $Out_{\hat{s}_i} = \bigcup_{i=0}^{\infty} Out^i_{\hat{s}_i}$. Here $Out^0_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ with outputting edges to the states in $h^-(\hat{s_{i+1}}, V_V)$, and $Out^1_{\hat{s}_i}$ stands for the set of states in $h^-(\hat{s}_i, V_V)$ with outputting edges to the states in $Out^0_{\hat{s}_i}$, and $Out^2_{\hat{s}_i}$ means the set of states in $h^-(\hat{s}_i, V_V)$ with outputting edges to the states in $Out^1_{\hat{s}_i}$, and so on. Thus, $Out_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ from which some state in $h^-(\hat{s_{i+1}}, V_V)$ are reachable as depicted in the higher iregular shape in Fig.5. Similar to $In_{\hat{s}_i}$, there must exist a natural number n, such that $\bigcup_{i=0}^{n+1} Out^i_{\hat{s}_i} = \bigcup_{i=0}^{n} Out^i_{\hat{s}_i}$. It is also pointed out that for the last state $\hat{s_n}$ in a finite counterexample,

$$\begin{aligned}
Out^0_{\hat{s}_n} &= \{s \mid s \in h^-(\hat{s_n}, V_V) \cap F\}\\
Out^1_{\hat{s}_n} &= \{s \mid s \in h^-(\hat{s_n}, V_V), s' \in Out^0_{\hat{s}_n}, \text{ and } (s, s') \in R\}\\
&\cdots\\
Out^n_{\hat{s}_n} &= \{s \mid s \in h^-(\hat{s_n}, V_V), s' \in Out^{n-1}_{\hat{s}_n}, \text{ and } (s, s') \in R\}
\end{aligned}$$

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

Based on the definitions of $In_{\hat{s}_i}$ and $Out_{\hat{s}_i}$, if $In_{\hat{s}_i} \cap Out_{\hat{s}_i} = \emptyset$, Π is spurious obviously since $\hat{s_{i-1}}$ cannot reach to $\hat{s_{i+1}}$ through $\hat{s}_i$.

Example 4: Fig.6 shows a spurious counterexample where $In_{\hat{2}} = \{9\}$, $Out_{\hat{2}} = \{7\}$, and $In_{\hat{2}} \cap Out_{\hat{2}} = \emptyset$. Obviously, $\langle \hat{0}, \hat{1}, \hat{2}, \hat{3} \rangle$ is a spurious counterexample that does not exist in the original model. □

However, for each state $\hat{s}_i$ in a counterexample Π such that $In_{\hat{s}_i} \cap Out_{\hat{s}_i} \neq \emptyset$, Π may still be spurious. For instance, Fig.7 shows a counterexample without any state $\hat{s}_i$ ($i = 0$, 1, or 2) such that $In_{\hat{s}_i} \cap Out_{\hat{s}_i} = \emptyset$ ($In_{\hat{s}_0} \cap Out_{\hat{s}_0} = \{s_1, s_2\}$, $In_{\hat{s}_1} \cap Out_{\hat{s}_1} = \{s_5\}$, and $In_{\hat{s}_2} \cap Out_{\hat{s}_2} = \{s_6\}$). Nevertheless, this counterexample is obviously a spurious one because green states are not reachable from the red ones.

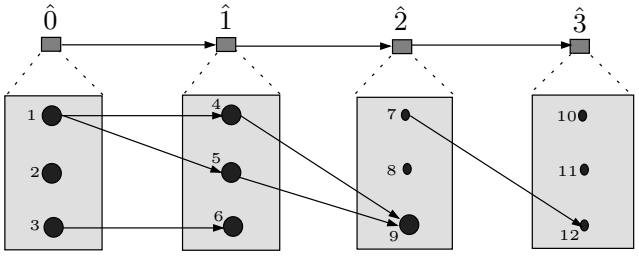


Fig. 6. A spurious path where $In_{\hat{2}} \cap Out_{\hat{2}} = \emptyset$

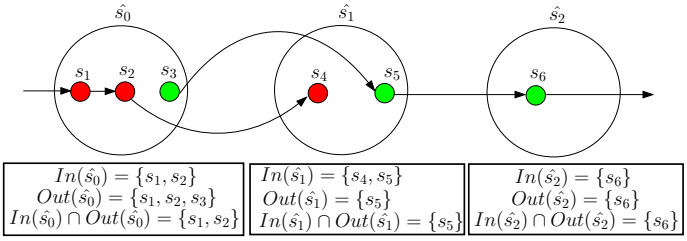


Fig. 7. A spurious path

Let $E_i = In_{\hat{s_i}} \cap Out_{\hat{s_i}}$ for each $0 \leq i \leq n$. As illustrated in Fig.8, given a counterexample Π with $In_{\hat{s_i}} \cap Out_{\hat{s_i}} \neq \emptyset$ for

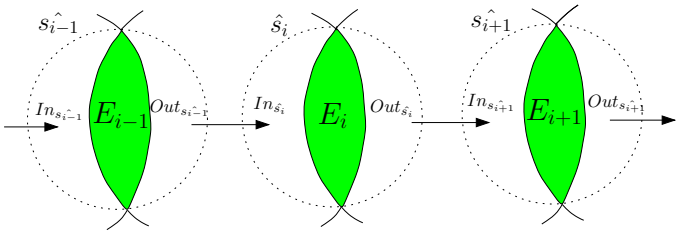


Fig. 8. Checking spurious counterexamples

each state $\hat{s_i}$ in Π, to check whether Π is spurious, we need to further check $\langle E_0, E_1, \cdots, E_n \rangle$ again similar to $\langle \hat{s_0}, \hat{s_1}, \cdots, \hat{s_n} \rangle$ by treating each of E_0, E_1, $\cdots$, E_n as a state. This process will be repeated until $In(E_i) = Out(E_i)$ (i.e. $E_i = In(E_i) \cap Out(E_i)$ will keep unchanged) for each i, or $In(E_i) \cap Out(E_i) = \emptyset$ for some i. In case $In(E_i) = Out(E_i)$ for each i, the counterexample is a real one; otherwise, if there exists some i such that $In(E_i) \cap Out(E_i) = \emptyset$, the counterexample is a spurious one. For instance, for the counterexample in Fig.7, we need to further check $\langle E_0, E_1, E_2 \rangle$ as illustrated in Fig. 9. Note that here E_0,

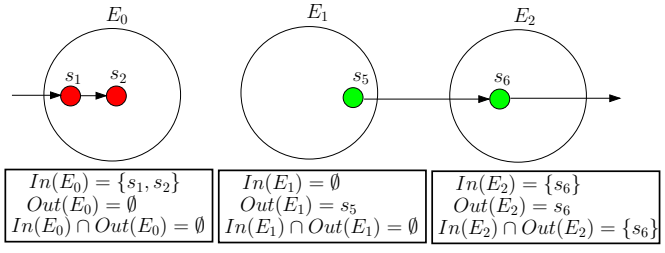


Fig. 9. A spurious path

E_1, and E_2 equals to $In_{\hat{s_0}} \cap Out_{\hat{s_0}} = \{s_1, s_2\}$, $In_{\hat{s_1}} \cap Out_{\hat{s_1}} = \{s_5\}$, and $In_{\hat{s_2}} \cap Out_{\hat{s_2}} = \{s_6\}$ in Fig. 7, respectively. By computing $In(E_i) \cap Out(E_i)$, $i = 0$, 1, or 2, it can be obtained that the counterexample is spurious since $In(E_i) \cap Out(E_i) = \emptyset$ for both $i = 0$ and 1.

The above procedure for checking whether a counterexample is spurious is intuitively illustrated in Fig. 10. For clarity, we use E_i^j ($j \geq 1$) as a temporal variable to record the result of $In(E_i^{j-1}) \cap Out(E_i^{j-1})$. Here the superscript $j \geq 0$ indicates the times of the run. In the 0th run, for each $0 \leq i \leq n$, E_i^0 is assigned with $h^-(\hat{s_i}, V_V)$.

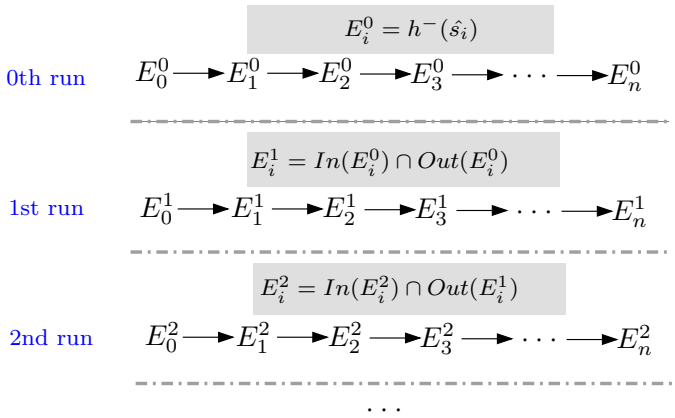


Fig. 10. Checking spurious counterexamples

For convenience, a state $\hat{s_i}$ with $In(E_i^j) \cap Out(E_i^j) = \emptyset$ is called a *false state*. Further, given a *false state* $\hat{s_i}$ in a counterexample $\hat{\Pi}$, the set of the origins of $\hat{s_i}$, $h^-(\hat{s_i}, V_V)$, is divided into three sets, $\mathcal{D} = In_{E_i^j}$ (the set of dead states), $\mathcal{B} = Out_{E_i^j}$ (the set of bad states) and $\mathcal{I} = h^-(\hat{s_i}, V_V) \setminus (\mathcal{D} \cup \mathcal{B})$ (the set of the isolated states). For instance, state $\hat{2}$ in Fig. 6 is a *false state*. In $h^-(\hat{2}, V_V)$, i.e. $\{7, 8, 9\}$, $\{9\}$ is the set of dead states, $\{7\}$ the set of bad states, and $\{8\}$ the set of isolated states. In a counterexample $\langle \hat{s_0}, \cdots, \hat{s_n} \rangle$, suppose $\hat{s_i}$ and $\hat{s_j}$ are two *false states* with $E_i^n = \emptyset$ and $E_j^m = \emptyset$ ($m > n$), respectively. We call the *false state* $\hat{s_i}$ is hidden shallower than $\hat{s_i}$. By the above procedure, always, the shallowest *false state* is detected if it exists.

Armed with these notations, a spurious counterexample is formally defined below.

Definition 1: **(Spurious Counterexamples)** A counterexample $\hat{\Pi}$ in an abstract model $\hat{K}$ is spurious if, and only if, there exists at least one *false state* in $\hat{\Pi}$. □

To confirm the equivalence of the new definition of spurious counterexamples with the original one, the following theorem is proved.

Theorem 1: A counterexample Π is spurious if, and only if, there exists at least one *false* state in Π.

Proof: $\Rightarrow$: By the definitions of *false* states, if there exists a *false* state in Π, Π does not exist in the concrete model.

$\Leftarrow$: If there exist no *false states* in Π, it has $E_0 \neq \emptyset$, $E_1 \neq \emptyset$, $\cdots$, and $E_n \neq \emptyset$ for each $1 \leq i \leq n$ (by the definition of *false* states). Thus, for each state $s \in E_i$, s is reachable from E_{i-1}, and can access to E_{i+1} since s is in both $In_{\hat{E_{i-1}}}$ and $Out_{\hat{E_{i+1}}}$.

So, for each state in E_n, it is reachable from E_0, through E_0, E_1, $\cdots$, and E_{n-1}. Accordingly, $\langle \hat{s_0}, \hat{s_1}, \cdots, \hat{s_n} \rangle$ exists in the concrete model. □

VI. Algorithms for Checking Spurious Counterexamples

Based on the new definition of spurious counterexamples, algorithms for checking whether a counterexample is spurious are presented in this section. Generally speaking, to check whether a counterexample is spurious can be determined by detecting the existence of *false states* in the path. If a *false state* is found, the counterexample is spurious; otherwise, the counterexample is a real one.

A. Detecting False States on $\langle E_0, \cdots, E_n \rangle$

By the definition of spurious counterexamples based on *false states*, to check whether a counterexample is spurious, we need to detect the *false states* on $\langle E_0, \cdots, E_n \rangle$, recursively. We first present Algorithm CheckFalse-I for checking *false states* on $\langle E_0, \cdots, E_n \rangle$ in one run. Algorithm CheckFalse-I takes $\langle E_0, \cdots, E_n \rangle$ as input and outputs the first detected *false state* if it exists.

Algorithm 2 : CheckFalse-I($\langle E_0, \cdots, E_n \rangle$)

Input: $\langle E_0, \cdots, E_n \rangle$

Output: the first detected false state s_f in the counterexample

1: **Initialization**: *int* $i = 0$;
2: **while** $i \leq n$ **do**
3: **if** $In_{\hat{E}_i} \cap Out_{\hat{E}_i} \neq \emptyset$, $i = i + 1$;
4: **else** return $s_f = \hat{s}_i$; break;
5: **end while**

In Algorithm CheckFalse-I, to check whether a state $\hat{s}_i$ is a *false state* only relies on $\hat{E}_i$'s previous and successor states, $\hat{E_{i-1}}$ and $\hat{E_{i+1}}$; while in Algorithm SplitPath, to check state $\hat{s}_i$ is up to checking the whole prefix, $\langle \hat{s_0}, \cdots, \hat{s_{i-1}} \rangle$, of $\hat{s}_i$. Therefore, compared with Algorithm SplitPath, CheckFalse-I can be parallelized naturally as presented in Algorithm CheckFalse-II.

In CheckFalse-II, anytime, if a *false state* is detected by a processor, all the processors will stop and the *false state* is returned. That is the algorithm always reports the first detected *false state* obtained by the processors. Note that a boolean array $c[n]$ is used to indicate whether a state in the counterexample is a false one. Initially, for all $0 \leq i \leq n$, $c[i]$ is $\perp$ ($c[i]$ is undefined). $c[i] == false$ means state $\hat{s}_i$ is a *false state*.

B. Checking Spurious Counterexamples

Based on the algorithms for detecting *false states*, Algorithm CheckSpurious-I is presented to check whether a given (finite) counterexample is spurious. In CheckSpurious-I, initially, E_0, $\cdots$, E_n is initialized by $h^-(\hat{s_0}, V_V)$, $\cdots$, $h^-(\hat{s_n}, V_V)$, respectively; then CheckFalse-II is called recursively until each of $E_0, \cdots, E_n$ keeps unchanged or a *false state* is detected. To perform it, a *Boolean array c* with length $n + 1$ is utilized to memorize the situation of each E_i. If $c[k] == \perp$, it means that currently neither $In_{E_k} = Out_{E_k}$ nor $In_{E_k} \cap Out_{E_k} = \emptyset$. That is E_k needs to be further updated with $In_{E_k} \cap Out_{E_k}$, and then $In_{E_k} \cap Out_{E_k}$ should be recalculated. If $c[k] == true$, it indicates that E_k will keep unchanged ($\hat{s_k}$ cannot be a *false state*). Any time, if $In_{E_k} \cap Out_{E_k} = \emptyset$, the algorithm will stop and a *false state* is returned. In case $c[k] == true$ for each k, $\langle \hat{s_0}, \cdots, \hat{s_n} \rangle$ is reported to be a real counterexample.

Algorithm 3 : CheckFalse-II($\hat{\Pi}$)

Input: $\langle E_0, \cdots, E_n \rangle$
n: the number of processors
k: processor id
Output: a false state s_f

1: **Initialization**: *bool* $c[n + 1] = \{\perp, \cdots, \perp\}$;
2: **for** $k = 0$ to n do in parallel **do**
3: **if** $In_{\hat{E}_k} \cap Out_{\hat{E}_k} = \emptyset$ **then**
4: $c[k] = false$; return $s_f = \hat{s}_k$; stop all processors;
5: **end if**
6: **end for**

Algorithm 4 : CheckSpurious-I($\hat{\Pi}$)

Input: $\langle \hat{s_0}, \cdots, \hat{s_n} \rangle$
n: the number of processors
k: processor id
Output: a false state s_f

1: **Initialization**: $E_0 = h^-(\hat{s_0})$; $\cdots$; $En = h^-(\hat{s_n})$;
bool $c[n + 1] = \{\perp, \cdots, \perp\}$;
2: **CheckFalse:**
3: **for** $k = 0$ to n in parallel **do**
4: **if** $c[k] == \perp$ **then**
5: **if** $In_{E_k} \cap Out_{E_k} = \emptyset$, **then**
6: return $s_f = \hat{s}_k$; stop all processors;
7: **end if**
8: **if** $In_{E_k} \cap Out_{E_k} = E_k$, **then**
9: $c[k] = ture$;
10: **end if**
11: **if** $In_{E_k} \cap Out_{E_k} \neq E_k$ and $In_{E_k} \cap Out_{E_k} \neq \emptyset$, **then**
12: $E_k = In_{E_k} \cap Out_{E_k}$;
13: **end if**
14: **end if**
15: **end for**
16: **if** for all $0 \leq i \leq n$, $c[i] == ture$, return $\hat{\Pi}$ has no false states;
17: **else** goto **CheckFalse**;

Note that a counterexample may be a finite path $\langle s_0, s_1, \cdots, s_n \rangle$, $n \geq 0$, or an infinite path $\langle s_0, s_1, \cdots, (s_i, \cdots, s_j)^\omega \rangle$, $0 \leq i \leq j$, with a loop suffix (a suffix produced by a loop). For the finite one, it can be checked directly with Algorithm CheckSpurious-I while for an infinite one, we need only to check its Complete Finite Prefix (CFP) $\langle s_0, s_1, \cdots, s_i, \cdots, s_j \rangle$ since whether or not a state s_i is a *false state* only relies on its previous and successor states. It is pointed out that in the

CFP $\langle s_0, s_1, \cdots, s_i, \cdots, s_j\rangle$ of an infinite counterexample,

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

since the successor state of $\hat{s}_j$ is $\hat{s}_i$.

C. Algorithm for Detecting the Heaviest False State

In ALGORITHM CHECKSPURIOUS-I, always, the first detected *false state* is returned. Then further refinement will be done based on the analysis of this *false state*. Possibly, several *false states* may occur in one counterexample, so which one is chosen to be refined is not considered. Obviously, if a *false state* shared by more paths is refined, a number of model checking iterations are hopefully to be saved in the whole abstract-refinement loop.

Under this consideration, we will check the state shared by more paths first. To do so, for an abstract state $\hat{s}$ as illustrated in Fig.11, $EIn(\hat{s})$ and $EOut(\hat{s})$ are defined. $EIn(\hat{s})$ equals to

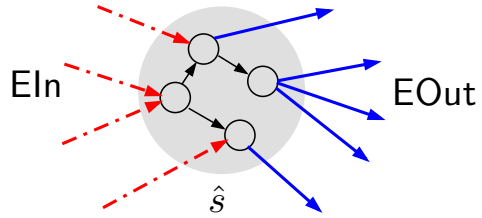


Fig. 11. In and out edges

the number of edges connecting to the states in $h^-(\hat{s}, V_V)$ from the states outside of $h^-(\hat{s}, V_V)$; and $EOut(\hat{s})$ is the number of edges connecting to the states out of $h^-(\hat{s}, V_V)$ from the states in $h^-(\hat{s}, V_V)$. 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 $\hat{s}$.

In CHECKSPURIOUS-II, The counterexample is recursively checked until each E_i keeps unchanged or is detected as a *false state*. Then the heaviest *false state* is returned for further refinement. That is all the *false states* are detected out first, and then the heaviest one is returned.

D. Algorithm Analysis

Compared with ALGORITHM SPLITPATH for detecting *failure states*, to check whether a state is a *false state* by ALGORITHM CHECKSPURIOUS-I only relies on its previous and direct successor states. Thus when dealing with infinite counterexamples by ALGORITHM CHECKSPURIOUS-I, the finite prefix to be checked will be polynomially shorter than the one dealt by ALGORITHM SPLITPATH. That is given an infinite counterexample, by ALGORITHM CHECKSPURIOUS-I, the number of the states to be analyzed will be polynomial less than the one to be considered by ALGORITHM SPLITPATH. Further, for the finite counterexamples, in each iteration, E_0, E_1, $\cdots$, E_n, are checked in parallel. Also, the algorithms analyze each state in the counterexample gradually by considering its previous and successor states. Thus, some *false states* that are hidden shallower will be detected earlier, and lots of iterations can be avoided in practise.

Algorithm 5 : CHECKFALSE-IV($\hat{\Pi}$)

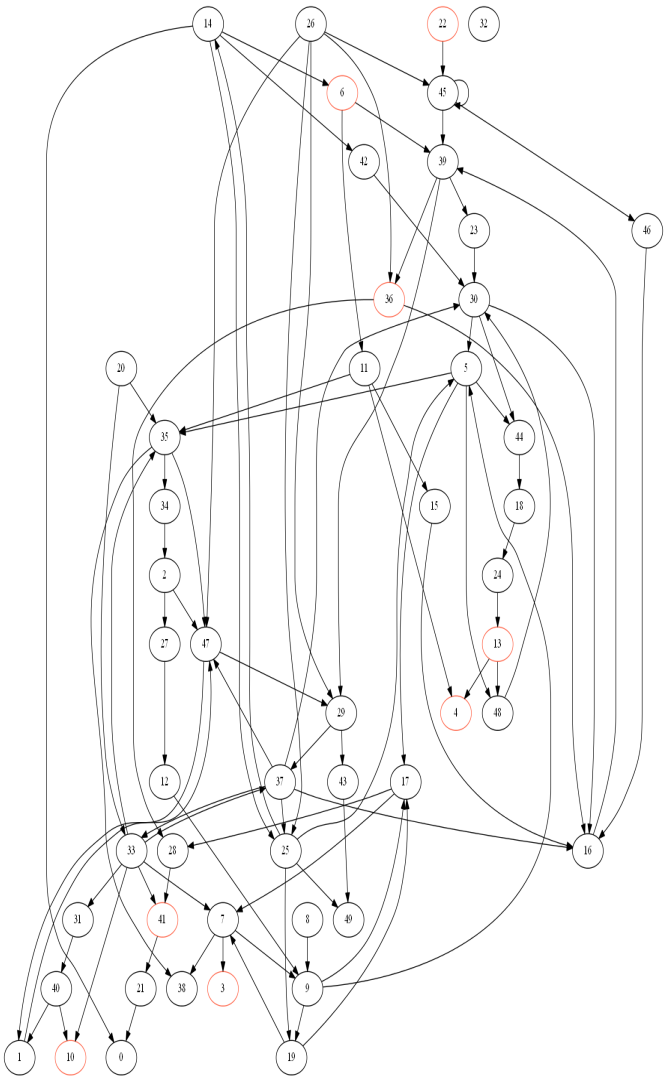
Input: a counterexample $\hat{\Pi}=\langle \hat{s}_0, \hat{s}_1, \cdots, \hat{s}_n\rangle$ in the abstract model $\hat{K}=(\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$, and the original model $K = (S, S_0, R, L)$ in shared memory
n: the number of processors
k: processor id
Output: a false state s_f

1: **Initialization**: $E_0 = h^-(\hat{s}_0)$; $\cdots$; $En = h^-(\hat{s}_n)$;
   bool $c[n+1] = \{\perp, \cdots, \perp\}$;
2: **CHECKFALSE:**
3: **for** $k = 0$ to n in parallel **do**
4: **if** $c[k] = \perp$ **then**
5: **if** $In_{E_k} \cap Out_{E_k} = \emptyset$, **then**
6: $c[k] = false$;
7: **end if**
8: **if** $In_{E_k} \cap Out_{E_k} = E_k$, **then**
9: $c[k] = ture$;
10: **end if**
11: **if** $In_{E_k} \cap Out_{E_k} \neq E_k$ and $In_{E_k} \cap Out_{E_k} \neq \emptyset$, **then**
12: $E_k = In_{E_k} \cap Out_{E_k}$;
13: **end if**
14: **end if**
15: **end for**
16: **if** for each $0 \leq i \leq n$, $c[i] \neq \perp$, **then**
17: **if** $c[i] == ture$ for each i **then**
18: return $\hat{\Pi}$ has no false states;
19: **end if**
20: **if** $c[i]==false$ and $\hat{s}_i$ is the heaviest one among the false states **then**
21: return $\hat{s}_i$;
22: **end if**
23: **else**
24: goto **CHECKFALSE**;
25: **end if**

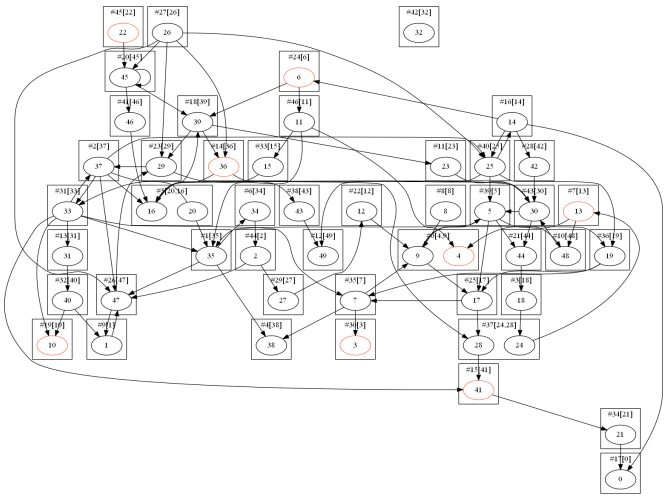
VII. EVALUATION

To evaluate the proposed approach, we implemented a Counterexample Checker (called CC for short) (http://web.xidian.edu.cn/home/ctian/files/20120808_210955.zip) that contains both Algorithm CHECKSPURIOUS-I and SPLITPATH for checking spurious counterexamples. Note that here we do not compare the results of Algorithm CHECKSPURIOUS-II with CHECKSPURIOUS-I and SPLITPATH since its advantages can only be presented in the whole abstraction-refinement loop. To make the results more general, we (1) randomly generate the original models by providing the numbers of states and transitions; (2) achieve the abstract models by providing the insensitive variables; and (3) select a path (i.e. counterexample) randomly in the abstract model. Subsequently, we implement Algorithm CHECKSPURIOUS-I and SPLITPATH on the same selected path, respectively, and record the time consumed by both the two algorithms. We use Graphviz 2.28 [22] to display the original and abstract models. Fig.12 (a) and (b) shows a randomly generated model and its abstract model, respectively. We just present small models, since for the ones containing several thousands of states, the graphs are unclear.

The following experiments are performed on 4-core PC. We randomly generate five models with the size



(a) Original model



(b) Abstract model

Fig. 12. Original and abstract models illustrated by Graphviz

(s,t) being $(10,50)$, $(50,1250)$, $(100,5000)$, $(500,125000)$, $(1000,500000)$, $(5000,13000000)$, $(10000,50000000)$, and $(50000,150000000)$, respectively. Here s means the number of states while t the number of transitions in the original model. By selecting counterexamples at random, Algorithms CHECKSPURIOUS-I are compared with Algorithm SPLITPATH. The experimental data are recorded in Table VII and the curves depicting the time consumed by the two algorithms on different models are presented in Fig. 13. The vertical axis depicts the size of the model, and the horizontal axis describes the time (ms) used for checking whether or not the selected path is spurious. From the table and the curves, it can be seen that in case the scale of the model is small, because of the expense in creating and destroying threads, the merit of parallel algorithm is not obvious. However, as the scale of the model grows, the advantages of the new algorithm turn out to be evident.

TABLE I
RESULTS OF EXPERIMENT

Model Size		SPLITPATH		CHECKSPURIOUS-I	
States (number)	Transitions (number)	5 models (ms)	Average (ms)	5 Models (ms)	Average (ms)
10	50	2	2.4	4	4.4
		3		4	
		3		4	
		2		5	
		2		5	
50	1250	2	2.6	5	5.2
		3		6	
		2		4	
		3		5	
		3		5	
100	5000	3	3.2	4	3.4
		3		4	
		3		4	
		4		4	
		3		5	
500	125000	4	4.6	4	5
		6		5	
		4		5	
		4		5	
		5		6	
1000	500000	5	6	5	7.6
		6		17	
		6		8	
		8		4	
		5		4	
5000	13000000	16	14.8	10	16.8
		13		43	
		15		5	
		15		13	
		15		13	
10000	50000000	61	49.4	45	37.8
		50		64	
		50		34	
		45		42	
		41		4	
50000	180000000	192	174.8	51	58.42
		151		53	
		218		89	
		201		42	
		112		57	

VIII. CONCLUSION

Based on the formal analysis of spurious paths, a new approach for detecting spurious counterexamples is presented in this paper. With this approach, for an infinite counterexample,

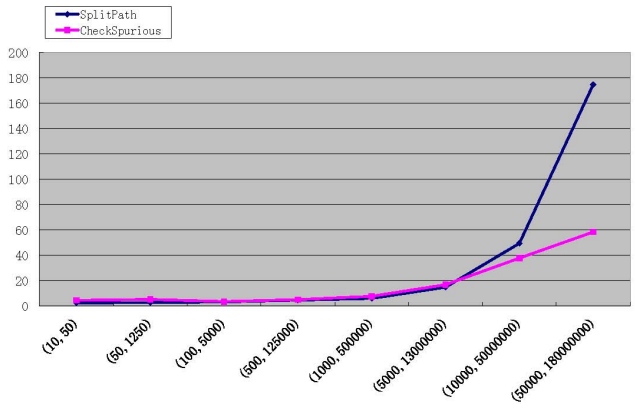


Fig. 13. Comparation between Algorithm CheckSpurious-I and SplitPath

the polynomial number of unwinding of the loop is avoided. That is the finite prefix to be checked will be polynomially shorter than the one checked by the existing algorithm. Besides, for a given finite counterexample (or finite prefix of an infinite counterexample), the new algorithm still performs well. The reasons are two: (1) The algorithm is parallelized in each run for detecting *false states*; (2) The shallowest *false state* is always detected such that many iterations in the algorithm will be avoided in practise.

The presented algorithms are useful in improving efficiency of abstract model checking, especially CEGAR based abstract model checking. In the near future, together with our previous results for refining abstract models, the new proposed algorithms will be implemented and integrated into several model checkers such as SLAM and BLAST where CEGAR is implemented. In addition, we will also investigates how these algorithms can be applied in abstract model checking of shared-variable concurrent programs.

ACKNOWLEDGMENT

Zhenhua Duan is the corresponding author. This research is supported by the NSFC Grant No. 61003078, 61272117, 60910004, 61133001, 61272118, and 61202038, 973 Program Grant No. 2010CB328102 and ISN Lab Grant No. ISN1102001. We thank Wenqiang Fan in implementing the counterexample checker CC.

REFERENCES

[1] 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.

[2] S.A.Kripke. *Semantical analysis of modal logic I: normal propositional calculi*, Z. Math. Logik Grund. Math. 9, 67-96, 1963.

[3] R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenig/in. *Static partial order reduction*. In Tools for the Construction and Analysis of Systems, LNCS 1394, pages 345C357, 1998.

[4] K.L. McMillan. *A technique of state space search based on unfolding*. Formal Methods in System Design 6: 45-65, 1995.

[5] E.M. Clarke, O. Grumberg, D. Long. *Verification tools for finite state concurrent systems*. In A Decade of Concurrency-Reflections and Perspectives, LNCS 803, pages 124-175, 1993.

[6] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[7] E.M. Clarke and E.A.Emerson. *Desigh and syntesis of of synchronization skeletons using branching time temporal logic*. In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, LNCS 131, Springer, 1981.

[8] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking* volume 58 of Advances in computers. Academic Press, 2003

[9] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. *Counterexample-guided abstraction refinement for symbolic model checking*. J. ACM 50(5): 752-794 (2003)

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

[12] 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.

[13] E.M.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.

[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] 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.

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

[17] F. He, X. Song, W.N. N. Hung, M. Gu, J. Sun. *Integrating Evolutionary Computation with Abstraction Refinement for Model Checking*. IEEE Trans. Computers 59(1): 116-126 (2010)

[18] R.P.Kurshan. *Computer Aided Verificaton of Coordinating Processes.* Princeton Univ. Press, 1994.

[19] 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.

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

[21] 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.

[22] The homepage of Graphviz. http://www.graphviz.org/.

[23] C. Tian, Z. Duan, N. Zhang. *An efficient approach for abstraction-refinement in model checking*. Theoretical Computer Science, doi:10.1016/j.tcs.2011.12.014, 2012.

[24] C.Tian and Z.Duan. Making Abstraction Efficient in Model Checking. The 17th Annual International Computing and Combinatorics Conference (COCOON 2011) LNCS6842, 90-111, 2011.

[25] Thomas Ball and Rupak Majumdar and Todd Millstein and Sriram K. Rajamani. *Automatic predicate abstraction of C programs*. IN PROC. ACM PLDI, 2001, 203–213, ACM Press.

[26] T. Ball, E. Bounimova, R. Kumar, V. Levin. *SLAM2: Static Driver Verification with Under 4% False Alarms*. FMCAD 2010: 35-42.

[27] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. *Software Verification with Blast*. Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN), Lecture Notes in Computer Science 2648, Springer-Verlag, pages 235-239, 2003.

[28] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Thomas Wahl. *Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs*. CAV 2011: 356-371

[29] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, Thomas Wahl. *Counterexample-guided abstraction refinement for symmetric concurrent programs*. Formal Methods in System Design 41(1): 25-44 (2012)