Theoretical Computer Science 461 (2012) 76–85

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

An efficient approach for abstraction-refinement in model checking☆

Cong Tian, Zhenhua Duan *, Nan Zhang

ICTT and ISN Laboratory, Xidian University, Xi'an, 710071, PR China

ARTICLE INFO

Keywords:
Model checking
Formal verification
Abstraction
Refinement
Algorithm

ABSTRACT

Abstraction is one of the most important strategies for dealing with the state space explosion problem in model checking. In an abstract model, the state space is largely reduced, however, a counterexample found in such a model may not be a real counterexample. Accordingly, the abstract model needs to be further refined where an NP-hard state separation problem is often involved. In this paper, a novel approach is presented, in which extra boolean variables are added to the abstract model for the refinement. With this approach, not only the NP-hard state separation problem can be avoided, but also a smaller refined abstract model can be obtained.

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 small designs and did not scale to examples with industrial complexity. To combat the methods such as abstraction, partial order reduction, OBDD, symmetry and bound 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 the industrial community.

Among the techniques for reducing the state space, abstraction is certainly the most important one. The abstraction technique preserves all the behaviors of the concrete system but may introduce behaviors that are not presented 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.

There are many techniques for generating the initial abstraction and refining the abstract models. We follow the counterexample guided abstraction and refinement method proposed by Clarke et al. [5] where abstraction is performed by selecting a set of variables which are insensitive to the desired property to be invisible. In each iteration, a model checker is employed to check whether or not the abstract model satisfies the desired property. If a counterexample is reported,

☆ This research is supported by the NSFC Grant No. 61003078, 91018010, 61133001, 60873018 and 60910004, 973 Program Grant No. 2010CB328102, SRFDP Grant No. 200807010012 and ISN Lab Grant No. ISN1102001.

* Corresponding author. Tel.: +86 13571496501.
E-mail address: zhenhua_duan@126.com (Z. Duan).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.12.014

it is simulated with the concrete model by a SAT solver or checked by other algorithms. Then, if the counterexample is checked to be spurious, a set of invisible variables are made visible to refine the abstract model. With this method, to find the coarsest (or smallest) refined model is NP-hard [3]. Further, it is important to find a small set of variables in order to keep the size of the abstract state space smaller. However, to find the smallest set of variables is also NP-hard [9]. To combat this, an Integer Linear Program (ILP) based separation algorithm which outputs the minimal separating set is given in [5]. A polynomial approximation algorithm based on Decision Trees Learning (DTL) is also presented in [5]. Moreover, Heuristic-Guided separating algorithms are presented in [8], and evolutional algorithms are introduced in [9] for the state separation problem. These approximate algorithms are compared with experimental results.

In this paper, we follow the abstract method used in [5,8,9] by selecting some set of variables to be invisible. Then we evaluate the counterexample with Algorithm CheckSpurious. When a failure state is detected, instead of selecting some invisible variables to be visible, extra variables are added to the abstract model for refinement. With this method, the NP-hard state separation problem can be avoided, and a smaller refined abstract model can be also obtained.

The remainder of the paper is organized as follows. The next section briefly presents related work concerning abstraction refinement in model checking. In Section 3, the abstraction algorithm is formalized by making insensitive variables invisible. In Section 4, by formally defining spurious counterexamples, the algorithm for checking whether or not a counterexample in the abstract model is spurious is presented. Further, a new abstraction refinement algorithm is given. Subsequently, an abstraction model checking framework based on the new proposed algorithms is illustrated in Section 5. Finally, conclusions are drawn in Section 6.

2. Related work

We focus on the Counter-Example Guided Abstraction Refinement (CEGAR) framework which was first proposed by Kurshan [10]. Recently, some variations of the basic CEGAR were given [5,11–16]. 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.

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 et al. [5,12]. 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. With this method, to find the smallest refined model is NP-hard [3]. To combat this, both optimal exponential and approximate polynomial algorithms are given. The first one is done by using an ILP solver which is known to be NP complete; and the second one is based on machine learning approaches.

Some heuristics for refinement variable selection were presented in [8]. It studied effective greedy heuristic algorithms on the state separation problem. Further, in [6], a probabilistic learning approach which utilized the sample learning technique, an evolutionary algorithm and effective heuristics were proposed. The performances were illustrated by experimental results.

3. Abstraction function

As usual, a Kripke structure [4] is used to model a system. Let $V = \{v_1, \ldots, v_n\}$ ranging over a finite domain $D \cup \{\perp\}$ be the set of variables involved in a system. For any $v_i \in V$, $1 \leq i \leq n$, 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 of all the possible states of the system, Σ, is defined by, $\Sigma = \Sigma_{v_1} \times \cdots \times \Sigma_{v_n}$. Let AP be the set of propositions. A Kripke structure over AP is a tuple $K = (S, S_0, R, L)$, where $S \subseteq \Sigma$ is the set of states (i.e. a state in S is a valuation of variables in V), $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, $L : S \to 2^{AP}$ is the labeling function. 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, $\Pi = s_1, s_2, \ldots$, where $s_1 \in S_0$ and $(s_i, s_{i+1}) \in R$ for any $i \geq 1$.

Following the idea given in [5], we separate V into two parts V_V and V_I with $V = V_V \cup V_I$. V_V stands for the set of visible variables while V_I denotes the set of invisible variables. Invisible variables are those that we do not care about and will be ignored when building the abstract model. In the original model $K = (S, S_0, R, L)$, all variables are visible ($V_V = V, V_I = \emptyset$). To obtain the 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 $k = |V_V| < n$, and for each $1 \leq i \leq k$, $v_i \in V_V$. That is $\hat{S} \subseteq \hat{\Sigma}$. 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 making V_V visible 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 making V_V visible. 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 the original model $K = (S, S_0, R, L)$ and the selected visible variables V_V, the 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: the original model $K = (S, S_0, R, L)$ and a set of selected visible variables V_V

Output: the 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: $L(\hat{s}) = \bigcup_{s \in S, h(s,V_V)=\hat{s}} L(s)$;

5: return $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$;

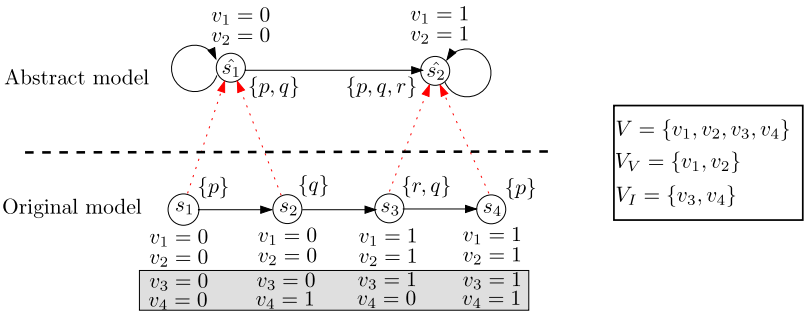


Fig. 1. Abstraction.

Example 1. As illustrated in Fig. 1, the original model is a Kripke structure with four states. Initially, the system has four variables v_1, v_2, v_3 and v_4. Assume 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 projection of s_1 and s_2, while $\hat{s_2}$ is the projection 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), $(s_3, s_4) \in R$. □

4. Refinement

4.1. Why refining?

It can be observed that the state space is largely reduced in the abstract model. However, when implementing model checking with the abstract model, some reported counterexamples will not be real counterexamples that violate the desired property, since the abstract model contains more paths than the original model. This is further illustrated in the traffic lights controller example [3] given below.

Example 2. For the traffic light controller in Fig. 2, we want to prove $\Box\Diamond(state = stop)$ (any time, the state of the light will be *stop* sometime in the future). By implementing model checking with the abstract model in the right hand side of Fig. 2 where the variable *color* is made invisible, a counterexample, $\hat{s_1}, \hat{s_2}, \hat{s_2}, \hat{s_2}, \ldots$ will be reported. However, in the concrete model, such a behavior cannot be found. So, this is not a real counterexample. □

4.2. Spurious counterexamples

As pointed in [5,6], a counterexample in the abstract model which does not exist in the concrete model is called a spurious counterexample. To formally define a spurious counterexample, we first introduce failure states. To this end, $In^0_{\hat{s_i}}, In^1_{\hat{s_i}}, \ldots, In^n_{\hat{s_i}}$ and $In_{\hat{s_i}}$ are defined first:

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

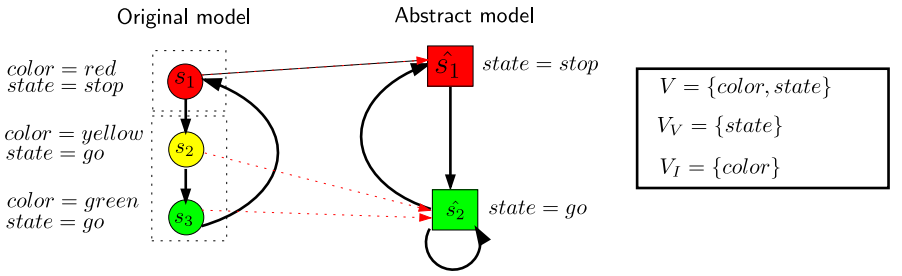


Fig. 2. Traffic light controller.

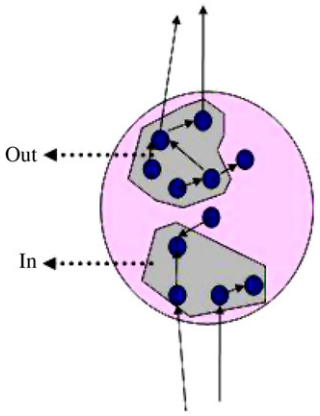


Fig. 3. $In_{\hat{s}_i}$ and $Out_{\hat{s}_i}$.

Clearly, $In^0_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ with input 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 input 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 input 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 gray part in Fig. 3. 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. Similarly, $Out^0_{\hat{s}_i}, Out^1_{\hat{s}_i}, \ldots, Out^n_{\hat{s}_i}$ and $Out_{\hat{s}_i}$ can also be defined.

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

where $Out^0_{\hat{s}_i}$ denotes the set of states in $h^-(\hat{s}_i, V_V)$ with output 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 output 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 output 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 gray part in Fig. 3. Similar to $In_{\hat{s}_i}$, there must exist a natural number n, such that $\bigcup_{i=0}^{n+1} Out^i_{\hat{s}_i} = \bigcup_{i=0}^{n} Out^i_{\hat{s}_i}$. Accordingly, a failure state can be defined as follows.

Definition 1 (*Failure States*)**.** A state $\hat{s}_i$ in a counterexample $\hat{IT}$ is a failure state if $In_{\hat{s}_i} \neq \emptyset$, $Out_{\hat{s}_i} \neq \emptyset$ and $In_{\hat{s}_i} \cap Out_{\hat{s}_i} = \emptyset$. □

Note that for the first state $\hat{s_1}$ of a counterexample, $In_{\hat{s_1}} = \emptyset$ and $Out_{\hat{s}_1} \neq \emptyset$; for the last state $\hat{s}_l$ of a finite counterexample, $In_{\hat{s}_l} \neq \emptyset$ and $Out_{\hat{s}_l} = \emptyset$; and for any other state $\hat{s}_i$ in a counterexample, $In_{\hat{s}_i} \neq \emptyset$ and $Out_{\hat{s}_i} \neq \emptyset$. So the first state of a counterexample and the last state of a finite counterexample will never be a failure state.

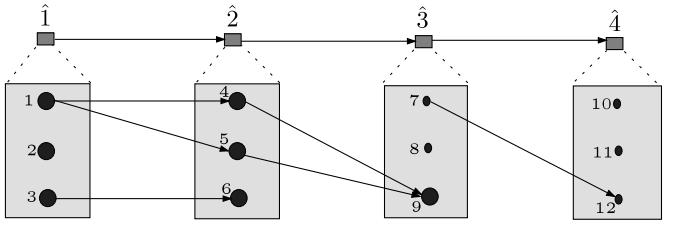


Fig. 4. A spurious path.

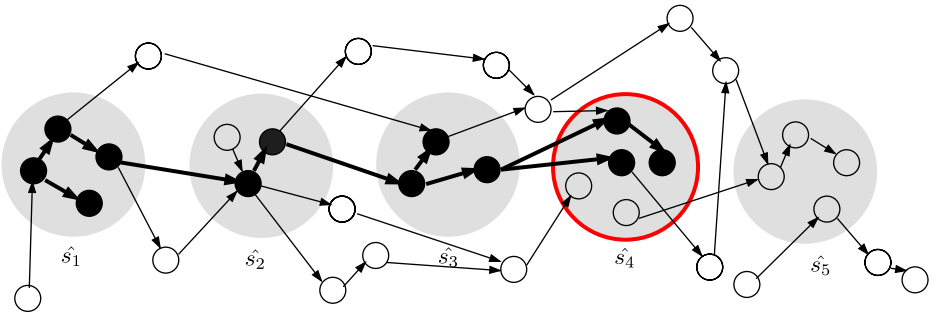


Fig. 5. Algorithm SplitPath.

Further, given a failure state $\hat{s}_i$ in a counterexample $\hat{\Pi}$, the set of the origins of $\hat{s}_i$, $h^-(\hat{s}_i, V_V)$, is separated into three sets, $\mathcal{D} = In_{\hat{s}_i}$ (the set of dead states), $\mathcal{B} = Out_{\hat{s}_i}$ (the set of bad states) and $\mathcal{I} = h^-(\hat{s}_i) \setminus (\mathcal{D} \cup \mathcal{B})$ (the set of the isolated states). Note that by the definition of a failure state, $\mathcal{D}$ and $\mathcal{B}$ cannot be empty sets, while $\mathcal{I}$ may be empty.

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 3. Fig. 4 shows a spurious counterexample where state $\hat{3}$ is a failure state. In the set, $h^-(\hat{3}, V_V) = \{7, 8, 9\}$, of the origins of state $\hat{3}$, 9 is a dead state, 7 is a bad state, and 8 is an isolated state. □

In [3], Algorithm SplitPath is presented for checking whether or not a counterexample is spurious, and a SAT solver is employed to implement it [5]. To compare our algorithms with Algorithm SplitPath, we briefly present the basic idea of SplitPath. As illustrate in Fig. 5, in SplitPath, reachable states from the states in $h^-(\hat{s_1}, V_V)$ are computed first; then for the ones that fall into $h^-(\hat{s_2}, V_V)$, the reachable states in $h^-(\hat{s_3}, V_V)$ are computed continuously, and so on. If no reachable states fall into $h^-(\hat{s_i}, V_V)$, the previous state, s_{i-1}, is a failure state. For instance, in Fig. 5, no reachable states fall into $h^-(\hat{s_5}, V_V)$. So, $\hat{s_4}$ is a failure state. For a finite counterexample, at most, all the states in the counterexample are checked. However, to check a periodic infinite counterexample, several repetitions of the periodic parts are needed. This will be extremely difficult in software model checking, since the state space of software are often tremendously large. Note that with Algorithm SplitPath, the first failure state in a spurious counterexample is always detected.

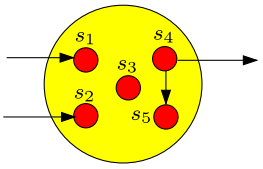
Based on the formal definition of a failure state, a new algorithm, named CheckSpurious, for checking whether or not a counterexample is spurious is proposed. Algorithm CheckSpurious takes a counterexample as input and outputs the first failure state as well as $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{I}$ with respect to the failure state by checking whether or not $In_{\hat{s}_i} \cap Out_{\hat{s}_i} = \emptyset$ for each state $\hat{s}_i$ in the counterexample. Note that a counterexample may be a finite path $\langle s_1, s_2, \ldots, s_n \rangle$, $n \geq 1$, or an infinite path $\langle s_1, s_2, \ldots, (s_i, \ldots, s_j)^\omega \rangle$, $1 \leq i \leq j$, with a loop suffix (a suffix produced by a loop). For the finite counterexample, it will be checked directly while for an infinite one, we need only to check its Complete Finite Prefix (CFP) $\langle s_1, s_2, \ldots, s_i, \ldots, s_j, s_i \rangle$.

Algorithm 2 : CheckSpurious($\hat{\Pi}$)

Input: a counterexample $\hat{\Pi} = \langle \hat{s_1}, \hat{s_2}, \ldots, \hat{s_n} \rangle$ 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: a failure state s_f, $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{I}$

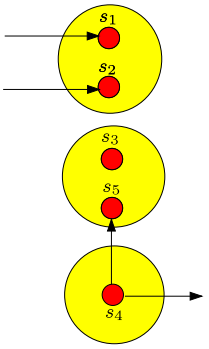
1: **Initialization**: *int* $i = 2$;
2: **while** $i \leq n - 1$ **do**
3: **if** $In_{\hat{s}_i} \cap Out_{\hat{s}_i} \neq \emptyset$, $i = i + 1$;
4: **else** return $s_f = \hat{s}_i$, $\mathcal{D} = In_{\hat{s}_i}$, $\mathcal{B} = Out_{\hat{s}_i}$, and $\mathcal{I} = h^-(\hat{s}_i) \setminus (\mathcal{B} \cup \mathcal{D})$; break;
5: **end while**
6: **if** $i = n$, return $\hat{\Pi}$ is a real counterexample;



	V_{v1}	V_{v2}	V_I	B
s_1	3	4		0
s_2	3	4		0
s_3	3	4		$\perp$
s_4	3	4		1
s_5	3	4		$\perp$
s_i	x	y		$\perp$

$i \neq 1, 2, 3, 4, 5$
$x = 3,\ y \neq 4$
$x \neq 3,\ y = 4$
$x \neq 3,\ y \neq 4$

Fig. 6. A failure state.



	V_{v1}	V_{v2}	B	V_I
s_1	3	4	0	
s_2	3	4	0	
s_3	3	4	$\perp$	
s_4	3	4	1	
s_5	3	4	$\perp$	
s_i	x	y	$\perp$	

$i \neq 1, 2, 3, 4, 5$
$x = 3,\ y \neq 4$
$x \neq 3,\ y = 4$
$x \neq 3,\ y \neq 4$

Fig. 7. Refined abstract states.

With Algorithm CheckSpurious, to check whether or not a state $\hat{s}_i$ is a failure state, it only relies on its pre- and post-states, $\hat{s_{i-1}}$ and $\hat{s_{i+1}}$; while in Algorithm SplitPath, to check state $\hat{s}_i$, it relies on all states in the prefix, $\hat{s_1}, \ldots, \hat{s_{i-1}}$, of $\hat{s}_i$. Based on this, to check a periodic infinite counterexample, several repetitions of the periodic parts are needed. In contrast, this can be easily done by checking the complete finite prefix $\langle s_1, s_2, \ldots, s_i, \ldots, s_j, s_i \rangle$ by Algorithm CheckSpurious.

4.3. Refining algorithm

When a failure state and the corresponding $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{I}$ are reported by Algorithm CheckSpurious, we need to further refine the abstract model such that $\mathcal{D}$ and $\mathcal{B}$ are separated into different abstract states. This can be achieved by making a set of invisible variables, $U \subseteq V_I$, visible [5]. With this method, to find the coarsest refined model is NP-hard. Further, to keep the size of the refined abstract state space smaller, it is important to make U as small as possible. However, to find the smallest U is also NP-hard [6]. In [5], an ILP solver is used to obtain the minimal set. It is inefficient when the problem size is large, since IPL is an NPC problem. To combat this, several approximate polynomial algorithms were proposed [5,8,9] with non-optimal results. Moreover, even though a coarser refined abstract model may be produced by making U smaller, it is uncertain that the smallest U will induce the coarsest refined abstract model. Motivated by this, a new refinement approach is proposed by adding extra boolean variables to the set of visible variables. With this approach, the NP-hard problem can be turned away, and a coarser refined abstract model can be also obtained. The basic idea for the refining algorithm is described below.

Assume that a failure state is found with $\mathcal{D} = \{s_1, s_2\}$, $\mathcal{B} = \{s_4\}$ and $\mathcal{I} = \{s_3, s_5\}$ as illustrated in Fig. 6 where the abstract model is obtained by making V_{v1} and V_{v2} visible and other variables invisible. To make $\mathcal{D}$ and $\mathcal{B}$ separated into two abstract states, an extra boolean variable B is added to the system with the valuation being 0 at the states in $\mathcal{D}$, 1 at the state in $\mathcal{B}$, and $\perp$ at the states in $\mathcal{I}$ and other states. That is $s_1(B) = 0, s_2(B) = 0, s_4(B) = 1$, and $s_i(B) = \perp$ where $s_i \in S$ and $i \neq 1, 2$, or 4. Subsequently, by making $V_V' = V_V \cup \{B\}$ and $V_I' = V_I$, the failure state is separated into three states in the refined abstract model as illustrated in Fig. 7. Note that, only the failure state is separated into three states, and other states are the same as in the abstract model. Especially, when $\mathcal{I} = \emptyset$, the failure state is separated into two new states.

Therefore, given a failure state s_i (as well as $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{I}$) in the abstract model $K = (S, S_0, R, L)$ where $S \subseteq \Sigma = \Sigma_{v_1} \times \cdots \times \Sigma_{v_n}$ and $V_V = \{v_1, \ldots, v_n\}$, to obtain the abstract model $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$, a boolean variable B is added as a visible variable with $s(B) = 0$ if $s \in \mathcal{D}$, $s(B) = 1$ if $s \in \mathcal{B}$, and $s(B) = \perp$ if $s \notin (\mathcal{D} \cup \mathcal{B})$. Thus, the set of all possible states in the refined abstract model will be $\hat{\Sigma} = \Sigma \times \Sigma_B$, where $\Sigma_B = \{B = d \mid d \in \{0, 1, \perp\}\}$. Accordingly, the refined abstract model $\hat{K} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ can be obtained by Algorithm Refine.

Complexity analysis. It can be observed that the new refinement algorithm is linear to the size of the state space, since it only needs to assign a value to the new added boolean variable at each state. Further, in each iteration, at most two more states are added (only one node is added when $\mathcal{I}$ is empty). Recall that in the approach by choosing a set of invisible variables to be visible again, to find the set which leads to a minimal refinement is NP hard. Further, even though such a set of variables is

Algorithm 3 : REFINE(K, $\mathcal{D}$, $\mathcal{B}$, $\mathcal{I}$, B)

Input: the abstract model $K = (S, S_0, R, L)$ with V_V being visible; $\mathcal{D}$, $\mathcal{B}$ and $\mathcal{I}$ reported by Algorithm CHECKSPURIOUS; the new boolean variable B which will be added

Output: the refined model $\hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$

1: $s(B) = 0$ if $s \in \mathcal{B}$; $s(B) = 1$ if $s \in \mathcal{D}$; $s(B) = \perp$ if $s \notin \mathcal{D} \cup \mathcal{B}$;
2: $\hat{S} = \{\hat{s} \in \hat{\Sigma} \mid$ there exists $s \in S$ such that $h(s, V_V \cup B) = \hat{s}\}$;
3: $\hat{S}_0 = \{\hat{s} \in \hat{S} \mid$ there exists $s \in S_0$ such that $h(s, V_V \cup B) = \hat{s}\}$;
4: $\hat{R} = \{(\hat{s_1}, \hat{s_2}) \mid \hat{s_1}, \hat{s_2} \in \hat{S}$, and there exist $s_1, s_2 \in S$ such that $h(s_1, V_V \cup B) = \hat{s_1}$, $h(s_2, V_V \cup B) = \hat{s_2}$ and $(s_1, s_2) \in R\}$;
5: $L(\hat{s}) = \bigcup_{s \in S, h(s, V_V \cup B) = \hat{s}} L(s)$;
6: return $\hat{K} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$;

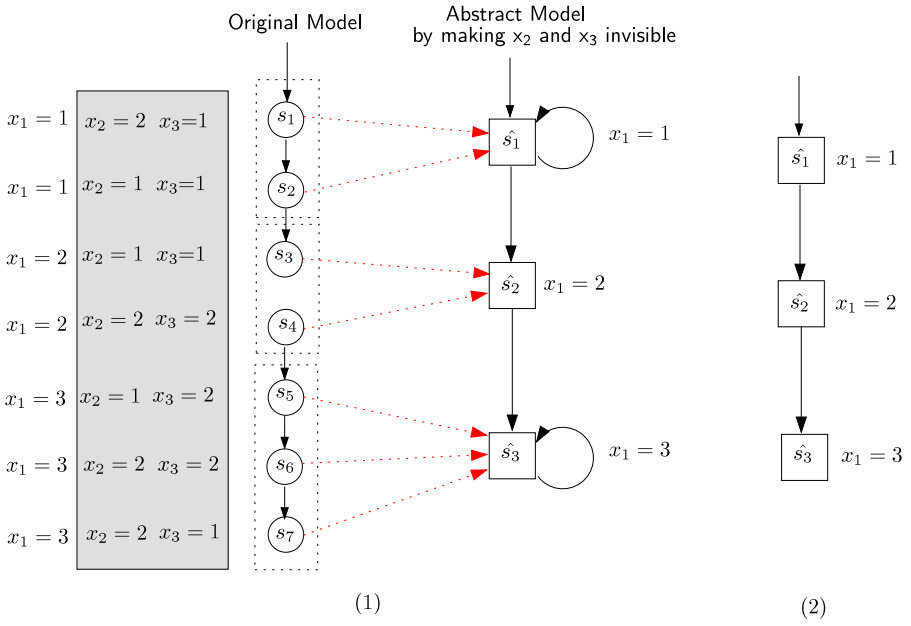


Fig. 8. Abstraction by making x_2 and x_3 invisible.

found, the refined model may be unbelievably large since lots of non-failure nodes (usually a huge number in the real systems in practice) will also be separated. In our approach by adding new boolean variables, not only is the NP hard problem for finding the minimal set of variables avoided, but also a coarser refined model is obtained.

To illustrate the intrinsic property of the new refining algorithm, a simple example is given below.

Example 4. The Kripke structure illustrated in the l.h.s of Fig. 8(1) presents an original model where three variables x_1, x_2 and x_3 are involved. Assume that x_2 and x_3 are insensitive to the property which is expressed in a temporal logic formula. Thus, by making x_2 and x_3 invisible, the abstract model can be obtained by Algorithm ABSTRACT as illustrated in the r.h.s of Fig. 8(1).

Suppose that a counterexample is found by a model checker as depicted in Fig. 8(2). Then, by Algorithm CHECKSPURIOUS, it will report that $\hat{s_2}$ is a failure state, and $\mathcal{D} = \{s_3\}$, $\mathcal{B} = \{s_4\}$. First, we show the refined abstract models by the method in the related works [5,12,8,9]. The refined abstract model obtained by making x_2 and x_3 visible are illustrated in Figs. 9 and 10, respectively.

It can be observed that the one by making x_3 visible is the smallest refined model under the method by making some invisible variables visible. Clearly, to find the coarsest refined model, in this way, is an NP-hard problem.

By our method, as depicted in Fig. 11, a new boolean variable B is added to the system and made visible. Then the refined abstract model is obtained where only the failure state is separated into two states with other states unchanged. Clearly, the new refining algorithm avoids the NP-hard problem for finding the smallest set of visible variables. Moreover, the new refined abstract model is smaller than the best result produced in the method by further making some invisible variables visible. □

Clearly, the refined model obtained by Algorithm REFINE is not the smallest one. And the smallest refined abstract model can be easily obtained by assigning the new added variable B by 0 or 1 at the states in $\mathcal{I}$, i.e. the failure state is separated into $\mathcal{D} \cup \mathcal{I}$ and $\mathcal{B}$, or $\mathcal{D}$ and $\mathcal{B} \cup \mathcal{I}$. This is intuitively presented in Fig. 12. Compared to Algorithm REFINE, only one state is saved

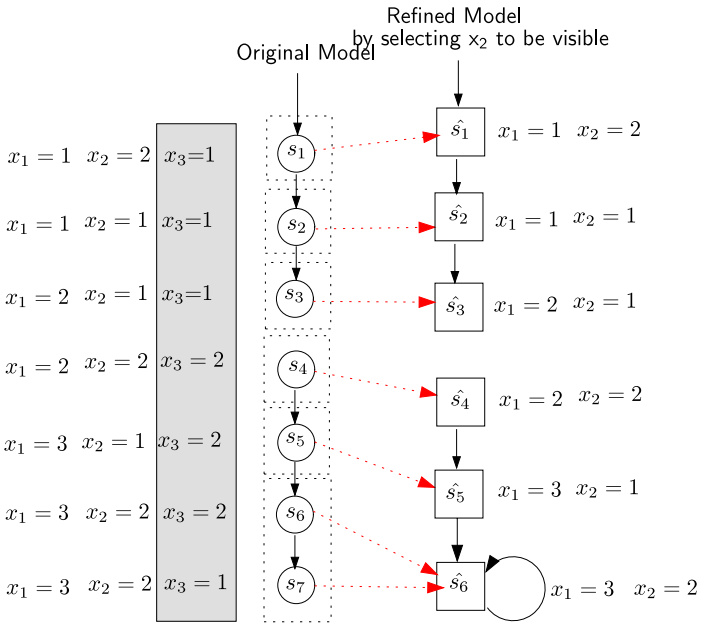


Fig. 9. Refinement by the old algorithm.

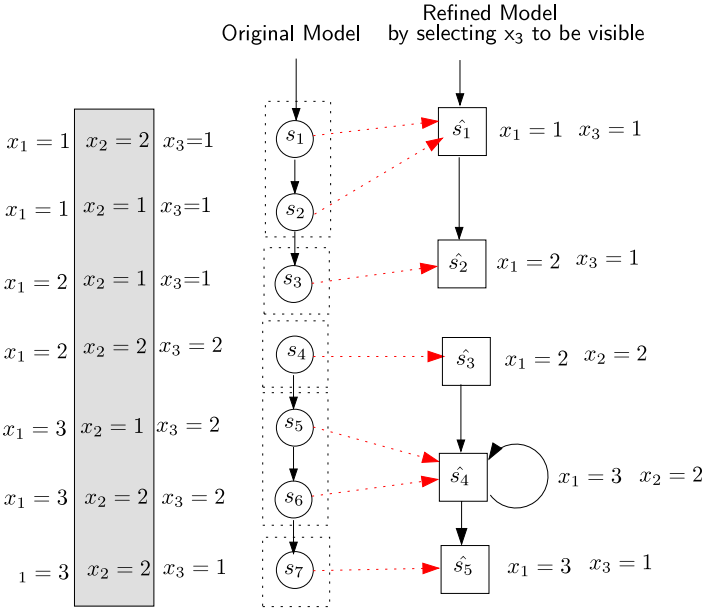


Fig. 10. Refinement by the old algorithm.

in the refinement. However, possibly, more iterations will be introduced into the abstract-refinement loop since $\mathcal{D} \cup \mathcal{I}$ or $\mathcal{B} \cup \mathcal{I}$ may further be found as a failure state.

5. Abstract-refinement loop

With the new proposed algorithms, the abstract model checking framework is presented. First, the abstract model is obtained by Algorithm Abstract. 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. However, if a counterexample is reported, it is checked by Algorithms CheckSpurious. If the counterexample is not spurious, it will be a real counterexample that violates the system; otherwise, the counterexample is spurious, and Algorithm Refine is used to refine the abstract model by adding a new visible boolean variable B to the system. Then 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. This process is formally described in Algorithm AbstractMC

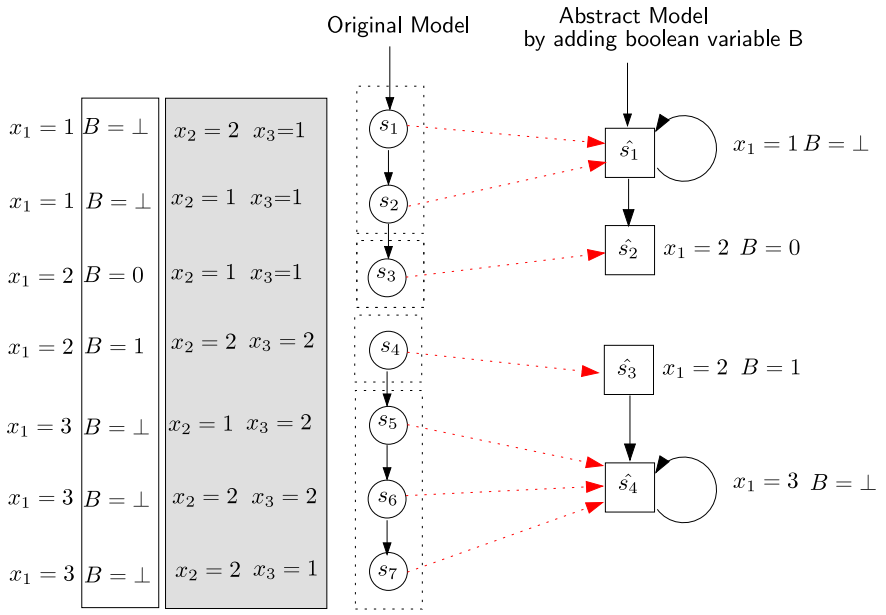


Fig. 11. Refinement by the new algorithm.

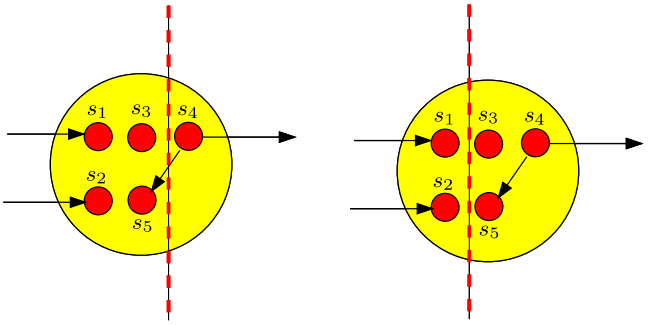


Fig. 12. Smallest refinement.

Algorithm 4 : AbstractMC

Input: A model $K = (S, S_0, R, L)$ in Kripke structure, and a desired property ϕ in temporal logic
Output: a counterexample that violates ϕ

1: **Initialization**: *int* $i = 1$;
2: $\hat{K}$ =Abstract(K, V_I);
3: $MC(\hat{K}, \phi)$;
4: **while** a counterexample $\hat{\Pi}$ is found **do**
5: CheckSpurious($\hat{\Pi}$);
6: **if** $\hat{\Pi}$ is a real counterexample, return $\hat{\Pi}$; break;
7: **else** $\hat{K}$ =Refine($\hat{K}$, $\mathcal{D}$, $\mathcal{B}$, $\mathcal{I}$, B_i); $i = i + 1$; $MC(\hat{K}, \phi)$;
8: **end while**
9: **if** no counterexample is found, K satisfies ϕ.

where a subscript i is used to identify different boolean variables that are added to the system in each refinement process. Initially, i is assigned by 1. After each iteration of Algorithm Refine, i is increased by 1. Basically, finitely many boolean variables will be added since the systems to be verified with model checking are finite systems.

Termination analyzing. We can confirm the termination of the new abstraction-refinement loop. Extremely, all the nodes in the abstract model are separated and the original model is obtained again. In this case, at most n boolean variables are added to the model, and then the abstraction-refinement loop ends.

6. Conclusion

An efficient approach for abstraction refinement is given in this paper. With this approach, (1) whether or not a counterexample is spurious can be checked easily; (2) the NP-hard state separation problem is avoided; (3) a smaller refined abstract model is also obtained. This can improve the abstract based model checking, especially the counterexample guided abstraction refinement model checking. In the near future, the proposed algorithms will be implemented and integrated into the tool CEGAR. Further, some case studies will be conducted to evaluate the proposed approach.

References

[1] E.M. Clarke, E.A. Emerson, Design and synthesis of of synchronization skeletons using branching time temporal logic, in: Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, in: LNCS, vol. 131, Springer, 1981.
[2] J.P. Quielle, J. Sifakis, Specification and verification of concurrent systems in CESAR, in: Proceedings of the 5th international symposium on programming, 1981, pp. 337–350.
[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample guided abstraction refinement, in: E. Emerson, A. Sistla (Eds.), Proc. 12th Int. Conf. Computer-Aided Verification, CAV00, vol. 1855, New York, 2000.
[4] S.A. Kripke, Semantical analysis of modal logic I: normal propositional calculi, Z. Math. Logik Grund. Math. 9 (1963) 67–96.
[5] E.M. Clarke, A. Gupta, O. Strichman, SAT based counterexample-guided abstraction-refinement, IEEE Transactions on Computer Aided Design 23 (7) (2004) 1113–1123.
[6] Fei He, Xiaoyu Song, William N.N. Hung, Ming Gu, Jiaguang Sun, Integrating evolutionary computation with abstraction refinement for model checking, IEEE Transactions on Computers 59 (1) (2010) 116–126.
[7] 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.
[8] Fei He, Xiaoyu Song, Ming Gu, Jia-Guang Sun, Heuristic-guided abstraction refinement, Comput. J. 52 (3) (2009) 280–287.
[9] Fei He, Xiaoyu Song, William N.N. Hung, Ming Gu, Jiaguang Sun, Integrating evolutionary computation with abstraction refinement for model checking, IEEE Transactions on Computers 59 (1) (2010) 116–126.
[10] R.P. Kurshan, Computer Aided Verification of Coordinating Processes, Princeton Univ. Press, 1994.
[11] C. Wang, B. Li, H. Jin, G.D. Hachtel, F. Somenzi, Improving Ariadne's bundle by following multiple threads in abstraction refinement, IEEE Transactions on Computer Aided Design 25 (11) (2006) 2297–2316.
[12] E.M. Clarke, A. Gupta, J.H. Kukula, O. Strichman, SAT based abstraction-refinement using ILP and machine learning techniques, in: E. Brinksma, K.G. Larsen (Eds.), Proc. Computer-Aided Verification, CAV, 2002, pp. 265–279.
[13] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, D. Wang, Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis, in: Proc. Formal Methods in Computer-Aided Design, FMCAD, 2002.
[14] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proc. Symp. Principles of Programming Languages, 2002, pp. 58–70.
[15] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, M.Y. Vardi, Multiple-counterexample guided iterative abstraction refinement: an industrial evaluation, in: Proc. Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2003, pp. 176–191.
[16] S.G. Govindaraju, D.L. Dill, Counterexample-guided choice of projections in approximate symbolic model checking, in: Proc. Intl Conf. Computer-Aided Design, ICCAD, 2000, pp. 115–119.