Huang SB, Huang HT, Chen ZY *et al.* Lazy slicing for state-space exploration. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(4): 872–890 July 2012. DOI 10.1007/s11390-012-1271-7

Lazy Slicing for State-Space Exploration

Shao-Bin Huang (黄少滨), *Senior Member*, *CCF*, Hong-Tao Huang (黄宏涛), Zhi-Yuan Chen (陈志远),
Tian-Yang Lv (吕天阳), *Member*, *CCF*, and Tao Zhang (张　涛)

College of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China

E-mail: huangshaobin@hrbeu.edu.cn; horntau@gmail.com; chenzhiyuan@hrbeu.edu.cn;
{raynor1979, zhangtaohrbeu}@163.com

Received April 23, 2011; revised February 23, 2012.

Abstract CEGAR (Counterexample-guided abstraction refinement)-based slicing is one of the most important techniques in reducing the state space in model checking. However, CEGAR-based slicing repeatedly explores the state space handled previously in case a spurious counterexample is found. Inspired by lazy abstraction, we introduce the concept of lazy slicing which eliminates this repeated computation. Lazy slicing is done on-the-fly, and only up to the precision necessary to rule out spurious counterexamples. It identifies a spurious counterexample by concretizing a path fragment other than the full path, which reduces the cost of spurious counterexample decision significantly. Besides, we present an improved over-approximate slicing method to build a more precise slice model. We also provide the proof of the correctness and the termination of lazy slicing, and implement a prototype model checker to verify safety property. Experimental results show that lazy slicing scales to larger systems than CEGAR-based slicing methods.

Keywords counterexample-guided abstraction refinement, spurious counterexample, over-approximate slicing, local refinement, lazy slicing

1 Introduction

With the rapid development of model checking technology[1] over the past two decades, various model checkers have been widely used in industry to automatically analyze finite state concurrent systems. However, the linear growth of the number of variables and concurrent execution components in software systems will lead to an exponential growth of state space, which is the major challenge in software model checking[2]. Program slicing[3] is one of the effective methods to alleviate state space explosion, and can eliminate the irrelevant portion of a software specification by reachability analysis. This technology has been successfully used to reduce the state space in model checking[4-7].

The next generation model checking framework proposed in [8], is capable of reducing the concurrent object-oriented source code significantly with the help of its Java program slicing component. [8] also indicates that slicing concurrent object-oriented source code provides significant reductions that are orthogonal to a number of other well-known model reduction techniques (such as partial order reduction[9] and symmetry reduction[10-11]), and that slicing should always be applied due to its automation and low computational costs. As a matter of fact, there have been lots of studies using the orthogonality between slicing and other reduction techniques to enhance the power of slicing. For example, [12] and [13] make slicing work together with data abstract to minimize the four-variable model. Slicing is also used in [14] and [15] to remove irrelevant states and transitions from the abstract model, which makes slicing and predicate abstract complement each other effectively.

Conventional wisdom holds that static program slicing can be an effective model reduction technique for software model checking[8]. However, existing experience with slicing for model reduction is sometimes inconclusive. Holzmann's experience shows that slicing in Spin usually does not yield much reduction for realistic Promela design models[16]. The main reason is that the compression capability of static slicing depends on not only the slicing criterion but also the dependency relationship between variables. That is why it does not always guarantee a slice model with a desired size.

In order to overcome this obstacle, incremental slicing[18], a typical CEGAR (counterexample-guided abstraction refinement)-based slicing method[17], first

---

Regular Paper

Supported by the National Natural Science Foundation of China under Grant No. 60873038, the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under Grant Nos. 2009BAH42B02 and 2012BAH08B02.

constructs an over-approximate slice model of original software specification, then continuously increases the precision of the over-approximate slice to meet the verification demand until the desired property is proved correct or a real counterexample is discovered. Compared with static slicing, incremental slicing scales to a remarkably larger state space. A variant of incremental slicing known as stepwise slicing was also proposed by [19], and the difference is that stepwise slicing verifies a given property on the dependence graph of program behavior model rather than a state machine. In addition, a bounded slicing algorithm decreases the iteration number of refinement iteration, thus reduces the appearance probability of spurious counterexamples in stepwise slicing. But on the other hand, stepwise slicing cannot be finished automatically, because it needs manual intervention to remove irrelevant variables in each refinement iteration.

Both incremental slicing and stepwise slicing are proposed in order to perform the verification at a minimum cost. However, both of these two methods restart a verification on the entire refined over-approximate slice, and the work done on earlier slices is completely ignored. This leads to the unnecessary repeated computation.

We believe that the slicing-verification-refinement iteration can be performed on one slice with different precisions by local refinement, which avoids repetitive verifications by preserving previous achievements. Our method, namely lazy slicing, which is inspired by the idea of lazy abstraction[20], can scale to large systems with an additional optimization on CEGAR-based slicing. It refines and then verifies only the unexplored portion of the slice if a spurious counterexample is identified, which improves the performance significantly by avoiding repetitively verifying the state space that has been proven correct. Besides, lazy slicing produces paths with ascending precisions, which makes it possible to determine whether a counterexample is spurious or not by concretizing a path fragment other than the full path. Therefore, the efficiency of spurious counterexample decision can be improved significantly. Finally, we give an improved over-approximate slicing method, which is able to build a more precise slice than incremental slicing does[18].

This paper considers a simplified software model in order to illustrate our state space exploration method. We assume that all system models consist of finite domain variables, since variables with infinite domain can be converted to variables with finite domain with the help of data abstraction[12-13], predicate abstraction[21-24], etc. Because the result of the parallel composition of multiple Kripke models is still a single Kripke model[25], we do not consider it in this paper. Besides, the discussion on state space exploration method in this paper is limited to reachability. We hope to apply our method to verify LTL (Linear Temporal Logic) properties by converting LTL model checking problem to a reachability problem, this will be done in future work.

This paper is organized as follows. Section 2 describes four key steps of lazy slicing, i.e., over-approximate slicing, spurious counterexample decision, local slice refinement and exploration. We then demonstrate how lazy slicing works with a full example of mutual exclusion algorithm. Sections 3 to 6 present the definitions, calculation processes and features of the four key steps in detail respectively. Section 7 presents an improved over-approximate slicing method which can produce a more precise slice than the method given in Section 3. Section 8 proves the correctness, termination and analyzes the savings of lazy slicing. Section 9 reports the experimental results which are consistent with the analysis in Section 8. Finally, Section 10 concludes the paper.

2 Main Steps and An Example

The repeated computation of CEGAR-based slicing can be eliminated by Lazy slicing. The main reason is lazy slicing refines and verifies only the remaining part of the state space, which saves the cost of unnecessary exploration on state space that is known to be correct. In this section, we will describe the main steps of lazy slicing with an example.

Intuitively, lazy slicing works as follows. In the refinement step, the dead end state suggests which variable should be added to refine the slice. (The dead end state is a state on the spurious counterexample, which is able to transit to its successor on the spurious counterexample, but this transition cannot happen on the original model.) Instead of building and verifying on an entire new slice, we refine only the state space which has not been verified, then restart a verification on the refined local slice with a higher precision. It means the desired property can be validated without revisiting the state space handled previously. Lazy slicing repeats the work until the desired property is established or a counterexample is found. If it terminates with outcome that the model satisfies the desired property, the proof is a slice whose precision changes in different parts; while if it terminates with the outcome that the model violates the given property, the proof is a counterexample with ascending precisions.

We will use a simple mutual exclusion algorithm to demonstrate how lazy slicing works.

Example 1. Let $V = \{x, y, z\}$ be a variable set, D_x, D_y, D_z are the domain of x, y, z respectively, where

$D_x = D_y = \{n, t, c\}$, and $D_z = \{0, 1\}$. x and y denote two processes, n, t, c are the possible values of x and y. $x = n, t, c$ (or $y = n, t, c$) denotes process x (or y) is not in its critical section, is trying to enter its critical section and is already in its critical section respectively. And variable z determines which process can enter the critical section when both x and y are equal to t. $z = 0$ denotes x has the permission, while $z = 1$ denotes y has the permission. The event set which determines how the algorithm runs is given in Table 1.

Table 1. Event Set of Process Mutual Exclusion Algorithm

No.	Guard	Assignment
e_1	$x = n$	$x = t$
e_2	$x = t \wedge y = n$	$x = c$
e_3	$x = t \wedge y = t \wedge z = 0$	$x = c$
e_4	$x = c \wedge z = 0$	$z = 1$
e_5	$x = c$	$x = n$
e_6	$y = n$	$y = t$
e_7	$y = t \wedge x = n$	$y = c$
e_8	$y = t \wedge x = t \wedge z = 1$	$y = c$
e_9	$y = c \wedge z = 1$	$z = 0$
e_{10}	$y = c$	$y = n$

The function of an event is to execute assignments when the guard is satisfied. For example, if the guard of e_1 holds, i.e., $x = n$ is true, then e_1 assigns t to x. Let $x = y = n$ initially, then Fig.1 shows the state transition system of process mutual exclusion algorithm.

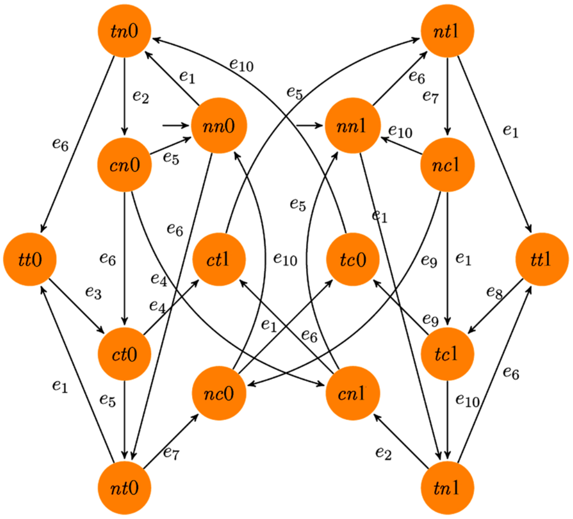


Fig.1. State transitions system of process mutual exclusion algorithm.

If $\varphi_1 = \neg(x = c \wedge y = c)$ is a desired property, then lazy slicing runs as follows.

Step 1 (Corresponding to Section 3). *Calculate an over-approximate slice with regard to Fig.*1 *and* φ_1.

Because φ_1 contains only x and y, the assignments of e_1, e_2, e_3, e_5, e_6, e_7, e_8, e_{10} introduce none variable other than x and y, so the over-approximate slice is related to a variable set containing x and y, and an event set (see Table 2) containing e_1, e_2, e_3, e_5, e_6, e_7, e_8 and e_{10}. Besides, the guards of e_3 and e_8 contain z (not in $\{x, y\}$) which causes the guards of e_3 and e_8 are set to true.

Table 2. Event Set of the Over-Approximate Slice with Regard to Fig.1 and φ_1

No.	Guard	Assignment
e_1	$x = n$	$x = t$
e_2	$x = t \wedge y = n$	$x = c$
e_3	true	$x = c$
e_5	$x = c$	$x = n$
e_6	$y = n$	$y = t$
e_7	$y = t \wedge x = n$	$y = c$
e_8	true	$y = c$
e_{10}	$y = c$	$y = n$

So the over-approximate slice with regard to Fig.1 and φ_1 is depicted in Fig.2.

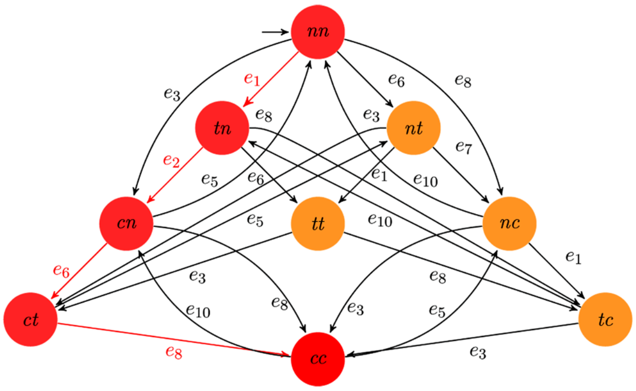


Fig.2. Over-approximate slice of Fig.1 with regard to φ_1.

Note that in this example, we calculate over-approximate slice according to the method given in [18]. Its purpose is to introduce a spurious counterexample (Fig.2, the path colored red). We will give an improved method which will build a more precise over-approximate slice than Fig.2 under the same conditions in Section 7.

Step 2 (Corresponding to Section 4). *Spurious counterexample decision.*

Assume lazy slicing searches (DFS) along path $\pi = s_{nn} s_{tn} s_{\rm cn} s_{\rm ct} s_{\rm cc}$ (Fig.2, the path colored red), let R denote the states which have already been checked. As s_{cc} does not satisfy φ_1, so π is a counterexample on Fig.2, and at this time $R = \{s_{nn}, s_{tn}, s_{cn}, s_{ct}, s_{cc}\}$. If there does not exist a corresponding path of π on Fig.1, then π is not a real counterexample, i.e., a spurious counterexample. In order to determine whether π is a spurious counterexample, lazy slicing tries to find a corresponding path of π on Fig.1. While we can only find two path fragments $\pi_1 = s_{nn0} s_{tn0} s_{cn0} s_{ct0}$

and $\pi_2 = s_{nn1}s_{tn1}s_{cn1}s_{ct1}$ corresponding to path fragment $s_{nn}s_{tn}s_{cn}s_{ct}$ (see Fig.3, where S_1 is the feasible or reachable equivalent state set of S_{nn} and the calculation of S_i is defined in Proposition 3 in Section 4).

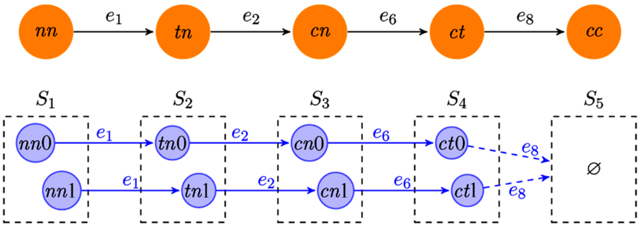


Fig.3. Spurious counterexample decision of π.

We can conclude from Fig.3 that there is no path corresponding to π in Fig.1, i.e., π is a spurious counterexample and s_{ct} is the dead-end state. We call $s_{nn}s_{tn}s_{cn}s_{ct}$ the feasible prefix of π due to π_1 and π_2. Then we remove s_{cc} from R because s_{cc} is infeasible, so we have $R = \{s_{nn}, s_{tn}, s_{cn}, s_{ct}\}$ now.

Note that $s_{nn0}s_{tn0}s_{cn0}s_{ct0}$ corresponds to $s_{nn}s_{tn}s_{cn}s_{ct}$ means that s_{nn0}, s_{tn0}, s_{cn0}, s_{ct0} are covered by s_{nn}, s_{tn}, s_{cn}, s_{ct} respectively, and the state transitions of $s_{nn0}s_{tn0}s_{cn0}s_{ct0}$, $s_{nn}s_{tn}s_{cn}s_{ct}$ are caused by the same event sequence $e_1e_2e_6$ (see Fig.3). Informally, if s_{nn} in Fig.2 satisfies a property φ implies s_{nn0} in Fig.1 satisfies φ too, then we say s_{nn} covers s_{nn0} or s_{nn0} is covered by s_{nn}.

Now that π is a spurious counterexample, so the next step is to determine which part of Fig.2 needs to be refined.

Step 3 (Corresponding to Section 5). *Refine a part of Fig.2 which has not been explored previously.*

Because neither s_{ct0} nor s_{ct1} can transit to a state in Fig.1 corresponding to s_{cc} via event e_8 (see Fig.3), the guard of e_8 suggests variable z (belongs to the guard of e_8) should be used to refine Fig.2.

According to Step 1, the precision of the refined over-approximate slice is $\{x, y, z\}$, so e_4 and e_9 are added to the event set of Fig.2, and the guards of e_3 and e_8 will not be set to true because the variables of their guards are contained in $\{x, y, z\}$, which makes the event set of the refined over-approximate slice the same as Table 1. Finally, we get a refined over-approximate slice which is identical to the original model (i.e., Fig.1).

The key difference between lazy slicing and CEGAR-based slicing is that lazy slicing explores state space from the successors of the feasible prefix of the spurious counterexample π instead of exploring the entire refined over-approximate slice (see Fig.4).

Fig.4 shows the feasible equivalent states (colored blue) and the feasible equivalent successors (colored green and red) of each state on the feasible prefix of π. $S^{fe}(s_{nn})$ $S^{fe}(s_{tn})$, $S^{fe}(s_{cn})$, $S^{fe}(s_{ct})$ denote the feasible equivalent state sets of s_{nn}, s_{tn}, s_{cn}, s_{ct} respectively. $Posts(S^{fe}(s_{nn}))$ denotes the successor of $S^{fe}(s_{nn})$, which is called the feasible equivalent successor set of s_{nn}. In the following parts, we use $Post_e(s)$ to represent the single successor of a given state s via a given event e. Let take s_{ct} as an example, s_{ct0} and s_{ct1} are the feasible equivalent states of s_{ct}, because they satisfy the following two conditions: firstly, they are on the corresponding paths of the feasible prefix of π (see Fig.3); secondly, they are covered by s_{ct}. s_{nt0} and s_{ct1} are successors of s_{ct0}, and s_{nt1} is the successor of s_{ct1}, so we call s_{nt0}, s_{ct1} and s_{nt1} the feasible equivalent successors of s_{ct} (see Fig.4). s_{ct1} is colored green because it is covered by s_{ct} which is in R, and there is no state in R covers s_{nt0} and s_{nt1}, thus they are colored red.

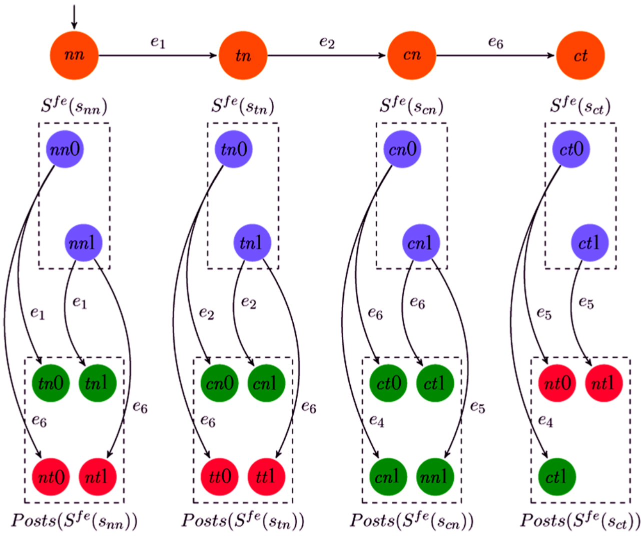


Fig.4. Feasible equivalent successors of the feasible prefix of π.

We take the feasible equivalent successors of the feasible prefix of π which are not covered by R (states colored red) as the initial states of the refined local over-approximate slice, and continue exploring the state space on the refined over-approximate slice from these states on.

Step 4 (Corresponding to Section 6). *Search state space lazily.*

Note that the feasible equivalent successors of the feasible prefix of π belong to the original Kripke model, so before exploring the remained state space, we have to project these states to the refined over-approximate slice. In this case, the refined over-approximate slice is the same as the original Kripke model, so we search the remained state space from the feasible equivalent successors directly.

We continue the depth first search from the feasible equivalent successors of s_{ct}. The blue states and transitions in Fig.5 show the search paths along s_{nt0}. The dashed arrow from s_{ct} to s_{nt0} means this transition is between states with different precisions. Then we traverse along path $s_{nt0}s_{nc0}s_{nn0}$, as s_{nn0} is covered by s_{nn}, so we take s_{nn} as the successor of s_{nc0}, and draw a dashed arrow from s_{nc0} to s_{nn}. Then we stop traversing along this path, and turn to deal with other successors of s_{nc0}. So we continue traversing along $s_{tc0}s_{tn0}$, and for the same reason, we stop exploring this path with drawing a dashed arrow from s_{tc0} to s_{tn}. As neither s_{tc0} nor s_{nc0} has a successor that has not been traversed, we turn to deal with s_{tt0}, the successor of s_{nt0}, and search along path $s_{tt0}s_{ct0}$, finally we stop exploring this path with drawing a dashed arrow from s_{tt0} to s_{ct}. The next step is to search along s_{nt1} which is the only successor of s_{ct} that has not been handled till now. The states and transitions colored red show the search results along s_{nt1} (see Fig.5). As to s_{ct1}, it is covered by s_{ct}. This means $s_{\rm ct}$ can transit to itself and this self loop is omitted.

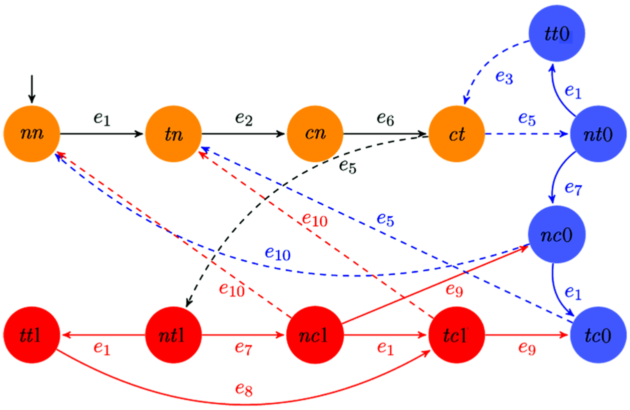


Fig.5. State space has been expanded by lazy slicing after the exploration along s_{ct} is finished.

Then lazy slicing turns to deal with s_{cn}. The blue transitions in Fig.6 depict the search paths along s_{cn}. As s_{ct0} and s_{ct1} are covered by s_{ct}, we should have drawn a solid arrow from s_{cn} to s_{ct}, but this arrow already exists. s_{cn1} is covered by s_{cn}, and this self loop is omitted. s_{nn1} is covered by s_{nn}, so we draw a blue solid arrow from s_{cn} to s_{nn}. s_{tn} and s_{nn} are handled in the same way. The red transitions and green transitions in Fig.6 show the search path along s_{tn} and s_{nn} respectively.

Lazy slicing terminates after all paths starting from s_{nn} have been explored with the result that Fig.1 satisfies φ_1. The final state space expanded by lazy slicing is shown as Fig.6.

Improvement. Intuitively, in this example, lazy slicing explores only 13 states to accomplish the verification (12 states on Fig.6, 1 state in the infeasible suffix of π_1, i.e., s_{cc}). While CEGAR-based slicing has to traverse 21 states to accomplish the same task (5 states in the spurious counterexample π_1, and 16 states in Fig.1).

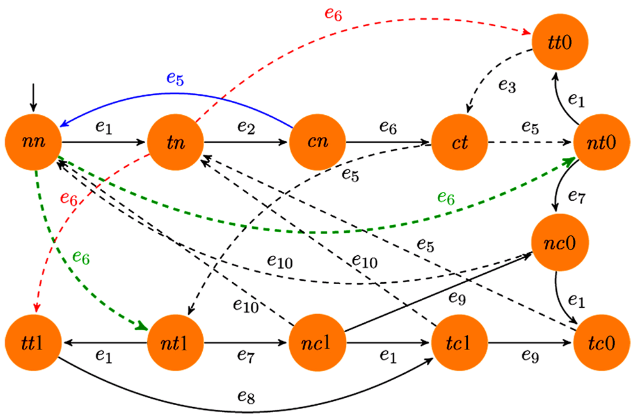


Fig.6. State space expanded by lazy slicing.

Furthermore, we do not need to draw the dashed arrows only for the purpose of verification. These arrows are designed to provide a comparison between the original model (Fig.1) and the over-approximate slice with multiple precisions (Fig.6). As a matter of fact, all the behaviours of Fig.1 can be found in Fig.6. This means that we can find a counterpart in Fig.6 for every state or transition in Fig.1. In the following sections, we make this intuitive algorithm precise.

3 Over-Approximate Slicing

We consider program models which are similar to the definition of [21].

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the variable set of a program, $D_1, D_2, \ldots, D_n$ be the domains of $v_1, v_2, \ldots, v_n$ respectively; let *Evt* denote the event set of a program, an event $e \in Evt$ is a conditional assignment defined on V:

$$guard(V_{Evt}) \mapsto \begin{cases} v_1 = Expr_1, \\ v_2 = Expr_2, \\ \vdots \\ v_m = Expr_m. \end{cases} \quad (1)$$

The left part of (1), i.e., $guard(V_{Evt})$, is called the guard of an event. It is a logic expression defined on V_{Evt}, where $V_{Evt} \subseteq V$. The right part of (1) is an assignment which consists of a set of assignment expressions with the form $v_1 = Expr_i$, where $v_i \in V$, and v_i is evaluated as the value of $Expr_i$. And $s'' = e(s')$ is a transition from s' to s'' caused by e, where $e \in Evt$. For convenience, we write $guard(e)$ to denote the guard of e, and $assign(e)$ the assignment of e. $AE(assign(e))$ is a set of assignment expressions of $assign(e)$, $target(ae)$ denotes the target variable of ae, where $ae \in AE(assign(e))$. A program can be defined as a Kripke structure directly.

Definition 1. *A Kripke structure is a tuple* $K = (S, T, I, L, AP)$ *where*

- $S \subseteq D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$ *is a set of states, where* $V = \{v_1, v_2, \ldots, v_n\}$,
- $T \subseteq S \times Evt \times S$ *is a set of transitions,*
- $I \subseteq S$ *is a set of initial states,*
- AP *is a set of atomic propositions, and*
- $L : S \rightarrow 2^{AP}$ *is a labelling function,*

where $I = d_{v_{i1}} \times d_{v_{i2}} \times \cdots \times d_{v_{im}} \times D_{v_{j1}} \times D_{v_{j2}} \times \cdots \times D_{v_{jm}}$, $d_{v_{i1}}, d_{v_{i2}}, \ldots, d_{v_{im}}$ are initial values of $v_{i1}, v_{i2}, \ldots, v_{im}$, and $\{v_{i1}, v_{i2}, \ldots, v_{im}\} = V \backslash \{v_{j1}, v_{j2}, \ldots, v_{jn}\}$. For instance, in Example 1, x and y are both evaluated as n initially, so $I = \{n\} \times \{n\} \times \{0, 1\} = \{s_{nn0}, s_{nn1}\}$. S can be defined recursively as follows: for any $s \in I$, we have $s \in S$; for any state $s \in S$, if s transmits to a state s' via $e \in Evt$, then $s' \in S$. For any two states $s_1, s_2 \in S$, if s_1 can transit to s_2 via $e \in Evt$, we have $(s_1, e, s_2) \in T$.

Definition 2. *A path is a state sequence* s_0, $s_1, \ldots, s_n$ *on a Kripke structure* K*, where* $s_0 \in I$*, and for every* s_i, s_{i+1}*, there is an* $e \in Evt$ *such that* $(s_i, e, s_{i+1}) \in T$, $0 \leqslant i < n$. *Paths*(K) *is the set containing all paths of* K.

Definition 3. *Let* $\pi = s_0, s_1, \ldots, s_n$ *be a path on a Kripke structure* K*, a sequence of the form* $L(s_0), L(s_1), \ldots, L(s_n)$ *is called the trace of* π*, denoted trace*(π). *The traces of Kripke structure* K *are thus words over the alphabet* 2^{AP}*, denoted as Traces*(K).

Definition 4. *A slicing criterion of* $K = (S, T, I, L, AP)$ *is a tuple* $C = (I, var(\varphi))$*, where* K *is a Kripke model,* I *is the initial state set of* K*,* φ *is the desired property,* $var(\varphi) \subseteq V$ *denotes the variables appearing in* φ.

Let $K^0 = (S^0, T^0, I^0, L^0, AP^0)$ be the over-approximate slice of K with respect to $C = (I, var(\varphi))$. There are two key steps in computing K^0. We first compute the variable set V^0 of K^0 by dependency analysis, then compute the event set Evt^0 of K^0 according to V^0. V^0 is computed as follows:

$$\begin{aligned} &V_0 = var(\varphi), \\ &V_{i+1} = V_i \bigcup_{\substack{ae \in AE(assign(e)) \wedge \\ target(ae) \in V_i \wedge e \in Evt}} var(ae). \end{aligned} \tag{2}$$

This computation terminates at some $i = k$, where $1 \leqslant k < |V|$. When the computation is finished, we have $V^0 = V_{k+1} = V_k$. Evt^0 is computed as follows.

$$Evt^0 = \bigcup_{\substack{e \in Evt \wedge \exists ae \in AE(assign(e)). \\ (target(ae) \in V^0 \rightarrow var(ae) \subseteq V^0)}} e. \tag{3}$$

For $\forall e \in Evt^0$, if $\exists ae \in assign(e).(target(ae) \notin V^0)$, then ae is removed from $assign(e)$.

In order to show the advantages of lazy slicing compared with CEGAR-based methods, we used the method of [18] to produce the event set of an over-approximate slice[18], which will lead to a much coarser slice. We will give an improved over-approximate slicing method to build a more precise model in Section 7. Note that, Evt^0 originates from Evt, an event e' in Evt^0 is just a different version of its corresponding event in Evt.

Definition 5. e' *is called an equivalent event of* e*, or* e *is called an equivalent event of* e' *iff* $e \in Evt$, $e' \in Evt^0$ *and* e' *is a corresponding version of* e*, denoted as* $e' \cong e$ *or* $e \cong e'$.

Definition 6. *We say Evt weakly contains* Evt^0 *or* Evt^0 *is weakly contained in Evt iff for* $\forall e' \in Evt^0$, $\exists e \in Evt$ *such that* $e' \cong e$*, denoted as* $Evt \sqsupseteq Evt^0$ *or* $Evt^0 \sqsubseteq Evt$.

The initial state set of K^0 can be obtained by projecting I on V^0 directly. Then the state space and transition relation of K^0 can be generated by V^0 and Evt^0 respectively. So the over-approximate slice of the original model K with regard to $C = (I, var(\varphi))$ is defined as follows.

Definition 7. *Let* $K = (S, T, I, L, AP)$ *be the original Kripke model, the over-approximate slice model of* K *with respect to* $C = (I, var(\varphi))$ *is* $K^0 = (S^0, T^0, I^0, L^0, AP^0)$*, where*

- $S^0 \subseteq D_{v_1} \times D_{v_2} \times \cdots \times D_{v_m}$ *is a set of states, where* $v_1, v_2, \ldots, v_m \in V^0$, $D_{v_1}, D_{v_2}, \ldots, D_{v_m}$ *are the domains of* $v_1, v_2, \ldots, v_m$ *respectively,* $m = |V^0|$,
- $T^0 \subseteq S^0 \times Evt^0 \times S^0$ *is a set of transitions,*
- $I^0 \subseteq S^0$ *is a set of initial states,* $I^0 = \{s | s \in S^0 \wedge s = Prj_{V^0}(s') \wedge s' \in I\}$,
- $L^0 : S^0 \rightarrow 2^{AP^0}$, $L^0(s^0) = Prj_{AP^0}(L(s))$*, where* $s^0 \in S^0$, $s \in S$, $Prj_{V^0}(s) = s^0$,
- $AP^0 = \{p | p \in AP \wedge var(p) \subseteq V^0\}$.

In Definition 7, $v_1, v_2, \ldots, v_m \in V^0$ has a compatible order with $v_1, v_2, \ldots, v_n \in V$, $m \leqslant n$; $Prj_{V^0}(s')$ denotes the projection state of s' on V^0. For example, let s_{nt0} be a state on Fig.1, then $Prj_{\{x,y\}}(s_{nt0}) = s_{nt}$, $Prj_{\{x,z\}}(s_{nt0}) = s_{n0}$, and $Prj_{\{y,z\}}(s_{nt0}) = s_{t0}$. Let s be a state of S^0, we define $[s] = \{s' | s' \in S \wedge Prj_{V^0}(s') = s\}$. Similarly, $Prj_{AP^0}(L(s))$ is a subset of $L(s)$ by projecting $L(s)$ on AP^0.

Now we can demonstrate how the over-approximate slice in Fig.2 is built in a precise way. First, we have the slicing criterion $C = \{\{s_{nn0}, s_{nn1}\}, \{x, y\}\}$; second, the precision $V^0 = \{x, y\}$ of Fig.2 is calculated according to (2); third, the event set $Evt^0 = \{e_1, e_2, e_3, e_5, e_6, e_7, e_8, e_{10}\}$ (see Table 2) of Fig.2 is obtained by (3). Finally, we generate Fig.2 (denoted as K^0) according to Definition 7.

We get K^0 by ignoring the irrelevant portion of

Fig.1 (denoted K) which is indirectly related to φ_1. The state space of K^0 is reduced exponentially at the cost of sacrificing the strong property resistance power of static slicing, because over-approximate slicing may introduce additional behaviour that does not exist in original specification (see Figs.1 and 2). That means K^0 is a superset of K.

However, we are interested in verifying properties such that whenever a system satisfies a given property, then also does any subset of this system, i.e., if $K^0 \models \varphi$, then $K \models \varphi$, conversely, if $k^0 \not\models \varphi$, then $K \not\models \varphi$ may not be true. We can prove this by constructing a simulation relation between the original Kripke model and its over-approximate slice. Before that, the definition of simulation defined on Kripke models is given as follows.

Definition 8. *Let $K_i = \{S_i, T_i, I_i, L_i, AP_i\}$ be two Kripke models, where $i = 1, 2$, $Evt_1 \sqsupseteq Evt_2$ and $V_1 \supseteq V_2$. If $AP = AP_1 \cap AP_2$ is considered as the common atomic proposition set of K_1 and K_2, then a simulation relation defined on (K_1, K_2) is a binary relation $\mathscr{R} = S_1 \times S_2$ such that*

1) For all $s_1 \in I_1$, $\exists s_2 \in I_2$ such that $(s_1, s_2) \in \mathscr{R}$;

2) For all $(s_1, s_2) \in \mathscr{R}$, where $s_i \in S_i$, the following conditions hold.

a) $Prj_{AP}(L_1(s_1)) = Prj_{AP}(L_2(s_2))$;

b) For all $(s_1, e_1, s_1') \in T$ we have $Prj_{V_2}(s_1) = Prj_{V_2}(s')$ implies $(s_1', s_2) \in \mathscr{R}$; $Prj_{V_2}(s_1) \neq Prj_{V_2}(s_1')$ implies $\exists s_2' \in S_2$ and $\exists e_2 \in Evt_2$ such that $(s_2, e_2, s_2') \in T_2 \wedge (s_1', s_2') \in \mathscr{R} \wedge e_2 \cong e_1$.

K_1 is simulated by K_2 (or, equivalently, K_2 simulates K_1), denoted as $K_1 \preceq K_2$, if there exists a simulation relation $\mathscr{R}$ for (K_1, K_2).

Theorem 1. *$K^0 = (S^0, T^0, I^0, L^0, AP^0)$ is an over-approximate slice of $K = (S, T, I, L, AP)$ with regard to $C = (I, var(\varphi))$, if AP^0 is considered as the atomic proposition set of K and K^0, $K \preceq K^0$ holds.*

Proof. $AP^0 = AP^0 \cap AP$ (according to Definition 7), let $\mathscr{R} \subseteq S \times S^0$ be a binary relation defined on (K, K^0), $(s_1, s_2) \in \mathscr{R}$ holds iff $Prj_{V^0}(s_1) = s_2$, where $s_1 \in S$, $s_2 \in S^0$.

1) For $\forall s_1 \in I$, $\exists s_2 \in I^0$ such that $Prj_{V^0}(s_1) = s_2$ holds (according to Definition 7), it follows that $(s_1, s_2) \in \mathscr{R}$;

2) For $\forall (s_1, s_2) \in \mathscr{R}$:

a) As $s_2 = Prj_{V^0}(s_1)$, we have $L^0(s_2) = Prj_{AP^0}(L(s_1))$ holds; besides, $Prj_{AP^0}(L^0(s_2)) = L^0(s_2)$ such that $Prj_{AP^0}(L(s_1)) = Prj_{AP^0}(L^0(s_2))$ holds.

b) For all $(s_1, e_1, s_1') \in T$, $Prj_{V^0}(s_1) = Prj_{V^0}(s_1')$ implies $Prj_{V^0}(s_1') = Prj_{V^0}(s_2)$ holds, namely $(s_1', s_2) \in R$ (as $Prj_{V^0}(s_1) = \mathrm{Prj}_{V^0}(s_2)$). If $Prj_{V^0}(s_1) \neq Prj_{V^0}(s_1')$, then there exists $ae \in AE(assign(e_1))$ such that $target(ae) \in V^0$ and $var(ae) \subseteq V^0$ hold, so there exists an event $e_1' \in Evt^0$ such that $e_1' \cong e_1$ (according to (3)). Note that, in this case, if $var(guard(e_1)) \not\subseteq V^0$ holds, then $guard(e_1')$ is set to *true*, so s_2 can trigger $guard(e_1')$; if $var(guard(e_1)) \subseteq V^0$ holds, $guard(e_1') = guard(e_1)$ holds, s_2 can also trigger $guard(e_1')$. So there must exist a state $s_2' \in S^0$ such that $(s_2, e_1', s_2') \in T^0$. Because $AE(e_1')$ consists of assignment expressions (whose assignment target variable belongs to V^0) in $AE(e_1)$, e_1 has the same effect on variables of V^0 as e_1'. As $Prj_{V^0}(s_1) = Prj_{V^0}(s_2)$, it follows $Prj_{V^0}(s_1') = Prj_{V^0}(s_2')$, i.e., $(s_1', s_2') \in \mathscr{R}$ holds. □

Definition 9. *Let $K_i = (S_i, T_i, I_i, L_i, AP)$ be two Kripke structures defined on AP, for path $\pi_i \in Paths(K_i)$, where $i = 1, 2$, π_1 and π_2 are stutter trace equivalent, denoted as $\pi_1 \overset{\Delta}{=} \pi_2$, if there exists a sequence $A_0 A_1 A_2 \cdots$ with $A_i \subseteq AP$ and natural numbers $n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \cdots \geqslant 1$ such that*

$$trace(\pi_1) = \underbrace{A_0 \cdots A_0}_{n_0 times} \underbrace{A_1 \cdots A_1}_{n_1 times} \underbrace{A_2 \cdots A_2}_{n_2 times} \cdots,$$

$$trace(\pi_2) = \underbrace{A_0 \cdots A_0}_{m_0 times} \underbrace{A_1 \cdots A_1}_{m_1 times} \underbrace{A_2 \cdots A_2}_{m_2 times} \cdots,$$

where $trace(\pi_1)$ and $trace(\pi_2)$ belong to the language given by the regular expression $A_0^+ A_1^+ A_2^+ \cdots$.

Corollary 1. *If AP^0 is considered as the atomic proposition set of K and K^0, then for each trace τ of K, there exists a trace τ^0 that is stutter equivalent to τ in K^0.*

Proof. Assume that $\pi = s_1 e_1 s_2 e_2 s_3 e_3 \cdots$ is an arbitrary path of K, where $s_1 \in I$. As $K \preceq K^0$ holds, $\exists \tilde{s}_1 \in I^0$ such that $(s_1, \tilde{s}_1) \in \mathbb{R}$ holds (according to 1) of Definition 8), then there exists a path π^0 of K^0 such that $\tilde{s}_1$ is the first state of π^0. According to 2) of Definition 8, we have $Prj_{AP^0}(L(s_1)) = L(s_1) = Prj_{AP^0}(L^0(\tilde{s}_1)) = L^0(\tilde{s}_1)$, let $A_1 = L(s_1) = L^0(\tilde{s}_1)$.

Assume that $(s_1, \tilde{s}_j) \in \mathbb{R}$, i.e., $A_k = Prj_{AP^0}(L(s_i)) = L(s_i) = L^0(\tilde{s}_j)$, where state s_i transits to state s_{i+1} via e_i, s_i is the i-th state on π, e_i is the i-th event on π, $\tilde{s}_j$ is the j-th state on π^0.

According to 2) of Definition 8, if $Prj_{V^0}(s_i) = Prj_{V^0}(s_{i+1})$ holds, then $(s_{i+1}, \tilde{s}_j) \in R$, i.e., $A_k = L(s_{i+1}) = L^0(\tilde{s}_j)$; if $Prj_{V^0}(s_i) \neq Prj_{V^0}(s_{i+1})$ holds, then there exists a state $\tilde{s}_{j+1} \in S^0$ such that $(s_{i+1}, \tilde{s}_{j+1}) \in \mathscr{R}$ holds, in this case, we have $A_{k+1} = L(s_{i+1}) = L^0(\tilde{s}_{j+1})$, where $\tilde{s}_{j+1}$ is the $(j+1)$-th state of π^0.

We can conclude that for every path π on K there exists a path π^0 on K^0 such that $trace(\pi)$ and $trace(\pi^0)$ belong to the same regular expression $A_1^+ A_2^+ A_3^+ A_4^+ \cdots A_k^+ \cdots$, i.e., for each trace τ of K, there exists a trace τ^0 that is stutter equivalent to τ on K^0. □

Corollary 2. *If AP^0 is considered as the atomic proposition set of K and K^0, then $K^0 \models \varphi$ implies $K \models \varphi$, where φ is an $LTL_{\backslash O}$ formula ($LTL_{\backslash O}$ denotes the LTL formula without the next step operator O[25]).*

Proof. We only consider the language constituted by AP^0, because it is sufficient to prove the property of interest. As $K^0 \models \varphi$, any trace of $Traces(K^0)$ belongs to $\mathbb{L}(\varphi)$ ($\mathbb{L}(\varphi)$ is a language of words over the alphabet 2^{AP}). It follows that any trace of $Traces(K)$ belongs to $\mathbb{L}(\varphi)$ (according Corollary 1), i.e., $K \models \varphi$. $\square$

Corollary 3. *$K^0 = (S^0, T^0, I^0, L^0, AP^0)$ is an over-approximate slice of $K = (S, T, I, L, AP)$ implies*

$$S \subseteq \bigcup_{s \in S^0} [s].$$

Proof. According to Theorem 1, we have $K \preceq K^0$ immediately, so for $\forall s \in S$ there exists a state $s' \in S^0$ such that $Prj_{V^0}(s) = s'$ holds (according to Corollary 1), i.e., $s \in [s']$. Thus

$$S \subseteq \bigcup_{s' \in S^0} [s']. \qquad \square$$

4 Spurious Counterexample Decision

We have to identify whether a counterexample found on K^0 can be concretized on K or not (according to Corollary 2). This is spurious counterexample decision, which essentially is to check if there exists a path on K stutter equivalent to the counterexample found on K^0. Let $\tilde{\pi} = \tilde{s}_1 e'_1 \tilde{s}_2 e'_2 \cdots \tilde{s}_{n-1} e'_{n-1} \tilde{s}_n$ be a counterexample found on K^0, where $\tilde{s}_1 \in I^0$. Spurious counterexample decision starts from $I_0 = \{s | s \in I \wedge s \in [\tilde{s}_1]\}$, where $[\tilde{s}_1] = \{s | s \in S \wedge Prj_{V^0}(s) = \tilde{s}_1\}$ is the equivalence class of $\tilde{s}_1$ with regard to simulation relation $\mathbb{R}$. Let $Reach_{Evt\backslash Evt^0}(I_0)$ be the reachable state set of I_0 via event set $Evt\backslash Evt^0 = \{e_i | e_i \in Evt \wedge \neg(\exists e'_i \in Evt^0.(e'_i \cong e_i))\}$, then the following proposition holds.

Proposition 1. *$s \in Reach_{Evt\backslash Evt^0}(I_0)$ implies $Prj_{V^0}(s) = \tilde{s}_1$.*

Proof. Assume there exist two states s and s' that belong to $Reach_{Evt\backslash Evt^0}(I_0)$, and an event e belongs to $Evt\backslash Evt^0$ such that s can transit to s' via e, where $Prj_{V^0}(s) = \tilde{s}_1$, and $Prj_{V^0}(s') \neq \tilde{s}_1$. It follows that $\exists ae \in AE(e)$ (which can change the value of variables in V^0) such that $target(ae) \in V^0$. In this case, $\exists e' \in Evt^0$ such that $e' \cong e$, this is contradictory to $e \in Evt\backslash Evt^0$. So for all paths starting from states in I_0 via $Evt\backslash Evt^0$, there does not exist any transition that can change the value of variables in V^0. As for all $s \in I_0$ we have $Prj_{V^0}(s) = \tilde{s}_1$ holds, therefore, there does not exist state s' in $Reach_{Evt\backslash Evt^0}(I_0)$ such that $Prj_{V^0}(s) \neq \tilde{s}_1$ holds. $\square$

Proposition 1 states that every state in $Reach_{Evt\backslash Evt^0}(I_0)$ belongs to $[\tilde{s}_1]$. Let $S_1 = Reach_{Evt\backslash Evt^0}(I_0)$, then whether the path fragment $\tilde{s}_1 e'_1 \tilde{s}_2$ can be concretized is equivalent to whether there exists a state in $Reach_{Evt\backslash Evt^0}(I_0)$ that can transit to another state in $[\tilde{s}_2]$ via e_1. Let $Posts_{e_1}(S_1)$ denote the successor set of S_1 through e_1, then the following proposition holds.

Proposition 2. *$(\tilde{s}_1, e'_1, \tilde{s}_2) \in T^0$ implies $Posts_{e_1}(S_1) \subseteq [\tilde{s}_2]$.*

Proof. Let $s_1 \in S_1$, then we have $Prj_{V^0}(s_1) = \tilde{s}_1$ hold (according to Proposition 1). Let $s_2 = Post_{e_1}(s_1)$ be the direct successor of s_1 through e_1, then we have $(s_1, e_1, s_2) \in T$. Because $e'_1 \cong e_1$, $(Prj_{V^0}(s_1), e'_1, Prj_{V^0}(s_2)) \in T^0$ holds, i.e., $(\tilde{s}_1, e'_1, Prj_{V^0}(s_2)) \in T^0$. As $(\tilde{s}_1, e_1, \tilde{s}_2) \in T^0$ holds, it follows that $Prj_{V^0}(s_2) = \tilde{s}_2$, i.e., $s_2 \in [\tilde{s}_2]$. Therefore, we can conclude that $Posts_{e_1}(S_1) \subseteq [\tilde{s}_2]$ holds. $\square$

From Proposition 2, if $Posts_{e_1}(S_1)$ is not empty, the states in $Posts_{e_1}(S_1)$ belong to $[\tilde{s}_2]$, i.e., there exists a concrete path corresponding to $\tilde{s}_1 e'_1 \tilde{s}_2$ in the original model. Conversely, $\tilde{s}_1 e'_1 \tilde{s}_2$ is not feasible in the original model. We have the following conclusions generalized from Propositions 1 and 2.

Proposition 3. *The following conclusions hold.*

1) For all $s \in S_i$ we have $Prj_{V^0}(s) = \tilde{s}_i$ holds,

2) $(\tilde{s}_i, e_i, \tilde{s}_{i+1}) \in T^0$ implies $Posts_{e_i}(S_i) \subseteq [\tilde{s}_{i+1}]$, where $1 \leqslant i < |\tilde{\pi}|$, and S_i is defined as follows.

$$S_i = Reach_{Evt\backslash Evt^0}(Posts_{e_i}(S_{i-1})). \qquad (4)$$

It follows that $\tilde{\pi}$ can be concretized by computing S_i iteratively starting from S_i, and it will be finished after at most n iterations, where $n = |\tilde{\pi}|$. In this case, if $S_n \neq \varnothing$, then $\tilde{\pi}$ can be concretized, so the verification is finished with a counterexample $\tilde{\pi}$; otherwise, we can conclude that $\tilde{\pi}$ is a spurious counterexample. We call S_i the feasible equivalent state set of $\tilde{s}_i$ on the original model K iff $S_i \neq \varnothing$, denoted as $S^{fe}(\tilde{s}_i)$.

Theorem 2. *Assume that $\tilde{\pi}$ is a counterexample on over-approximate slice K^0 of K. If there exists an i such that $S_1, S_2, \ldots, S_{i-1} \neq \varnothing$ and $S_i, S_{i+1}, \ldots, S_n = \varnothing$, then $\tilde{\pi}$ is a spurious counterexample, where $1 < i \leqslant n$.*

Theorem 2 can be proved by Proposition 3 easily. Now we can show why counterexample π in Example 1 is spurious according to Theorem 2. In order to find a concrete path in Fig.1 that is corresponding to π, we first calculate S_1. In Example 1, $S_1 = \{s_{nn0}, s_{nn1}\}$, because $[s_{nn}]$ will not trigger any event of $Evt\backslash Evt^0 = \{e_4, e_9\}$, we have $S_2 = \{s_{tn0}, s_{tn1}\}$ according to (4). Similarly, we have $S_3 = \{s_{cn0}, s_{cn1}\}$ and $S_4 = \{s_{ct0}, s_{ct1}\}$. As there does not exist any state in S_4 that can trigger the guard of e_8, we have $S_5 = \varnothing$. It follows that $\tilde{\pi}$ is a spurious counterexample according to Theorem 2 (see Fig.3).

5 Refine Local Slice

Spurious counterexample means the over-approximate slice is too rough to prove a given property. So we must refine the over-approximate slice. Lazy slicing only refine a portion of the slice that has not been checked, which is the main difference between our method and CEGAR-based slicing.

Let $\tilde{\pi} = \tilde{s}_1 e'_1 \tilde{s}_2 e'_2 \cdots \tilde{s}_{n-1} e'_{n-1} \tilde{s}_n$ be a spurious counterexample on K^0, then there exists an i such that $S_1, S_2, \ldots, S_{i-1} \neq \varnothing$ and $S_i, S_{i+1}, \ldots, S_n = \varnothing$ (Theorem 2), where K^0 is an over-approximate slice of an original Kripke model K, $1 < i \leqslant n$.

Definition 10. *Let* $\tilde{\pi}[..i-1] = \tilde{s}_1 e'_1 \tilde{s}_2 e'_2 \cdots \tilde{s}_{i-1}$ *be a feasible prefix of* $\tilde{\pi}$, *where* $\tilde{s}_{i-1}$ *is the dead end state of* $\tilde{\pi}[..i-1]$.

Feasible prefix is the path fragment of spurious counterexample that can be concretized in original model. Dead end state $\tilde{s}_{i-1}$ is able to transit to $\tilde{s}_i$ through e'_{i-1} on K^0. But this cannot happen on the original model K, because the guard of e_{i-1} contains variables indirectly related to the property of interest. These variables are left out when computing the over-approximate event set Evt^0, which causes the guard of e'_{i-1} to be triggered on $\tilde{s}_{i-1}$, and results in the transition from $\tilde{s}_{i-1}$ to $\tilde{s}_i$ via e'_{i-1}.

Theorem 3. $\tilde{\pi} = \tilde{s}_1 e'_1 \tilde{s}_2 e'_2 \cdots \tilde{s}_{n-1} e'_{n-1} \tilde{s}_n$ *is a spurious counterexample on* K^0, $\tilde{s}_{i-1}$ *is the dead end state, then* $var(guard(e_{i-1})) \not\subseteq V^0$.

Proof. Assume $var(guard(e_{i-1})) \subseteq V^0$ holds, let $s \in S_{i-1} = S^{fe}(\tilde{s}_{i-1}) \subseteq [\tilde{s}_{i-1}]$. Because $\tilde{s}_{i-1}$ is the dead end state, there does not exist any state in S_{i-1} that can trigger the guard of e_{i-1} (Theorem 2), it follows that $guard(e_{i-1})$ does not hold on state s. As $var(guard(e_{i-1})) \subseteq V^0$, we have $guard(e'_{i-1}) = guard(e_{i-1})$, where $e_{i-1} \cong e'_{i-1} \wedge e_{i-1} \in Evt \wedge e'_{i-1} \in Evt^0$. Therefore, $guard(e'_{i-1})$ is false on $Prj_{V^0}(s)$, so $guard(e'_{i-1})$ cannot be triggered on $\tilde{s}_{i-1}$ too. It means that $\tilde{s}_{i-1}$ cannot transit to $\tilde{s}_i$ via e'_{i-1}, but this is contradictory to the fact that $\tilde{s}_{i-1}$ can transit to $\tilde{s}_i$ via e'_{i-1} on K^0. □

Theorem 3 guarantees that the precision of the refined over-approximate slice is inevitably higher than the one before refinement, and it also ensures that the refinement iteration terminates when the precision is increased to the same as the original model in the worst case. In order to refine only a local slice which has not been explored before, we first need to compute the slicing criterion of this local slice, let it be $C_r = (I_r, V_r)$.

V_r consists of $var(guard(e_{i-1}))$ and the precision of K^0 (the over-approximate slice generates the spurious counterexample), so we have $V_r = V^0 \cup var(guard(e_{i-1}))$. Then we can obtain the precision V^0_r and the event set Evt^0_r of K^0_r which is refined from K^0 according to (3).

I_r consists of two parts. One is the uncovered initial states of K^0, denoted as I_I. The other is the uncovered feasible successor set of the feasible prefix of $\tilde{\pi}$ which is found on K^0, denoted as I_P. We give the definition of cover relation first before introducing how I_I and I_P are computed.

Definition 11. K^0_i *is an over-approximate slice of* K, *where* $i = 1, 2$. $\tilde{s}_1$ *covers* $\tilde{s}_2$ *or* $\tilde{s}_2$ *is covered by* $\tilde{s}_1$ *iff* $[\tilde{s}_1] \supseteq [\tilde{s}_2]$, *denoted as* $\tilde{s}_1 \lhd \tilde{s}_2$, *where* $\tilde{s}_i$ *is a state of* K^0_i *or* K.

I_I is computed as follows.

$$I_I = \bigcup_{\tilde{s} \in I^0 \wedge \forall s' \in R. \neg(s' \rhd \tilde{s})} [\tilde{s}], \tag{5}$$

where R stores the states that have been traversed. However, we cannot obtain I_P by simply adding the direct successor of $\tilde{s}_j$ (let $\tilde{s}_j$ be a state on the feasible prefix of the spurious counterexample). The main reason is that we cannot guarantee its direct successor, let it be $\tilde{s}$, is a feasible successor of $\tilde{s}_j$, i.e., the corresponding transition from $\tilde{s}_j$ to $\tilde{s}$ may not occur on the original model. So we take the uncovered direct successors of $S^{fe}(\tilde{s}_j)$ as the feasible equivalent successors of $\tilde{s}_j$. Note that if a state is covered it means that it has been traversed previously.

Definition 12. *Let* $\tilde{s}_j$ *be a state of the feasible prefix of the spurious counterexample* $\tilde{\pi}$, $Post^{fe}(\tilde{s}_j)$ *is the feasible equivalent successor set (on the original Kripke model K) of* $\tilde{s}_j$ *iff*

$$Post^{fe}(\tilde{s}_j) = \{s | s \in Posts(S^{fe}(\tilde{s}_j)) \wedge \forall s' \in R. \neg(s' \rhd s)\},$$

where $Posts(S^{fe}(\tilde{s}_j))$ denotes the direct successor set of $S^{fe}(\tilde{s}_j)$. So I_P is computed as follows.

$$I_P = \bigcup_{1 \leqslant j \leqslant i-1} \bigcup_{s \in Post^{fe}(\tilde{s}_j)} s. \tag{6}$$

In Step 3 of Example 1, we get $V_r = \{x, y\} \cup var(guard(e_8)) = \{x, y, z\}$, $I_I = \varnothing$ and I_P equals to the state set consisting of the red states in Fig.4. So we can obtain a refined local over-approximate slice K^0_r induced by $C_r = (I_I \cup I_P, V_r)$.

6 Lazy Slicing Based Exploration

In order to avoid the additional cost of CEGAR-based slicing, we refine a local slice which has not been checked, and then traverse along the initial state set of the refined local slice K^0_r induced by $C_r = (I_r, V_r)$. We can get the initial state set I^0_r directly by projecting I_r

onto K_r^0, but in order to integrate slicing-verification-refinement iteration into one step, we compute I_r^0 on-the-fly by focus operator and defocus operator.

Definition 13. *Focus operator is a projection function*

$$F(2^{I^0}, V_r^0) = \bigcup_{s \in I^0 \wedge \forall s' \in R. \neg(s' \rhd \tilde{s})} \bigcup_{Prj_{prec(\tilde{s})}(s) = s \wedge s \in S_r^0} s,$$

where $prec(\tilde{s})$ is the precision of $\tilde{s}$. The role of focus operator is to project the uncovered initial state of K^0 onto the refined over-approximate slice k_r^0 (with a higher precision). While the role of defocus operator is to project states on the original Kripke model K onto a given over-approximate slice K^0 of K.

Definition 14. *Defocus operator is a projection function,*

$$D(2^S, V^0) = \bigcup_{s \in 2^S} Prj_{V^0}(s),$$

where S is the state set of the original Kripke model K, V^0 is precision of K^0.

The initial state set I_r^o of K_r^0 consists of two parts: one part is projected from the uncovered initial state of K^0 by focus operator, the other part is projected from the feasible equivalent successors of the feasible prefix of the spurious counterexample by the defocus operator. Note that I_r^0 is not only the initial state set of K_r^0, but also the states which are on the boundary between the explored state space and the unexplored one (for example, the states which belong to both Fig.1 and Fig.6 and their predecessors which also belong to Fig.2 and Fig.6).

We first deal with the initial state projected from the feasible successors of the dead end state. Let it be $\tilde{s}_{i-1}$. In fact, we compute these initial states directly from the feasible equivalent successors of $\tilde{s}_{i-1}$ by the help of defocus operator. These states returned by defocus operator are both the initial states of K_r^0 and the feasible successors of $\tilde{s}_{i-1}$ on K^0. Our algorithm continues exploring along these initial states on K_r^0 in DFS (depth first search) order.

After all of the states on the feasible prefix have been handled, if there still exist unexplored initial states of K^0, we project these states onto K_r^0 by the help of focus operator, then continue searching from these states one by one in the same way as before.

Step 4 of Example 1 shows how lazy slicing works when a spurious counterexample is found. In Fig.5, s_{ct} is the dead end state, the feasible equivalent states of s_{ct} are $S^{fe}(s_{ct}) = \{s_{ct0}, s_{ct1}\}$, the feasible equivalent successors of s_{ct} are $Post^{fe}(s_{ct}) = \{s_{nt0}, s_{nt1}\}$. As the precision of K_r^0 is $V_r^0 = \{x, y, z\}$, we get $D(Post^{fe}(s_{ct}), \{x, y, z\}) = \{s_{nt0}, s_{nt1}\}$. Then our algorithm takes s_{nt0}, s_{nt1} as successors of s_{ct}, thus it works as shown in Fig.5.

In Example 1, there is no state that does not satisfy φ_1 has been found, and there does not exist any state corresponding to the initial state in K^0 (see Fig.2) which is not covered after exploring along $s_{nn}s_{tn}s_{cn}s_{ct}$. So we have $K \models \varphi_1$.

Lazy slicing detects cycle path with the help of cover relation. If a state is covered by R, let it be s_c, the exploration stops at s_c and turns to the other branch of the ancestor of s_c. This is because any error state reached from s_c will be traversed along the state that covers s_c, regardless of whether this error state has been found or not.

In practice, an equivalent condition of Definition 8 can be used to simplify cover relation decision.

Proposition 4. *$\tilde{s}_1 \rhd \tilde{s}_2$ iff $V_1^0 \subseteq V_2^0 \wedge Prj_{V_1^0}(\tilde{s}_2) = \tilde{s}_1$, where K_i^0 is the over-approximate slice of original model K, $\tilde{s}_i$ is a state on K_i^0, V_i^0 is the precision of K_i^0, $i = 1, 2$.*

Proof. First, we prove $[\tilde{s}_1] \supseteq [\tilde{s}_2] \Rightarrow V_1^0 \subseteq V_2^0 \wedge Prj_{V_1^0}(\tilde{s}_2) = \tilde{s}_1$. Assume $[\tilde{s}_1] \supseteq [\tilde{s}_2]$ and $V_1^0 \not\subseteq V_2^0$, i.e., $\exists v \in V_1^0$ such that $v \notin V_2^0$. In this case, $Prj_v(s_1) = Prj_v(\tilde{s}_1)$ holds for all $s_1 \in [\tilde{s}_1]$. Because $v \notin V_2^0$, there inevitably exists state $s_2 \in [\tilde{s}_2]$ such that $Prj_v(s_2) = Prj_v(\tilde{s}_1)$, namely $s_2 \notin [\tilde{s}_1]$. It is contradictory to the fact that $[\tilde{s}_1] \supseteq [\tilde{s}_2]$. Thus we have $[\tilde{s}_1] \supseteq [\tilde{s}_2]$ implies $V_1^0 \subseteq V_2^0$. In this case, $Prj_{V_1^0}(\tilde{s}_2) \neq \tilde{s}_1$, then there exists at least one variable $v' \in V_1^0$ such that $Prj_{v'}(\tilde{s}_1) = Prj_{v'}(\tilde{s}_2)$, i.e., $v' \notin V_2^0$, which is contradictory to $V_1^0 \subseteq V_2^0$.

$V_1^0 \subseteq V_2^0 \wedge Prj_{V_1^0}(\tilde{s}_2) = \tilde{s}_1 \Rightarrow [\tilde{s}_1] \supseteq [\tilde{s}_2]$. For all $s \in [\tilde{s}_2]$, because $Prj_{V_1^0}(\tilde{s}_2) = \tilde{s}_1$ and $V_1^0 \subseteq V_2^0$, we have $s \in [\tilde{s}_1]$. It follows that $[\tilde{s}_2] \in [\tilde{s}_1]$. □

Algorithm 1 describes the main steps of lazy slicing. The first step is to compute the first over-approximate slice K_1^0 from an initial Kripke model with regard to a given property φ, then we initialize $K_c^0 = K_1^0$ (lines 2∼5, Algorithm 1), where K_c^0 always denotes the refined over-approximate slice in the slicing-verification-refinement iteration, we call K_c^0 the current slice. The while iteration at lines 6∼12 guarantees all paths starting from the initial state set of K_1^0 will be explored (no matter in what precision). If an error state is found (line 11, Algorithm 1), lazy slicing determines whether a real counterexample is found or not (line 12). If lazy slicing discovers a spurious counterexample, it refines the current over-approximate slice (line 9, Algorithm 2), then continues searching on the refined over-approximate slice (lines 15∼16, Algorithm 1). If no error state is found, it goes on exploring the state space with current precision (line 19, Algorithm 1).

Note that K_c^0 denotes the first over-approximate

Algorithm 1 LazySlicing (K, φ)

```
Require: Kripke Structure of the original model $K$,
         property $\varphi$
Ensure: Return true if $K \models \varphi$, otherwise return false
        plus a counterexample
Compute over-approximate slicing $K_1^0$;
$V_c^0 := V_1^0$;
$R := \varnothing$; Stack $U :=$ null; {$R$ is the set of states that have been traversed, $U$ is the stack that stores the current search path}
Bool counterexample:=false;
$I_c^0 := I_1^0 := F(I_c^0, V_c^0)$;
while $\neg counterexample \wedge I_c^0 \neq \varnothing$ do
  Get a state $s$ form $I_c^0$ and remove it from $I_c^0$;
  $Push(s, U)$; $R := R \cup \{s\}$;
  repeat
    $s' := Top(U)$;
    if $s' \not\models \varphi$ then
      if $Cedecision(U, V_c^0)$ then
        counterexample:= true: {marks a real counterexample then terminates}
      else
        $I_c^0 := F(I_c^0, V_c^0)$;
        $Getpost(V_c^0, Evt_c^0)$;
      end if
    else
      $Getpost(V_c^0, Evt_c^0)$;
    end if
  until $counterexample \vee U =$ null
end while
if $(\neg counterexample)$ then
  return $K \models \varphi$;
else
  return $Reverse(U)$;
end if
```

Algorithm 2 Cedecision (U, V_c^0)

```
Require: Stack $U$, precision of the current over-
         approximate slicing $V_c^0$
Ensure: Return true if the counterexample in $U$ can be
        concretized, otherwise return false
Let $\tilde{s}_1\tilde{s}_2 \cdots \tilde{s}_n$ be the path fragment with precision $V_c^0$; {this path fragment is on the top of stack $U$}
if $\tilde{s}_1$ is not the bottom element of $U$ then
  Let $\tilde{s}_t$ be the element under $\tilde{s}_1$ in $U$; {$\tilde{s}_t$ is the dead end state}
  $S^{fe}(\tilde{s}_1) := Post^{fe}(\tilde{s}_t)$;
end if
for $i = 1$ to $n$ do
  if $S^{fe}(\tilde{s}_i) = \varnothing$ then
    Pop $\tilde{s}_n, \tilde{s}_{n-1}, \ldots, \tilde{s}_{i+1}, \tilde{s}_i$ from $U$ and move them from $R$;
    $Refine(V_c^0, evt)$; {$evt$ is the event making $\tilde{s}_{i-1} \xrightarrow{evt} \tilde{s}_i$ happen}
    return false;
  else
    $\tilde{s}_i.post^{fe} := Post^{fe}(\tilde{s}_i)$;
  end if
end for
return true;
```

slice K_1^0 initially, we explore on K_c^0 through V_c^0 and Evt_c^0 (Algorithm 3). If a spurious counterexample is found, we refine K_c^0, and assign the precision and event set of the refined over-approximate slice K_2^0 to V_c^0 and Evt_c^0 respectively (Algorithm 4). At this time, K_c^0 denotes K_2^0. We continue exploring on $K_c^0(K_2^0)$. If a new refinement is performed, then K_c^0 will denote K_3^0. It means that K_c^0 always denotes the over-approximate slice with the highest precision (i.e., K_i^0) in the i-th iteration.

Algorithm 3 Getpost (V_c^0, Evt_c^0)

```
Require: $V_c^0$, $evt_c^0$
Ensure: Find an immediate successor (not covered by
        $R$) of $Top(U)$
Bool $post :=$ false;
while $U \neq$ null $\wedge \neg post$ do
  Let $s := Top(U)$;
  if $prec(s) \neq V_c^0$ then
    $S_{post} := s.post^{fe}$;
  else
    $S_{post} := Posts_{Evt_c^0}(s)$; {$Posts_{Evt_c^0}(s)$ is the immediate successor set of $s$ wrt event set $Evt_c^0$}
  end if
  if $\exists s_{post} \in S_{post}$ s.t. $\nexists s_R \in R.(s_R \rhd s_D)$ then
    {$s_D$ is the only element in $D(\{s_{post}\}, V_c^0)$}
    $Push(s_D, U)$;
    $post :=$ true;
  else
    $Pop(U)$;
  end if
end while
```

Algorithm 4 Refine (V_c^0, evt)

```
Require: $V_c^0$: precision of the current over-approximate
         slicing, $evt$: event lead to the unreachable
         bad state
Ensure: Compute precision and event set of a refined
        over-approximate slicing
$V_1 := V_c^0 \cup var(guard(evt))$;
repeat
  $V_2 := V_1$;
  for every $e \in Evt$ do
    for every atom expression $ae \in assign(e)$ do
      if $target(ae) \in V_2$ then
        $V_1 := V_1 \cup var(ae)$;
      end if
    end for
  end for
until $V_1 = V_2$
$V_c^0 := V_1$;
$Evt_c^0 = \varnothing$;
for every $e \in Evt$ do
  if $\exists ae \in assign(e).(target(ae) \in V_c^0)$ then
    Generate an event $e'$, let $assign(e') := \{ae | ae \in$
```

```
        $assign(e) \wedge target(ae) \in V_c^0\}$;
17.     if $var(guard(e)) \subseteq V_c^0$ then
18.        $guard(e') := guard(e)$;
19.     else
20.        $guard(e') :=$ true;
21.     end if
22.     $Evt_c^0 := Evt_c^0 \cup e'$;
23.   end if
24. end for
```

Spurious counterexample decision (Algorithm 2) of lazy slicing can be done by dealing with only the path fragment with current precision instead of the whole path of a counterexample (lines 2∼5, Algorithm 2). This improves the efficiency of spurious counterexample decision remarkably. Theorem 2 and Theorem 5 ensure the correctness of our decision algorithm. The "for" iteration (lines 6∼14, Algorithm 2) identifies spurious counterexample according to Theorem 2. If the given counterexample is spurious (line 7, Algorithm 2), it pops the infeasible suffix of the spurious counterexample (line 8, Algorithm 2), then refines the current over-approximate slice (line 9, Algorithm 2). Else it preserves the feasible equivalent state set for each state on the feasible prefix of the spurious counterexample (line 12, Algorithm 2), $\tilde{s}_i.post^{fe}$ stores the feasible equivalent successors of s_i.

Algorithm 3 explores the state space on the current over-approximate slice. The while iteration guarantees that a successor of $Top(U)$ will be found except that all the successors of U are covered by R (lines 2∼16, Algorithm 3). We put the direct successors (the feasible equivalent successors or successors on the same over-approximate slice) of $Top(U)$ into S_{post} no matter whether the precision of $Top(U)$ is the same as the current over-approximate slice or not (lines 4∼8, Algorithm 3). If all the elements of S_{post} are covered by R, it means all successors of $Top(U)$ have already been explored, so we pop stack U (lines 13∼15, Algorithm 3). Else we will find an uncovered successor of $Top(U)$, then we traverse along this successor by pushing it into stack U (lines 9∼12, Algorithm 3). Algorithm 3 is carried out on-the-fly according to the event set determined by the precision of the current over-approximate slice. Let s be a state in stack U, Algorithm 3 ensures the paths that have not been traversed from s will be unfolded in current precision, which is one of the key steps to avoid repeating the work done before.

Algorithm 4 refines the over-approximate slice that produces the spurious counterexample. Lines 1∼12 is a fix point computation which generates the precision of the refined over-approximate slice ((2)). Lines 13∼24 generate the event set of the refined over-approximate slice according to (3). Note that lines 17∼21 deal with the guard in the same method with [18], we will introduce an improved method to generate the guards of the event set in Section 7 in order to produce a more precise over-approximate slice.

A remarkable feature of lazy slicing is that the search path with ascending precisions, which is caused by dynamic local refinement. The definition of a path with ascending precisions is given as follows.

Definition 15. *K is the Kripke model of the original specification, K_1^0 is the first over-approximate slice of K, K_i^0 is the local over-approximate slice refined from K_{i-1}^0, where K_i^0 is the i-th over-approximate slice expanded by lazy slicing. Then*

$$\tilde{\pi} = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}\tilde{s}_{m2}\cdots\tilde{s}_{mn_m}$$

is a search path of LazySlicing (φ, K), where $\tilde{\pi}_i = \tilde{s}_{i1}\tilde{s}_{i2}\cdots\tilde{s}_{in_i}$ is a path fragment on K_i^0 with the precision V_i^0, n_i is the length of $\tilde{\pi}_i$, $1 \leqslant i \leqslant m$.

Theorem 4. *Local slice refinement of LazySlicing (φ, K) iterates at most $|V| - 1$ times.*

Proof. Assume that the refinement of LazySlicing (φ, K) iterates more than $|V|-1$ times. As $V_1^0 = var(\varphi)$ is not empty, the minimum value of $|V_1^0|$ is 1. The minimum value of $|V_2^0|$ is 2 after the first refinement iteration (at least one variable is added into V_1^0 in refinement according to Theorem 3). In a similar way, the minimum value of $|V_3^0|$ is 3 after the second iteration. Thus we can conclude that the minimum value of $|V_{|V|}^0|$ is $|V|$ after the $|V| - 1$-th iteration by induction, and the minimum value of $|V_{|V|+1}^0|$ is $|V| + 1$ after the $|V|$-th iteration. It is impossible that $|V_{|V|+1}^0| > |V|$. □

Corollary 4. *A path of lazy slicing algorithm has at most $|V|$ path fragments with ascending precisions.*

The proof of Corollary 4 is straightforward. As we know a path of *LazySlicing*(φ, K) enters a new path fragment with higher precision after each refinement, and the refinement of *LazySlicing*(φ, K) iterates at most $|V| - 1$ times (according to Theorem 4), so the path of *LazySlicing*(φ, K) enters at most $|V| - 1$ new path fragments with ascending precisions in turn. It follows that the path of *LazySlicing*(φ, K) consists of at most $|V|$ path fragments with ascending precisions.

Spurious counterexample decision, a time-consuming work, benefits a lot from a path with ascending precisions. It can be done by identifying the last path fragment with the highest precision instead of the full path, which significantly reduces the cost of spurious counterexample decision.

Theorem 5. *Let $\tilde{\pi}$ be a path of LazySlicing (φ, K), then the path fragment starting from the first state on $\tilde{\pi}$ to the first state on the last path fragment (with the highest precision) of $\tilde{\pi}$ is feasible.*

Proof. Let $\tilde{\pi} = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}\tilde{s}_{m2}\cdots\tilde{s}_{mn_m}$ be a path of $LazySlicing(\varphi, K)$, it is equivalent to prove $\tilde{\pi}_m = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}$ is feasible, where $1 \leqslant m \leqslant |V|$. If $m = 1$, then $\tilde{\pi}_1 = \tilde{s}_{11}$. According to Definition 7 we know that there exists at least one state on the original model K corresponding to $\tilde{s}_{11}$, so $\tilde{\pi}_1$ is feasible. If $m = 2$, then $\tilde{\pi}_2 = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}$. According to Theorem 2 we know $\tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}$ is the feasible prefix of the spurious counterexample found on the first over-approximate slice, and $\tilde{s}_{1n_1}$ is the dead end state. Then there exists $s \in Post^{fe}(\tilde{s}_{1n_1})$ such that $\{\tilde{s}_{21}\} = D(\{s\}, V_2^0)$, i.e., $Prj_{prec(\tilde{s}_{21})}(s) = \tilde{s}_{21}$, so we can say $\tilde{\pi}_2$ is feasible. In a similar way, we can conclude that $\tilde{\pi}_m$ is feasible for all $1 \leqslant m \leqslant |V|$. □

7 Improved Over-Approximate Slice

In Sections 2 and 3, we build an over-approximate slice using the same method as [18] which leads to a much coarser slice. The reason is that too much additional behavior may be introduced into the slice model. For example, assume a large number of variables are involved in the guard of event e, if $var(guard(e)) \nsubseteq V^0$, the guard of e is set to true. In this case, the logical conditions related to $V^{rel} = \{v | v \in var(guard(e)) \wedge v \in V^0\}$ in $guard(e)$ are neglected, which is responsible for additional behavior.

In this section, we propose an improved approach to construct a more precise over-approximate slice by preserving the conditions related to V^{rel}, i.e., only the conditions related to $var(guard(e)) \backslash V^{rel}$ are ignored. We first convert the guard of an event to the disjunctive normal form. Let $C(guard(e))$ denote the clause set of $guard(e)$, $c \in C(guard(e))$ is the conjunction of literals which is composed of atomic propositions or its negations. Then all clauses of an event can be recalculated as follows.

$$c = \bigwedge_{p \in P(c) \wedge var(p) \subseteq V^0} p, \tag{7}$$

where $P(c)$ is the literal set of c. We get $guard(e)$ as follows.

$$guard(e) = \bigvee_{c \in C(guard(e)) \wedge P(c) \neq \varnothing} c. \tag{8}$$

An improved event set Evt^0_{imp} is recalculated from the result of (3) by (7) and (8). Then we get a more precise over-approximate slice from K^0 (built in Section 3) by replacing Evt^0 with Evt^0_{imp}, denoted as $K^0_{imp} = (S^0_{imp}, \rightarrow^0_{imp}, I^0_{imp}, L^0, AP^0)$. K^0_{imp} contains less unnecessary behavior than K^0 in most cases (at least no more than K^0 in the worst case). Furthermore, K^0_{imp} simulates the original Kripke model K just as K^0 simulates K, which can be proved with a similar method as Theorem 1. So lazy slicing can be performed on K^0_{imp} directly without loss of correctness, and the verification cost of lazy slicing on K^0_{imp} will be reduced to a large degree.

Theorem 6. *π belongs to $Paths(K^0_{imp})$ implies π belongs to $Paths(K^0)$.*

Proof. Let $\pi = s_1 s_2 s_3 \cdots$ be a path of K^0_{imp}, and e''_i leads to the transition from s_i to s_{i+1}, which is a different version of e_i of Evt, $i \geqslant 1$. Because K^0_{imp} and K^0 have the same precision and initial state set, s_1 also belongs to I^0. Let e'_1 be the version of e_1 in Evt^0. If $var(guard(e_1)) \subseteq Evt^0$, then $guard(e''_1) = guard(e'_1) = guard(e_1)$. So $guard(e'_1)$ can be satisfied on state s_1, and e'_1 can lead to the transition from s_1 to s_2 on K^0. If $var(guard(e_1)) \nsubseteq Evt^0$, then $guard(e'_1)$ is true. So there still exists a transition from s_1 to s_2 via e'_1 on K^0. Therefore, there always exists a transition from s_1 to s_2 via e'_1 on K^0 corresponding to the transition from s_1 to s_2 via e''_1 on K^0_{imp}. For the same reason, there exists a transition from s_2 to s_3 via e'_2 on K^0 corresponding to the transition from s_1 to s_2 via e''_1 on K^0_{imp}. So we can conclude that a path that is the same as π can always be found on K^0. □

The advantages of our improved over-approximate slice can be shown using Example 1. Table 3 lists the improved event set of Table 2.

Table 3. Improved Event Set

No.	Guard	Assignment
e''_1	$x = n$	$x = t$
e''_2	$x = t \wedge y = n$	$x = c$
e''_3	$x = t \wedge y = t$	$x = c$
e''_5	$x = c$	$x = n$
e''_6	$y = n$	$y = t$
e''_7	$y = t \wedge x = n$	$y = c$
e''_8	$y = t \wedge x = t$	$y = c$
e''_{10}	$y = c$	$y = n$

Fig.7 shows the state transition system of the improved over-approximate slice K^0_{imp}. Compared with

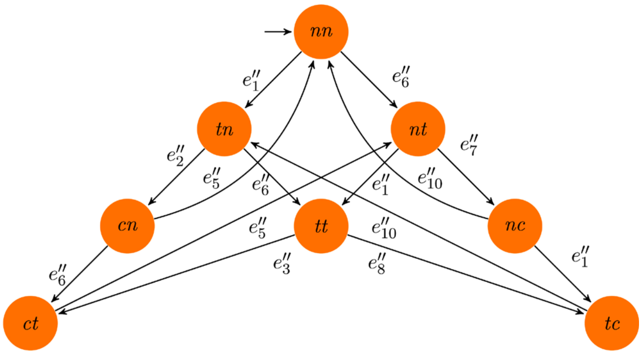


Fig.7. State transition system of K^0_{imp}.

Fig.2, the behavior of K^0_{imp} is obviously less than $K^0_{\varphi_1}$. For example, there is no transition reaches state S_{cc} in Fig.7. Lazy slicing can prove $K \models \varphi_1$ on K^0_{imp} without any refinement, i.e., the verification can be finished by exploring only eight states on Fig.7.

8 Algorithm Analysis

In this section we first consider the correctness and termination of lazy slicing, then give an analysis on the savings achieved by our method.

8.1 Correctness

The correctness of our lazy slicing is expressed by the following theorem.

Theorem 7. *$K = (S, T, I, L, AP)$ is the original Kripke model, φ is a desired safety property, for any terminating execution of LazySlicing (K, φ) we have:*

1) If LazySlicing(K, φ) returns true, then we have $S \subseteq \cup_{s \in R}[s]$, where R is a set storing the states explored by LazySlicing(K, φ);

2) Otherwise, LazySlicing(K, φ) returns a counterexample $\tilde{\pi} = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}\tilde{s}_{m2}\cdots\tilde{s}_{mn_m}$, where $\tilde{s}_{mn_m}$ is the state violating φ, then there exists at least one path $\pi = s_1 s_2 \cdots s_k$ in K such that $s_1 \in [\tilde{s}_{11}]$ and $s_k \in [\tilde{s}_{mn_m}]$.

Let $K^0_i = (S^0_i, \rightarrow^0_i, I^0_i, L^0_i, AP^0_i)$ denote the i-th over-approximate slice unfolded by *LazySlicing*(K, φ), where $1 \leqslant i \leqslant n$, n is the total number of over-approximate slices unfolded by *LazySlicing*(K, φ).

In case that *LazySlicing*(K, φ) returns true. If the exploration is finished on over-approximate slice K^0_1, we have $n = 1$ and $R = S^0_1$. According to Corollary 3 the following equation holds.

$$S \subseteq \bigcup_{s \in S^0_1}[s] = \bigcup_{s \in R}[s].$$

Otherwise, the rest states of K^0_q which have not been handled will be explored on the refined over-approximate slice K^0_2 after a spurious counterexample is found on K^0_1. If the verification can be finished on K^0_2, we have $n = 2$. But at this time R consists of two parts: one is the states that have been explored on K^0_1, denoted as $S^0_{1_e}$; the other is the explored states on K^0_2, denoted as $S^0_{2_c}$. Let $S^0_{2_e}$ denote the states on K^0_2 that are not included in R. According to Corollary 3 we have

$$S \subseteq \bigcup_{s \in S^0_2}[s] = \bigcup_{s \in S^0_{2_e}}[s] \cup \bigcup_{s \in S^0_{2_c}}[s].$$

Because states of $S^0_{2_c}$ are covered by states of $S^0_{1_e}$, we have

$$\bigcup_{s \in S^0_{2_c}}[s] \subseteq \bigcup_{s \in S^0_{1_e}}[s].$$

It follows that

$$S \subseteq \bigcup_{s \in S^0_{2_e}}[s] \cup \bigcup_{s \in S^0_{2_c}}[s] \subseteq \bigcup_{s \in S^0_{2_e}}[s] \cup \bigcup_{s \in S^0_{1_e}}[s] = \bigcup_{s \in R}[s].$$

Otherwise, namely the exploration is not finished on K^0_2, *LazySlicing*(K, φ) will enter the third over-approximate slice K^0_3. If it can be finished on K^0_3, then R consists of $S^0_{1_e}$, $S^0_{2_e}$ and $S^0_{3_e}$ (the states explored on K^0_3), and the unexplored states on K^0_3, denoted as $S^0_{3_c}$, are covered by states of $S^0_{1_e}$ and $S^0_{2_e}$. According to Corollary 3 we have

$$S \subseteq \bigcup_{s \in S^0_3}[s] = \bigcup_{s \in S^0_{3_e}}[s] \cup \bigcup_{s \in S^0_{3_c}}[s].$$

Because

$$\bigcup_{s \in S^0_{3_c}}[s] \subseteq \bigcup_{s \in S^0_{1_e}}[s] \cup \bigcup_{s \in S^0_{2_e}}[s],$$

it follows that

$$\begin{aligned} S &\subseteq \bigcup_{s \in S^0_{3_e}}[s] \cup \bigcup_{s \in S^0_{3_c}}[s] \\ &\subseteq \bigcup_{s \in S^0_{3_e}}[s] \cup \bigcup_{s \in S^0_{1_e}}[s] \cup \bigcup_{s \in S^0_{2_e}}[s] \\ &= \bigcup_{s \in R}[s]. \end{aligned}$$

If the exploration cannot terminate on K^0_3, then our algorithm enters K^0_4, the exploration may be finished on K^0_4 or enters K^0_5 and so on. But it will terminate on the n-th over-approximate slice according to Theorem 4 in the worst case $n = |V|$. Finally, we can conclude by induction that *LazySlicing*(K, φ) returns true implies

$$S \subseteq \bigcup_{s \in R}[s].$$

If *LazySlicing*(K, φ) returns a counterexample, let it be $\tilde{\pi} = \tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}\tilde{s}_{m2}\cdots\tilde{s}_{mn_m}$, then according to Theorem 5 we have the path fragment $\tilde{s}_{11}\tilde{s}_{12}\cdots\tilde{s}_{1n_1}\tilde{s}_{21}\tilde{s}_{22}\cdots\tilde{s}_{2n_2}\cdots\tilde{s}_{m1}$ is feasible. According to Theorem 2 path fragment $\tilde{s}_{m1}\tilde{s}_{m2}\cdots\tilde{s}_{mn_m}$ is feasible. So we can safely say $\tilde{\pi}$ is feasible, which means there exists at least one path $\pi = s_1 s_2 \cdots s_k$ in K corresponding to $\tilde{\pi}$ such that $s_1 \in [\tilde{s}_{11}]$ and $s_k \in [\tilde{s}_{mn_m}]$.

8.2 Termination

In this paper we make an assumption that the state space handled by lazy slicing is finite. The following theorem is the sufficient condition that ensures the termination of *LazySlicing*(K, φ).

Theorem 8. *$K = (S, T, I, L, A)$ is a finite Kripke model, φ is the property of interest, LazySlicing(K, φ) terminates.*

Actually the variable set V of K is a finite set, according to Theorem 4 the refinement of *LazySlicing*(K, φ) iterates at most $|V| - 1$ times in the worst case, which also is the reason why the path of lazy slicing algorithm consists of at most $|V|$ path fragments with ascending precisions (Corollary 4). This means the refinement iteration is terminable. In each iteration, cover relations can deal with cycle path, which guarantees the termination of each path of *LazySlicing*(K, φ). So we conclude that *LazySlicing*(K, φ) terminates.

8.3 Savings and Cost

Compared to CEGAR-based slicing, *LazySlicing*(K, φ) has achieved three savings.

First, if a spurious counterexample is found, the dead end state suggests which variable should be added to refine the slice model. Instead of building an entirely new over-approximate slice, lazy slicing refines a local slice by taking the feasible equivalent successors of the feasible prefix of spurious counterexample as the initial states. Refining only the unknown state space makes lazy slicing be able to avoid the repetitive computation of CEGAR-based slicing.

Second, since the refined local slice may contain loops to the state checked before, cover relation is able to identify the loop no matter whether this loop is between states with the same precision or not. This means the work done before is utilized to prove the correctness of the desired property. As we already know there is no error state in the state space explored before.

Third, counterexamples found on an over-approximate slice have to be validated. [18] and [19] identify spurious counterexamples by concretizing the whole counterexample on the original model, which is time-consuming. However, the path of lazy slicing consists of path fragments with ascending precisions, which makes it possible to identify a spurious counterexample by its last path fragment without lose of correctness (according to Theorem 5). The cost of concretizing the path fragments before the last path fragment is reduced.

However, states explored by lazy slicing may have different precisions. So we require an additional field to mark the precision of a state. According to Theorem 4 we know there appear at most $|V|$ different precisions during a verification process. So the extra space cost is only a mark that distinguishes $|V|$ different precisions for each state.

9 Experiments

We have implemented a model checking procedure as a small prototype tool named LSVT (Lazy Slicing-based Verification Tool). LSVT is implemented in Java (JRE 1.6) and consists of five main components: a specification parser which extracts variable set, event set, initial conditions and the given property from the specification; an over-approximate slicing procedure which computes the over-approximate slice with a given precision from the results of specification parser; a satisfiability (SAT) solver for satisfiability checking; a spurious counterexample decision procedure to identify spurious counterexamples; and the lazy slicing exploration procedure that performs verification according to the results of SAT solver and spurious counterexample decision procedure. Though our specification parser and SAT solver are not powerful enough for industrial applications so far, they are sufficient to prove the advantages of lazy slicing compared with CEGAR-based slicing.

The goal of our experimentation is twofold. First we wish to evaluate the feasibility of our approach. In addition, we wish to evaluate the relative performances of lazy slicing compared with CEGAR-based slicing algorithm. We have implemented a basic checking procedure (BC), an incremental slicing checking procedure (ISC)[18] and a lazy slicing checking procedure (LSC) on LSVT. Besides, an improved over-approximate slicing procedure given in Section 7 has been implemented to show the savings of verification on the improved model compared with the one not improved. When LSC performs verification on the improved model, we call it LSCIS (LSC on the improved over-approximate slice model, LSCIS). Our experiments were carried out on a Linux machine with a Pentium® Dual-Core E5200 processor and 2GB memory.

We performed three sets of experiments. The first set experiments were carried out on our own model of the Medical Insurance Audit system of Heilongjiang Province. The others were performed on two benchmarks of BEEM (BEnchmarks for explicit model checkers, http://anna.fi.muni.cz/models/): bridge puzzle and peterson mutual exclusion algorithms.

In the first set of experiments, we built a model of our Medical Insurance Audit system. The primary function of the audit system is to discover the behavior that violates the medical insurance policy of Heilongjiang province by data analysis, and the core business of this system is the audit method system constructed from the medical insurance policy. The model we built describes the payment and settlement link of the ordinary urban workers. The domains of variables, such as payment standard, self-paid ratio, proportion of reimbursement, age of workers and so on, were discretized by data abstraction. The state space of this model has 12 294 reachable states. This experiment is designed to verify the safety properties to ensure that the audit system can discover illegal data as expected. Table 4 summarizes the experimental results of six different properties which are chosen from 23 safety properties (described in propositional logic formula) used in our experiment. There are four rows in the verification results of each property, where $|R|_{\max}$ is the largest size of R in the verification process, *SatNum* denotes the number the SAT solver is called, *Cost* denotes the time cost of a verification process, and *RefineNum* is the number of refinement. Columns BC, ISC, LSC and LSCIS denote the basic checking procedure, the incremental slicing checking procedure, the lazy slicing checking procedure and the lazy slicing checking procedure running on our improved over-approximate slice respectively. The last column shows the verification results of each property.

Table 4. Experimental Results on the Medical Insurance Audit System

ID	Parameter	BC	ISC	LSC	LSCIS	Result
φ_1	$\lvert R\rvert_{\max}$	49	7	7	2	False
	SatNum	49	7	7	2	
	Cost (ms)	759	88	91	21	
	RefineNum	0	0	0	0	
φ_2	$\lvert R\rvert_{\max}$	663	139	89	53	False
	SatNum	663	163	117	61	
	Cost (ms)	1 193	331	225	122	
	RefineNum	0	2	2	1	
φ_3	$\lvert R\rvert_{\max}$	1 622	103	66	26	False
	SatNum	1 622	117	71	26	
	Cost (ms)	2 948	242	105	59	
	RefineNum	0	1	1	0	
φ_4	$\lvert R\rvert_{\max}$	12 294	80	69	69	True
	SatNum	12 299	115	83	83	
	Cost (ms)	28 762	285	223	232	
	RefineNum	0	1	1	1	
φ_5	$\lvert R\rvert_{\max}$	12 294	240	197	101	True
	SatNum	12 299	532	243	153	
	Cost (ms)	27 855	1 944	1 003	707	
	RefineNum	0	3	3	1	
φ_6	$\lvert R\rvert_{\max}$	12 294	514	349	261	True
	SatNum	12 294	961	517	314	
	Cost (ms)	26 733	4 753	901	565	
	RefineNum	0	4	4	2	

We first used BC procedure to perform a primitive verification for the six safety properties on our medical insurance settlement model. Then ISC procedure was applied to perform the incremental slicing verifications. As mentioned earlier, the advantage of ISC is that it allows for much coarser slices yielding smaller state spaces. The experimental results reveal the power of the state space reduction possessed by incremental slicing compared with BC procedure. However, ISC reduces the state space at the cost of additional refinements. Though it is necessary for ISC to rule out spurious counterexamples, it also will result in repeated computing cost at the same time.

Compared to ISC procedure, the experiment results confirm the correctness and the evident improvement of the reduction capability of our lazy slicing. However, lazy slicing cannot always guarantee a remarkable reduction of state space and computing cost. The six properties include a variety of different situations, which can help us observe the performance of lazy slicing from different perspectives. Properties 1∼3 are not satisfied by our testing model while properties 4∼6 are satisfied by our model. LSC gives the same result as ISC and BC, which is guaranteed by Theorem 7. We notice the performance of LSC is roughly the same as ISC in the situation where there is no refinement iteration in a verification process (the experimental results of Property 1), and model checking is finished on the first over-approximate slice in this situation. But the first approximation is often too rough to verify the given property, and refinement is inevitable in most cases. According to the experimental results of Properties 2∼6, it follows that the more refinement iterates, the higher performance LSC achieves. There are two main causes of this situation. First, LSC avoids the repeated computation cost only when refinement happens. Second, the cost of spurious counterexample decision decreases remarkably in proportion to the number of refinements. Another significant improvement is LSCIS, namely, performing lazy slicing on our improved over-approximate slice model, which actually enhances the performance of LSC irrespective of refinement according to experimental results (except in extreme cases that the improved slice is the same as the one not improved).

Bridge puzzle is a benchmark of BEEM about men crossing a bridge. Four men have to cross a bridge at night. The bridge is old and dilapidated and can hold at most two people at a time. There are no railings, and the men have only one flashlight. Any party who crosses, either one or two men, must carry the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, etc. Each man walks at a different speed. If two men cross together, they must walk at the slower man's pace. The problem is whether they can get to the other side in a given time. We generalized this model to different number of men and time limitation. Table 5 compares the run time performance of lazy slicing and incremental slicing (a CEGAR-based slicing algorithm) on bridge puzzle algorithm with different parameters. We checked six different properties on this model. In Column 1, N denotes the number of men and M denotes the maximum time for crossing. The last column provides the scale of the state space of bridge puzzle with regard to N and M.

Table 5. Experimental Results on Bridge Puzzle Algorihm

Parameters			ISC	LSC	LSCIS	$\|S\|$
$N = 4$	φ_7	*SatNum*	28	23	17	3 186
$M = 60$		*Cost* (ms)	253	194	101	
$N = 6$	φ_8	*SatNum*	250	143	121	
$M = 140$		*Cost* (ms)	485	375	298	
	φ_9	*SatNum*	184	111	97	3 186
		Cost(ms)	532	264	191	
	φ_{10}	*SatNum*	393	206	101	
		Cost (ms)	1 641	807	462	
$N = 8$	φ_{11}	*SatNum*	1 823	786	564	96 923
$M = 200$		*Cost* (ms)	10 126	3 878	2 391	
	φ_{12}	*SatNum*	4 703	1 689	1 069	
		Cost(ms)	33 110	12 465	8 398	

In order to investigate the performance of lazy slicing on larger examples, three groups of experiments were carried out on a peterson mutual exclusion algorithm. There are two or more processes reading and/or writing some shared data and the final result depends on who runs precisely. Code sections containing race conditions can be regarded as "critical", because such code can lead to inconsistent data. To avoid inconsistence in critical sections, exclusive access to shared data must be granted. This algorithm was also extended to supply a lager state space. Table 6 provides an overview of the experimental results on six different properties. Parameter N in the first column denotes the number of processes, and E denotes the presence of an artificial error.

Table 6. Experimental Results on Peterson Mutual Exclusion Algorithm

Parameters			ISC	LSC	LSCIS	$\|S\|$
$N = 3$	φ_{13}	*SatNum*	185	111	60	12 498
		Cost (ms)	431	361	143	
	φ_{14}	*SatNum*	379	298	227	
		Cost (ms)	1 043	891	715	
$N = 3$	φ_{15}	*SatNum*	6 523	2 787	802	124 704
$E = 1$		*Cost* (ms)	20 098	13 437	5 046	
	φ_{16}	*SatNum*	10 011	2 270	736	
		Cost (ms)	35 477	11 459	6 101	
$N = 4$	φ_{17}	*SatNum*	56 245	13 841	9 028	1 119 560
		Cost (ms)	238 378	76 229	59 257	
	φ_{18}	*SatNum*	52 046	11 310	3 533	
		Cost (ms)	275 153	62 934	23 546	

The performance difference of lazy slicing and incremental slicing lies in the fact that lazy slicing conserves the achievements that have been done by CEGAR-based slicing before a spurious counterexample has been found. Besides, our improved over-approximate slicing method (in Section 7) is able to provide a more precise slice than incremental slicing, which explains why LSCIS is better than LSC. Tables 5 and 6 also show that the slice state space expanded by LSC (LSCIS) grows obviously slower than the state space considered by ISC according to the number of calls for SAT solver. We also report the relative time difference between our approach and incremental slicing in Fig.8.

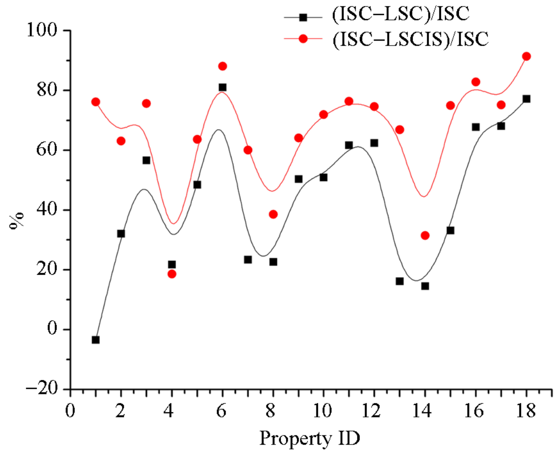


Fig.8. Relative time improvement of LSC and LSCIS w.r.t. ISC.

In Fig.8, the x-axis corresponds to the number of the properties in Tables 4∼6, and y-axis corresponds to the relative time differences (ISC-LSC)/ISC×100 and (ISC -LSCIS)/ISC×100). Fig.8 shows that the computation cost of lazy slicing (LSC) is notably lower than that of incremental slicing, and this reduction ability is strengthened by performing lazy slicing on an improved over-approximate slice (LSCIS). Note that the improvement of LSC and LSCIS is relative to the given property and the dependent relationship between variables of the model. Generally speaking, the fewer the number of variables in the desired property, the stronger the ability to reduce the state space.

10 Conclusions

We propose lazy slicing to eliminate the repeated computation cost of CEGAR-based slicing methods in this paper. Our algorithm reuses the work done previously which avoids traversing the known correct state space. By refining and exploring only a local slice, we benefit from the previous runs because the state space explored before is sufficient to prove the property of interest. Spurious counterexample decision can also take advantage of the path with ascending precisions of lazy slicing. Our improved over-approximate slicing method rules out additional behavior introduced by ISC which enhances the performance of LSC significantly. Experimental results show that LSC procedure, especially

LSCIS, scales much better to large systems compared with CEGAR-based slicing without loss of correctness.

We have three main directions for future research. First, as the object of LTL model checking is to find an acceptable trace on the automata which is the product of the given model and the desired property, our ultimate goal is to apply LSCIS to LTL model checking. Second, how to find a minimal variable set to decrease the refinement iterations is a promising work for us, and related efforts has been made in [19]. Third, counterexamples, especially long counterexamples, which are utilized to locate error positions, are difficult to understand. Much work[26-28] has been done in an effort to deal with this problem. How to understand counterexamples, especially counterexamples with ascending precisions is still a challenging task.

Acknowledgement We would like to thank Yan-Mei Li and Da-Peng Lang for their helpful comments as well as Yuan Cheng and Guo-Feng Liu for useful discussions on applying LSCIS to LTL model checking. We are especially grateful to Akinfenwa Olusheye Akin and Xiao-Xue Wang for proofreading the paper and Han Wang for providing the source code of an SAT solver.

References

[1] Clarke E M, Grunberg O, Peled D A. Model Checking. Cambridge, Massachusetts: The MIT Press, 1999, pp.5-19.

[2] Clarke E M, Emerson E A, Sifakis J. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 2009, 52(11): 74-84.

[3] Weiser M. Program slicing. In *Proc. the 5th Int. Conf. Software Engineering*, March 1981, pp.439-449.

[4] Brückner I, Wehrheim H. Slicing an integrated formal method for verification. In *Proc. ICFEM*, Nov. 2005, pp.360-374.

[5] Clarke E M, Fujita M, Rajan S P, Reps T W, Shankar S, Teitelbaum T. Program slicing of hardware description languages. In *Proc. the 10th IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods*, September 1999, pp.298-312.

[6] Hatcliff J, Dwyer M B, Zheng H. Slicing software for model construction. *Higher-Order Symbol. Comput.*, 2000, 13(4): 315-353.

[7] Yatapanage N, Winter K, Zafar S. Slicing behavior tree models for verification. In *Proc. the 6th IFIP Int. Conf. Theoretical Computer Science*, September 2010, pp.125-139.

[8] Dwyer M B, Hatclifi J, Hoosier M, Ranganath V, Robby, Wallentine T. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proc. the 12th TACAS*, March 25-April 2, 2006, pp.73-89.

[9] Godefroid P. Using partial orders to improve automatic verification methods. In*Proc. the 2nd International Workshop on Computer Aided Verification*, June 1990, pp.176-185.

[10] Emerson E A, Sistla A P. Symmetry and model checking. *Formal Methods in System Design*, 1996, 9(1): 10-131.

[11] Miller A, Donaldson A, Calder M. Symmetry in temporal logic model checking. *ACM Computing Surveys* (*CSUR*), 2006, 38(3): Article No.8.

[12] Heitmeyer C, Kirby J, Labaw B, Archer M, Bharadwaj R. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 1998, 24(11): 927-948.

[13] Bharadwaj R, Heitmeyer C L. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 1999, 6(1): 37-68.

[14] Hong H S, Lee I, Sokolsky O. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proc. the 15th SCAM 2005*, September 30-Oct.1, 2005, pp.25-34.

[15] Brückner I, Dräger K, Finkbeiner B, Wehrheim H. Slicing abstractions. *Fundam. Inf.*, 2008, 89(4): 369-392.

[16] Holzmann G J. Personal Communication. Oct. 2005.

[17] Clarke E M, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 2003, 50(5): 752-794.

[18] Wehrheim H. Incremental slicing. In *Proc. the 8th ICFEM 2006*, November 2006, pp.514-528.

[19] Sabouri H, Sirjani M. Slicing-based reductions for rebeca. *Electronic Notes in Theoretical Computer Science*, 2010, 260(1): 209-224.

[20] Henzinger T A, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In *Proc. the 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, Jan. 2002, pp.58-70.

[21] Graf S, Saïdi H. Construction of abstract state graphs with PVS. In *Proc. the 9th CAV*, June 1997, pp.72-83.

[22] Jung Y, Kong S, Wang B, Yi K. Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In *Proc. the 11th Int. Conf. Verification, Model Checking, and Abstract Interpretation*, January 2010, pp.180-196.

[23] Podelski A, Wies T. Counterexample-guided focus. In *Proc. the 37th POPL*, Jan. 2010, pp.249-260.

[24] Tonetta S. Abstract model checking without computing the abstraction. In *Proc. the 2nd World Congress on Formal Methods*, November 2009, pp.89-105.

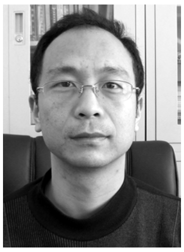
[25] Baier K C J, Principles of Model Checking. Cambridge, Massachusetts: The MIT Press, 2008, pp.468-595.

[26] Groce A, Kroening D, Lerda F. Understanding counterexamples with explain. *In Proc. the 16th International Conference on Computer Aided Verification*, July 2004, pp.453-456.

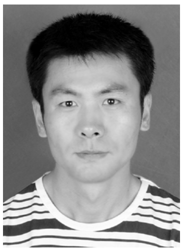
[27] Gastin P, Moro P, Zeitoun M. Minimization of counterexamples in SPIN. In *Proc. the 11th International SPIN Workshop on Model Checking Software*, April 2004, pp.92-108.

[28] Beer I, Ben-David S, Chockler H, Orni A, Trefler R. Explaining counterexamples using causality. In *Proc. the 21st CAV*, June 26-July 2, 2009, pp.94-108.

Shao-Bin Huang is a professor of College of Computer Science and Technology, Harbin Engineering University (HEU), Harbin, China, and the director of Distributed Computing and Simulation Laboratory of HEU. He received his B.S. and M.S. degrees in applied mathematics from Jilin University, China, in 1986 and 1989 respectively, and the Ph.D. degree in control theory and control engineering from Harbin Engineering University, China, in 2004. His research interest focuses on distributed computing and simulation and automatic verification and analysis of complex distributed software systems. His other interest areas include grid computing, massive date processing theory and technology in grid environment, and model checking.



Hong-Tao Huang is a Ph.D. candidate in computer science and technology of Harbin Engineering University. He received his B.S. degree from Hefei University of Technology, China, in 2004, and M.S. degree from Harbin Engineering University in 2009, both in computer application technology. His research interest focuses on model checking, formal method and software engineering.



Zhi-Yuan Chen is a Ph.D. candidate in computer science and technology in Harbin Engineering University. He received his B.S. and M.S. degrees from Jilin University in 2002 and 2005 respectively, both in computational mathematics. He is currently a lecturer in the College of Computer Science and Technology, Harbin Engineering University. His research interests include model checking and modal logic.

Tian-Yang Lv received the Ph.D. degree in computer application technology from Jilin University in 2007. In 2010, he received the 10th Science and Technology Award for Youth of Harbin. He has published over 50 papers, of which over 20 are indexed by SCI, EI. He has developed and applied the advanced intelligent analysis technology in auditing over 20 million attendees of Chinese social insurance.

Tao Zhang is a Ph.D. candidate in computer science and technology in Harbin Engineering University. He received the M.S. degree in computer application technology from the Institute of Computing Technology, Northeast Forestry University, China, in 2007. His research interests include model checking, software engineering, and formal method.