Model reduction using the orthogonality between overapproximate slicing and abstract

Hongtao Huang*, Shaobin Huang†, Zhiyuan Chen‡, Tao Zhang§
* College of Computer Science and Technology
Hargin Engineering University
Harbin, Heilongjiang, 150001, China
horntau@gmail.com, huangshaobin@hrbeu.edu.cn, chenzhiyuan@hrbeu.edu.cn, zhangtaohrbeu@163.com

***Abstract*—The orthogonality between static slicing and abstract method has been used to furtherly reduce the state space in model checking. However, static slicing can not always guarantee a slicing model with an desired size. This paper proposes a new approach which compute the over approximate slicing of an abstract state graph other than a counterpart of a static slicing. An overapproximate slicing is obtained by only considering the data dependence relations between predicates, which will always lead to a slice with an ideal size for verification. Though the overapproximate slice only has a weak property resistance power, it is an super set of the abstract state graph, which guarantees if a property ϕ is satisfied on the overapproximate slice, then the original specification is a model of ϕ. And if there appears a spurious counterexample, then it increases the precision of the overapproximate slice by refinement to keep the verification cost as low as possible. We also provide sufficient proof for the correctness of our method. The experimental result shows that our method improves the scalability of model checking remarkably and scales better to a larger system.**

I. Introduction

Model checking[1] is a proven successful technique for verifying hardware and software. 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[2], this phenomenon is known as state space explosion[3], [4] and is the main obstacle for applying model checking to software systems of industrial size. Program slicing[5] is a program analysis technique that has been proven to be useful in a variety of software engineering applications, such as program debugging, testing, understanding, maintenance, metrics, and reuse. Recently, program slicing has also been applied to state space reduction for model checking, and it can eliminate the portion which is irrelevant to a slicing criterion from a software specification by reachability analysis. A lot of studies[6], [7], [8], [9] show that program slicing is an effective technique for reducing the state space in model checking.

Model checking works only for finite state systems, but most software systems have infinitely many states due to unbounded variables and unbounded control structures. Program slicing is able to extract the portion which has a direct or indirect relation with slicing criterian from a software specification, and is essentially a method of rebuilding the state space by removing the irrelevant variables. However, slicing can not convert an infinite state space system to a finite one. Abstract[10] is the most widely used state space reduction method based on abstract interpretation[11], it is capable of reducing a potentially infinite state space to the finite set of valuations of a tuple of state predicates. Abstract rebuilds a state space by the abstract variables which are abstracted from concrete variables by data domain division and the logical relations among of concrete variables. Abstract which erects a bridge between a infinite state space and the finite one is a reduction method that is independent of program slicing. Therefore, program slicing and abstract can be utilized simultaneously to reduce state space.

M. Dwyer indicates that the relative benefits of state-of-the-art Java slicing techniques with state-of-the-art implementations of other well-known model reduction techniques such as abstract, partial order reductions[12], symmetry reduction[13], [14] , and demonstrates that reductions provided by slicing are largely orthogonal to the effect of these other techniques[15]. For example, document[16], [17] perform slicing after data abstract has been done to minimize the four-variable model. H. Seok Hong introduces a notion of abstract slicing[18] which is an approach to program slicing based on abstract interpretation and symbolic model checking. Abstract slicing extends static slicing with predicates and constraints by using as the program model and abstract state graph, which is obtained by applying predicate abstraction to a program. This leads to a program slice that is more precise and smaller than its static counterpart. Ingo Brückner presents a model checking procedure for infinite state concurrent systems which interleaves automatic abstraction refinement with slicing[19]. Abstract splits states according to new predicates obtained by Craig interpolation, while slicing removes irrelevant states and transitions from the abstraction. Abstract and slicing work together and complement each other which makes model checking scale to a larger system.

These methods reduces the state space significantly by the help of the orthogonality between abstract and static slicing. The orthogonality reduction method yield a smaller state space than the one produced by abstract or slicing alone. However, existing experience with static slicing for model reduction is sometime inconclusive. Holzmann's experience shows that static slicing in Spin usually does not yield much reduction for realistic Promela design models[20]. The main reason is the compression capability of static slicing depends on not only the slicing criterion but also the dependencies between variables. That's why it does not always guarantee a slicing model with a desired size. Compared with static slicing, overapproximate slicing is able to reduce the given model to an ideal size. In this paper, we propose a model reduction method that utilizes the orthogonality between abstract and overapproximate slicing. Our orthogonality method reduces the state space by computing the overapproximate slicing of an abstract model, which achieves a further reduction of system model and improves the scalability of model checking.

II. Preliminaries

We use the abstract approach proposed by S. Graf[10] to extract the abstract model from original specification. The result of abstract is an abstract state graph.

Definition 1: (Abstract State Graph) Let $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$ be the abstract state graph of an original specification, where

- $\mathcal{S}$ is a abstract state set, $\mathcal{S} \subseteq b_1 \times b_2 \times \cdots \times b_l$, $b_i \in B$ denotes a predicate φ_i of predicates set Ψ, $|\Psi| = l$, $1 \leq i \leq l$,
- $\mathcal{E}$ is an event set, $e \in \mathcal{E}$ is a conditional assignment $g(\overline{B_g}) \mapsto A$, where $A = \{b := exp \mid b \in B_a\}$ is a set of assignment expressions, $B_g, B_a \in 2^B$, exp is an evaluating expression defined over B, $g(\overline{B_g})$ is a guard, it is a boolean expression defined over B_g,
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ is an abstract transition relation, for all $\tau \in \mathcal{T}$, there exists a event $e \in \mathcal{E}$ and two states $s, s' \in \mathcal{S}$ such that the guard of e is true on state s and the evaluating expressions of e yields s', and
- $\mathcal{I} \subseteq \mathcal{S}$ is a set of abstract initial states.

Abstract state graph is an directed graph of an abstraction of the original specification, the node of an abstract state graph denotes an abstract state, an arc denote an abstract transitions between two abstract states, and the weight of an arc is an event which causes the transition.

Definition 2: (Initial path) A path of $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$ is an alternating sequence of states and events: $\pi = s_0 e_0 s_1 e_1 \cdots$, where $s_i e_i s_{i+1} \in \mathcal{T}$, π is a initial path if and only if $s_0 \in \mathcal{I}$.

A property ϕ is a predicate defined over B, ϕ characterizes the correctness criterion of $\mathcal{A}$, if there is no state that violates ϕ is discovered in $\mathcal{A}$, then $\mathcal{A}$ is correct, else we say $\mathcal{A}$ does not satisfy ϕ. The model checker returns a finite error path $\pi_e = s_0 e_0 s_1 e_1 \cdots s_{k-1} e_{k-1} s_k$ in the case $\mathcal{A}$ does not satisfy ϕ, where π_e is an initial path and s_k violates ϕ. π_e is concretizable if there exists a path in the original model corresponding to π_e.

Model checking is performed on the abstract model $\mathcal{A}$, the purpose of model checking is to prove $\mathcal{A}$ satisfies ϕ or to find an concretizable error path as the proof that the original specification is not a model of ϕ. We say $\mathcal{A}$ is a sound abstract model of the original model if and only if there exists a concretizable error path in $\mathcal{A}$ when the original specification is not correct. The soundness of $\mathcal{A}$ has been proved in document [19] and [21]. It has also been proven that static slicing is an effective way in model reduction working together with abstract[18], [19], we introduce overapproximate slicing which is orthogonal to abstract to furtherly reduce the state space in model checking.

III. Overapproximate slicing of abstract model

Compared with static slicing, overapproximate slicing is able to reduce the given model to an ideal size. In this section, we reduce the abstract state graph by overapproximate slicing. The selection of a slicing criterion is the key step for computing a slicing of an abstract model.

Definition 3: (Slicing criterion) The slicing criterion of an abstract state graph $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$ is a tuple $\mathcal{C}(\mathcal{I}, \mathcal{V})$, $\mathcal{V}$ consists of the variables contained in the given property, let ϕ be the property to be verified, then $\mathcal{V} = vars(\phi)$.

We take the initial state set $\mathcal{I}$ of $\mathcal{A}$ as the program points, and $\mathcal{V}$ is the variable set with respect to the given property. The slicing of $\mathcal{A}$ is the part of $\mathcal{A}$ which affects $\mathcal{I}$ with respect to $\mathcal{V}$. The difference between overapproximate slicing and static slicing is that the control dependence relations between varibles are neglected when computing overapproximate slicing, that's why overapproximate slicing can always guarantee a slicing model with a desired size. We give the definitions of data dependence and control dependence relation between variables as follows.

Definition 4: (Data dependence) b_1 is data dependent on b_2 if and only if $b_1 := exp$ belongs to $A(e)$ and $b_2 \in vars(exp)$, where e is an event, $vars(exp)$ denotes the variable set of exp.

Definition 5: (Control dependence) b_1 is control dependent on b_2 if and only if $b_1 := exp$ belongs to $A(e)$ and $b_2 \in vars(G(e))$, where e is an event, $G(e)$ is the guard of e, $vars(G(e))$ denotes the variable set of $G(e)$.

Let $\mathcal{A}_s = (\mathcal{S}_s, \mathcal{E}_s, \mathcal{T}_s, \mathcal{I}_s)$ be the overapproximate slicing of $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$ with respect to $\mathcal{C}(\mathcal{I}, vars(\phi))$, where ϕ is the given property. $\mathcal{A}_s$ can be obtained by computing the least fix point of $\mathcal{S}$, namely, compute the part of $\mathcal{A}$ that has an influence on the variable set of slicing criterion starting from $\mathcal{I}$ iteratively, until it reaches the fixed point. There are two key steps to compute $\mathcal{A}_s$. We first compute the variable set B_s which generates the abstract state space of $\mathcal{A}_s$ by dependency analysis, then compute the event set $\mathcal{E}_s$ which generates $\mathcal{T}_s$ according to B_s. We compute B_s according to the data dependence relation.

Let $B_{s_0} = var(\phi)$, then we compute the variables which are data dependent on B_{s_0}, let D_0 denote the set of these variables, then we have

$$\forall e \in \mathcal{E}. \forall a \in A(e). (tar(a) \in B_{s_0} \Rightarrow D_0 = D_0 \cup vars(exp(a)))$$

where $A(e)$ denotes the assignment expression set of e, $tar(a)$ is the target variable of a, $exp(a)$ is the evaluating expression of a. If D_0 is not empty, then we have

$$B_{s_1} = B_{s_0} \cup D_0$$

Next, we compute D_1 the same way as D_0.

$$\forall e \in \mathcal{E}. \forall a \in A(e). (tar(a) \in B_{s_1} \Rightarrow D_1 = D_1 \cup vars(exp(a)))$$

If D_1 is not empty, then we have

$$B_{s_2} = B_{s_1} \cup D_1$$

This computation will terminate at the ith iteration when $D_i = \emptyset$, where $0 \leq i \leq k$, $k = |B|$. Knaster-Tarski theorem[22], [23] guarantees this computation iterates at most k times. Assume it terminates at the ith iteration, then we have $B_s = B_{s_i}$.

$\mathcal{E}_s$ consists of events which are chosen from $\mathcal{E}$ according to B_s. Events that satisfy $e \in \mathcal{E} \wedge a \in A(e) \wedge tar(a) \in B_s$ are added to $\mathcal{E}_s$, where A is the assignment expression set of e. For all $e \in \mathcal{E}_s$ the evaluating expressions which satisfy $a \in A(e) \wedge tar(a) \notin A(e)$ are removed from $A(e)$. Then we handle the guards of events of $\mathcal{E}_s$ in order to eliminate the effect of control dependence relations.

A method has been proposed by H. Wehrheim[24] to compute $\mathcal{E}_s$, it neglects the whole guard of an event if the guard contains variables not belonging to B_s. This method may introduce too many additional behaviors into the overapproximate slicing because the method of handling the guard is too rough. In this section, we proposed an optimized approach to compute $\mathcal{E}_s$, the main idea of our method is to obtain a more accurate slicing model by the means of preserving guard conditions related to B_s. In our method, the guard of an event is normalized to the disjunctive normal form. Let $G(e)$ denote the guard of an event $e \in \mathcal{E}$, $C(G(e))$ denote the clause set of $G(e)$, $c \in C(G(e))$ is a conjunction of literals which is composed of predicates or its negations. We first recalculate the literals of c for all $c \in C(G(e))$ as follows.

$$L'(c) = \{l \mid l \in L(c) \wedge vars(l) \subseteq B_s\}$$

where $L(c)$ denotes the literal set of c and $L'(c)$ denotes the new literal set of c. Let c' be the new version of c, then c' consists of the elements of $L'(c)$.

$$c' = \bigwedge_{l \in L'(c)} l$$

Let $C'(G(e)) = \{c' \mid c' \neq \emptyset \wedge c \in C(G(e))\}$, where $C'(G(e))$ is a new version of $C(G(e))$. Then we have

$$G'(e) = \bigvee_{c \in C'(G(e))} c$$

where $G'(e)$ is the new version of $G(e)$. Finally, we replace $G(e)$ with a corresponding $G'(e)$ for all $e \in \mathcal{E}_s$.

$\mathcal{A}_s$, the overapptoximate slicing of $\mathcal{A}$ with regard to $(\mathcal{I}, vars(\phi))$, can be induced by B_s and $\mathcal{E}_s$.

***Definition** 6:* (Overapproximate slicing) $\mathcal{A}_s = (\mathcal{S}_s, \mathcal{E}_s, \mathcal{T}_s, \mathcal{I}_s)$ is the overapproximate slicing of an abstract state graph $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$, where

- $\mathcal{S}_s \subseteq b_{s_1} \times b_{s_2} \times \cdots \times b_{s_m}$, where $B_s = \{b_{s_1}, b_{s_2}, \cdots, B_{s_m}\} \subseteq B$, and $b_{s_1}, b_{s_2}, \cdots, B_{s_m}$ has a compatible order with $b_1, b_2, \cdots, b_l$,
- $\mathcal{E}_s$ is a set of events,
- $\mathcal{T}_s \subseteq \mathcal{S}_s \times \mathcal{E}_s \times \mathcal{S}_s$ is a transition set,
- $\mathcal{I} = \{s \mid s = P_{B_s}(s') \wedge s' \in \mathcal{I}\}$, where $P_{B_s}(s')$ denote the vector which is projected from s' on B_s.

The main difference between overapproximate slicing and static slicing is overapproximate slicing guarantees an ideal reduction on an abstract state graph. We consider a simple example to clarify how the overapproximate slicing works.

***Example** 1:* Let $\mathcal{A}_1 = (\mathcal{S}_1, \mathcal{E}_1, \mathcal{T}_1, \mathcal{I}_1)$ be an abstract state graph of a system, $B_1 = \{b_1, b_2, b_3\}$ be the variable set which generates $\mathcal{S}_1$, and the initial value of each variable in B_1 is 0, the event set $\mathcal{E}_1$ is depicted in table I, the property of interest is $\phi_1 = \neg(b_2 = 1 \wedge b_3 = 1)$.

TABLE I
THE EVENTS OF $\mathcal{E}_1$

No.	Event
e_1	$b_1 = 0 \mapsto b_1 = 1$
e_2	$b_2 = 1 \wedge b_1 = 1 \mapsto b_1 = 0$
e_3	$b_1 = 1 \wedge b_2 = 0 \mapsto b_2 = 1$
e_4	$b_3 = 1 \wedge b_2 = 1 \mapsto b_2 = 0$
e_5	$b_1 = 1 \wedge b_2 = 1 \wedge b_3 = 0 \mapsto b_3 = 1$
e_6	$b_3 = 1 \mapsto b_3 = 0$

According to the conditions given in example 1, we can draw a abstract state graph as shown in figure 1.

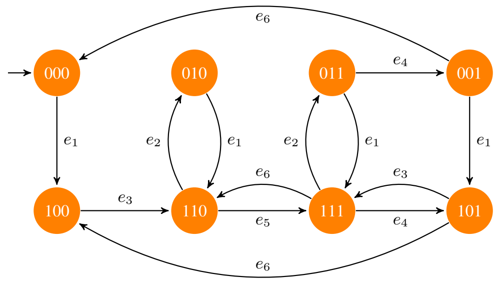

Fig. 1. Abstract state graph of example 1

We compute the overapproximate slicing of $\mathcal{A}_1$ with regard to $\mathcal{C}(\{000\}, \{b_2, b_3\})$ as follows. First we compute B_s which generates the slicing state space, because no variable in B_1 is introduced by data dependence relation into $\{b_2, b_3\}$, so we have $B_s = \{b_2, b_3\}$; Then we compute $\mathcal{E}_s$ which generates the transition set of the sliced abstract state graph, 4 transitions e_3, e_4, e_5, e_6 are added to $\mathcal{E}_s$ according to B_s. Next, the guards of these events will be handle using the method given in this section. Finally, we get $\mathcal{E}_s$ as shown in table II.

TABLE II
THE EVENT SET OF $\mathcal{E}_s$

No.	Event
e'_3	$b_2 = 0 \mapsto b_2 = 1$
e'_4	$b_3 = 1 \wedge b_2 = 1 \mapsto b_2 = 0$
e'_5	$b_2 = 1 \wedge b_3 = 0 \mapsto b_3 = 1$
e'_6	$b_3 = 1 \mapsto b_3 = 0$

The overapproximate slicing abstract state graph of $\mathcal{A}_1$ introduced by B_s and $\mathcal{E}_s$ with regard to $\mathcal{C}(\{000\}, \{b_2, b_3\})$ is shown in figure 2.

The size of the overapproximate slicing of $\mathcal{A}_1$ is only a half of $\mathcal{A}_1$, the reason is a variable b_1 was ruled out. As a

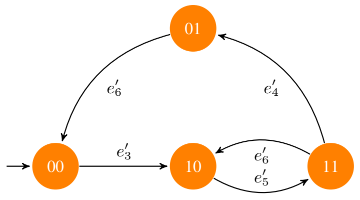

Fig. 2. The overapproximate slicing abstract state graph of example 1

comparision, the slicing state space will be the same as $\mathcal{A}_1$ if we use static slicing, the method of static slicing can be bound in document[24]. However, overapproximate slicing reduces the state space of an abstract state graph exponentially at the cost of sacrificing the strong property resistance power. We prove this property by stutter-equivalence paths between an abstract state graph and its overaprroximate slicing.

***Definition** 7:* (Stutter-equivalence path) $\mathcal{A}_i = (\mathcal{S}_i, \mathcal{E}_i, \mathcal{T}_i, \mathcal{I}_i)$ is a abstract state graph, where $\mathcal{S}_i$ is generated by B_i, π_i is a path of $\mathcal{A}_i$, $i = 1, 2$, let $\mathcal{W} \subseteq B_1 \cap B_2$, $B_1 \cap B_2 \neq \emptyset$, π_1 and π_2 are $\mathcal{W}$-stutter-equivalent, denoted $\pi_1 \triangleq_{\mathcal{W}} \pi_2$, if π_i satisfies the following conditions.

$$\pi_1|_{\mathcal{W}} = \underbrace{s^1_{0_0} s^1_{0_1} \cdots s^1_{0_{n_0}}}_{\substack{P_{\mathcal{W}}(s^1_{0_i}) = s_0 \\ 0 \le i \le n_0}} \underbrace{s^1_{1_0} s^1_{1_1} \cdots s^1_{1_{n_1}}}_{\substack{P_{\mathcal{W}}(s^1_{1_i}) = s_1 \\ 0 \le i \le n_1}} \underbrace{s^1_{2_0} s^1_{2_1} \cdots s^1_{2_{n_2}}}_{\substack{P_{\mathcal{W}}(s^1_{2_i}) = s_2 \\ 0 \le i \le n_2}} \cdots$$

$$\pi_2|_{\mathcal{W}} = \underbrace{s^2_{0_0} s^2_{0_1} \cdots s^2_{0_{m_0}}}_{\substack{P_{\mathcal{W}}(s^2_{0_i}) = s_0 \\ 0 \le i \le m_0}} \underbrace{s^2_{1_0} s^2_{1_1} \cdots s^2_{1_{m_1}}}_{\substack{P_{\mathcal{W}}(s^2_{1_i}) = s_1 \\ 0 \le i \le m_1}} \underbrace{s^2_{2_0} s^2_{2_1} \cdots s^2_{2_{m_2}}}_{\substack{P_{\mathcal{W}}(s^2_{2_i}) = s_2 \\ 0 \le i \le m_2}} \cdots$$

***Theorem** 1:* Let $\mathcal{A}_s = (\mathcal{S}_s, \mathcal{E}_s, \mathcal{T}_s, \mathcal{I}_s)$ be an overapproximate slicing of an abstract state graph $\mathcal{A} = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{I})$ w.r.t. $\mathcal{C}(\mathcal{I}, vars(\phi))$, for all initial path π_1 of $\mathcal{A}$ there exists an initial path π_2 of $\mathcal{A}_s$ such that $\pi_1 \triangleq_{B_s} \pi_2$.

Proof: Assume that $\pi_1 = s_{10}e_{10}s_{11}e_{11}s_{12}e_{12}\cdots$, then there exists a path $\pi_2 = s_{20}\cdots$ of $\mathcal{A}_s$ such that $s_{20} = P_{B_s}(s_{10})$. Let e_{1i} be the ith event of π_1, s_{2j} be the jth state of π_2, and $s_{2j} = P_{B_s}(s_{1i})$. If there does not exist a corresponding version of e_{1i} in $\mathcal{E}_s$, then the assignment expressions of e_{1i} do not change the value of variables in B_s, it follows that $P_{B_s}(s_{1(i+1)}) = S_{2j}$; If e_{1i} has a corresponding version in $\mathcal{E}_s$, let it be e_{2j}, then s_{2j} can transit to a state $s_{2(j+1)}$, because $G(e_{2j})$ only has relation to variables in B_s, so $G(e_{1j})$ can be satisfied on s_{1i} implies $G(e_{2j})$ is satisfied on s_{2j}, besides, e_{1i} and e_{2j} have the same effect on changing the values of variables in B_s, and the values of variables in B_s on s_{1i} are the same as s_{2j}, so we have the values of variables in B_s on $s_{1(i+1)}$ are the same as $s_{2(j+1)}$, i.e., $s_{2(j+1)} = P_{B_s}(s_{1(i+1)})$. So we can conclude by induction that $\pi_1 \triangleq_{B_s} \pi_2$. ■

$\mathcal{W}$-stutter-equivalence only guarantees if $\mathcal{A}_s$ satisfies ϕ then $\mathcal{A}$ satisfies ϕ. That means if the model checker proves ϕ is correct on $\mathcal{A}_s$, then $\mathcal{A}$ is a model of ϕ, so the verification is finished. But in the situation there is a counterexample on $\mathcal{A}_s$, we have to determine whether the counterexample can be concretized or not[21]. If the counterexample is an spurious one, then we refines the overapproximate slicing and continues to verify the given property on the refined slicing, this is the slicing-verification-refine iteration[24]. In example 1, we can find a counterexample which can be concretized by performing model checking on the overapproximate slicing of $\mathcal{A}_1$, which saves a significant cost of verification than performing model checking on $\mathcal{A}_1$. Algorithm 1 shows the main steps of our overapproximate slicing computation.

Algorithm 1 OverapproximateSlicing($\mathcal{A}, \phi$)

```
Require: An abstract state graph $\mathcal{A}$ and a given property $\phi$
Ensure: Compute the overapproximate slicing of $\mathcal{A}$ w.r.t. $\mathcal{C}(\mathcal{I}, vars(\phi))$
 1. $B_s := B_0 := vars(\phi)$;
 2. $flag := ture$;
 3. while $flag = ture$ do
 4.     for every $e \in \mathcal{E}$ do
 5.         for for every assginment expression $a \in A(e)$ do
 6.             if $tar(a) \in B_0$ then
 7.                 $B_s := B_s \cup vars(exp(e))$;
 8.             end if
 9.         end for
10.     end for
11.     if $B_s = B_0$ then
12.         $flag := false$;
13.     else
14.         $B_0 := B_s$;
15.     end if
16. end while
17. $\mathcal{E}_s = \emptyset$;
18. for every $e \in \mathcal{E}$ do
19.     if $\exists a \in A(e).(tar(a) \in B_s)$ then
20.         generate an event $e'$, let $A(e') := \{a | a \in A(e) \wedge tar(a) \in B_s\}$;
21.         if $vars(G(e)) \subseteq B_s$ then
22.             $G(e') := G(e)$;
23.         else
24.             $G(e') := G'(G(e'))$; $\{G'(G(e'))$ is a new version of $G(e')\}$;
25.         end if
26.     end if
27.     $\mathcal{E}_s := \mathcal{E}_s \cup \{e'\}$;
28. end for
29. construct an overapproximate slicing induced by $B_s$ and $\mathcal{E}_s$;
```

IV. Implementation and Experimentation

We have implemented a prototype model checking procedure to evaluate the feasibility and efficiency of our method. This prototype model checker consists of a specification parser, a static slicing procedure, a overapproximate slicing procedure, a SAT prover for satisfiability checking. The goal of our experimentation is to compare the efficiency of overapproximate slicing and static slicing when they are orthogonal to abstract. The experiment was carried out on a PC with a Pentium(R) Dual-Core E5200 processor, a 2 Gbyte memory and a Ubuntu 10.10 Linux OS. 7 different safety properties were verified on a medical insurance settlement specification. Table III summarizes the experimental results of 7 different safety properties used in our experiment. There are five columns in table III, where PN is property number, Vars is the number of variables contained in a property. We compare the results of Overapptoximate Slicing orthogonal to Abstraction (OSA) algorithm with the results of Static Slicing orthogonal to Abstraction (SSA) algorithm on the total state number, time and result in different verifications.

TABLE III
EXPERIMENTAL RESULTS

PNo	Vars	StateNum		Time(ms)		Result	
		SSA	OSA	SSA	OSA	SSA	OSA
1	2	32	16	193	127	Satisfied	
2	4	128	128	359	377	Satisfied	
3	4	186	105	433	314	Counterexample	
4	8	8192	1024	14528	2339	Satisfied	
5	8	5119	875	11506	1973	Counterexample	
6	10	16384	4096	25120	6982	Satisfied	
7	10	7167	2744	16093	6140	Counterexample	

The experiment results confirmed the correctness and the evident improvement of the reduction capability of OSA. These 7 properties include a variety of different situations, which can help us observe the performance of OSA from different perspectives. The results of these properties are sorted in an ascending order on the variable number, and property 1, 2, 4, 6 are satisfied by our testing model, while property 3, 5, 7 are negative. OSA gives the same verification result as SSA as we expect. We noticed the performance of OSA is roughly the same as SSA on property 2, this is caused by the weak property resistance power of overapproximate slicing, which will result in slicing refinement to eliminate spurious counterexamples. Overapproximate slicing assume that the model checking can be finished on the the slicing model which is the super set of the original model but has an ideal size. If the over approximate slicing is too rough to prove the given property, then it increases the precision of the slicing via refinement, and the upper bound of slicing refinement is the static slicing, this is why the state number of OSA is no more than the number of SSA. But OSA may take more time than SSA when the slicing models on which they finished verification respectively have the same size. Because if the overapproximate slicing is too rough to prove a given property, OSA has to refine the slicing model, then restarts the model checking process on the refined overapproximate slicing (property 2). Conclusively, the slicing abstract state space constructed by our method grows slower than the one constructed by static slicing, and OSA improves the state space reduction ability of SSA when they are orthogonal to abstract.

V. CONCLUSION

This paper proposes a state space reduction method that utilizes overapproximate slicing technique to reduce the abstract state graph. We described the notion of overapproximate slicing to instead of static slicing which is orthogonal to abstract method. We compute a overapproximate slicing of the abstract state graph by neglecting the control dependence relations of the guard, which guarantees a significant reduction of the abstract model. The experimental result shows that our method scales much better to large systems compared with static slicing.

ACKNOWLEDGMENT

This work is sponsored by the National Natural Science Foundation of China under grant number 60873038 and the National Key Project of Scientific and Technical Supporting Programs under grant number 2009BAH42B02.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. The MIT Press, 1999.

[2] E. M. Clarke, E. A. Emerson, and J. Sifakis, “Model checking: algorithmic verification and debugging,” *Commun. ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[3] F. J. Lin, P. M. Chu, and M. T. Liu, “Protocol verification using reachability analysis: the state space explosion problem and relief strategies,” in *Proceedings of the ACM workshop on Frontiers in computer communications technology*, 1987, pp. 126–135.

[4] A. Valmari, “The state explosion problem,” *Lectures on Petri Nets: advances in Petri Nets. Basic models*, pp. 4–29, 1998.

[5] M. Weiser, “Program slicing,” in *Proceedings of the 5th international conference on Software engineering*, 1981, pp. 439–449.

[6] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, “Program slicing of hardware description languages,” *Correct Hardware Design and Verification Methods*, pp. 72–72, 1999.

[7] M. B. Dwyer and J. Hatcliff, “Slicing software for model construction,” *Proceedings of Partial Evaluation and Semantic-Based Program Manipulation (PEPM’99)*, pp. 105–118, 1999.

[8] I. Brückner and H. Wehrheim, “Slicing an integrated formal method for verification,” *Formal Methods and Software Engineering*, pp. 360–374, 2005.

[9] N. Yatapanage, K. Winter, and S. Zafar, “Slicing behavior tree models for verification,” *Theoretical Computer Science*, pp. 125–139, 2010.

[10] S. Graf and H. Saïdi, “Construction of abstract state graphs with PVS,” in *Computer Aided Verification*, 1997, pp. 72–83.

[11] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[12] P. Godefroid, “Using partial orders to improve automatic verification methods,” in *Computer-Aided Verification*, 1991, pp. 176–185.

[13] E. A. Emerson and A. P. Sistla, “Symmetry and model checking,” *Formal methods in system design*, vol. 9, no. 1, pp. 105–131, 1996.

[14] A. Miller, A. Donaldson, and M. Calder, “Symmetry in temporal logic model checking,” *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, pp. 8–es, 2006.

[15] M. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, and T. Wallentine, “Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 73–89, 2006.

[16] R. Bharadwaj and C. L. Heitmeyer, “Model checking complete requirements specifications using abstraction,” *Automated Software Engineering*, vol. 6, no. 1, pp. 37–68, 1999.

[17] C. Heitmeyer, J. K. Jr, B. Labaw, M. Archer, and R. Bharadwaj, “Using abstraction and model checking to detect safety violations in requirements specifications,” *Software Engineering, IEEE Transactions on*, vol. 24, no. 11, pp. 927–948, 2002.

[18] H. S. Hong, I. Lee, and O. Sokolsky, “Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking,” in *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, 2005, pp. 25–34.

[19] I. Brückner, K. Drger, B. Finkbeiner, and H. Wehrheim, “Slicing abstractions,” *Fundam. Inf.*, vol. 89, no. 4, pp. 369–392, 2009.

[20] G. J. Holzmann, *Personal communication*, Oct. 2005.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[22] B. Knaster, “Un thorme sur les fonctions d’ensembles,” *Pacific Journal of Mathematics*, no. 6, pp. 133–134, 1928.

[23] A. Tarski, “A lattice-theoretical fixed point theorem and its applications,” *Pacific J. Math*, no. 5, pp. 285–309, 1955.

[24] H. Wehrheim, “Incremental slicing,” *Formal Methods and Software Engineering*, pp. 514–528, 2006.