http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 8 (7): 990-997, 2009
ISSN 1812-5638

Conditioning for State Space Reduction in Program Model Checking

Long Yuejin and Xiao Jianyu
Department of Computer, Changsha University, Changsha Hunan, 410003, China

Abstract: This study aim to propose a scheme of applying program conditioning to reduce state space for program model checking, in which the antecedent of a implication form in LTL formula of program property is taken as the constrained condition of program conditioning and the statements irrelevant to satisfiability of the property are deleted. Analysis and experiment show that not only this scheme can effectively reduce a program's state, but also it can preserve the program's property.

Key words: Program model checking, state explosion problem, linear temporal logic, property preservation, program conditioning, symbolic execution

INTRODUCTION

Model checking (Clarke *et al*., 1999) is a kind of formal verification technique for finite state system. Model checking is based on exhaustively searching all states of the given system model for state instances which violate the system's property specifications which are expressed as temporal logic formula. The degree of reliability supported by model checking can match that of the traditional theorem proving. Model checking is adept at finding subtle fault which is difficult for normal test techniques. Compared to theorem proving, model checking has two special strong points: It is an algorithmic method and the checking process is completely automatic as soon as the system's model and properties have been constructed. The counter-example is showed when violation of property is found, which is extremely helpful to system debugging and maintenance. Model checking has now been taken as a routine verification tool in the industry of hardware design. Recently, software model checking, especially source code model checking (named program model checking) attracts more and more attentions (Visser *et al*., 2003). But until now, there is a long way to go for program model checking to be practical. There are two main curbing factors in front of the program model checking: Model construction problem-there is semantic gap between the input language of the mainstream model checker and programming languages. The former is static (adaptable for description of hardware design) and the latter has many dynamic constructs such as heap space allocation, recursive structure, dynamic creation of thread and polymorphism etc. State space explosion problem-the state space of a program is proportional to the number of variables and their domain. It can have huge state space (possibly infinite) even for small programs, which exceeds the ability of the currently available model checker. For program model checking to be accepted in software engineering, the first issue to handle is state space reduction. But it is worth emphasizing that the reduced model is required to safely abstract the relevant semantics of the given program. That is, on one side the reduced program must small enough to make the checking process tractable on the available model checker; on the other side it must be large enough to capture all information relevant to the property being checked.

Program conditioning (Danicic *et al*., 2005) (conditioning for short) is a kind of program simplification technique which identifies and deletes the unexecutable statements according to given constrained conditions. Conditioning process has two phases: symbolic execution phase (Coen-Porisini *et al*., 1991; Coward, 1988), in which each statement of the program is analyzed and annotated based on symbolic semantics and the execution paths leading to each statement is ascertained. Path analysis phase, in which the execution paths leading to each statement are reasoned with the help of automatic theorem prover to decide if there exists at least one path conform the given conditions. If the answer is no, the statement can be determined to be unreachable and can be deleted safely.

In this study, a scheme of applying the conditioning technique to reduce state space for program model checking is proposed, in which the antecedent of a implication form in LTL formula of program property is taken as the constrained condition of conditioning and the statements irrelevant to satisfiability of the property are deleted. Analysis and experiments shows that this scheme can effectively reduce a program's state space and at the same time preserve the program's property.

Corresponding Author: Long Yuejin, Department of Computer, Changsha University, Changsha Hunan, 410003, China

State space reduction (also named abstraction) is recently a great concern in the field of program model checking (Vasudevan and Abraham, 2004; Yorav and Grumberg, 2004; Bozga *et al*., 2003; Hatcliff *et al*., 2000; Sistla and Godefroid, 2004; Flanagan and Godefroid, 2005) and many strategies have been proposed which can be divided into two groups. Abstraction in group one is performed on the state-transition model of a program, such as symmetry reduction (Sistla and Godefroid, 2004) and partial order reduction (Flanagan and Godefroid, 2005) etc. Abstraction in group two is performed on source code, such as static program analysis (Yorav and Grumberg, 2004), static program slicing (Hatcliff *et al*., 2000) and predicate abstraction (Visser *et al*., 2000) etc. We are interested in the methods performed on source code. Abstraction by means of static analysis deletes redundant statements irrelevant to a given property through examining the control-flow graph of a program to extract information on its semantics without creating the semantic model. Abstraction through static slicing extracts slicing criterion from the primitive propositions in temporal logic formula of property and generates a smaller program that is functionally equivalent to the original program with regard to the criterion. Predicate abstraction characterizing a program in terms of how it transforms the truth value of a finite set of predicates which is constructed according to the verified properties. Conditioning belongs to group two which is performed on the level of program's syntax and semantics. It reduces a program by deleting unexecutable statements based on the analysis of program's symbolic execution semantics and reasoning of the reachability condition of each statement. Obviously, conditioning is different from the above mentioned methods.

The concept of conditioning was introduced by Canfora (1994). The earlier study (Coen-Porisini *et al*., 1991) used symbolic execution and theorem proving to specialize Ada programs which is very similar to conditioning. Recently conditioning attracts more and more attentions in the study of software's maintenace, test, reuse and reengineering (Hierons *et al*., 2002; Canfora *et al*., 1998). But until now, the technique of conditioning is not mature enough and there is few prototype systems for use. The technique of conditioning has not attracted attention of researchers in the field of software model checking.

STATE SPACE REDUCTION IN PROGRAM MODEL CHECKING

Program model checking: Model checking is a formal method which automatically searches a given kind of fault in all behavior of a system. According to experience of usage in hardware system, program model checking should include three basic steps: model construction, property specification and counter-example analysis.

Model construction means modeling program's semantics as a finite state-transition system which is expressed as a Kripke structure defined in Def. 1.

- **Def 1:** State-transition system (Kripke structure). A state-transition system is a tuple <S, S_0, Tr, L>, where, S is a finite set of states; $S_0 \subseteq S$ is a set of initial states; Tr is a finite set of transitions such that for each $t \in Tr$, $t \subseteq S \times S$ and L: $S \rightarrow 2^{AP}$ and is a labelling function which associates each state with the set of atomic propositions true in that state

The state in a program model is more complex than in a hardware model which involves two parameters (n, σ), where, n is a program point representing the position where the n-th statement is going to be executed and σ is a set of valuations of program variables. The transition in a program model represents the execution of one or more statements which change the state.

Property specification describes the constraints imposed on the legal sequence of states in a finite state model, which is always expressed as a temporal formula (Huth and Ryan, 2004) Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). We take the LTL as the language of property in this study. Linear temporal logic is based on states whose syntactic elements include atomic proposition P, connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) and temporal operator ($\Box$,$\Diamond$,u). That is, program property:

$$\psi \triangleq P \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \mathbf{u} \varphi_2$$

The basic expression of atomic proposition is $P \triangleq [m] \mid [x \; rop \; c]$, where [m] holds when execution reaches the statement with unique identifier (i.e., the statement at node will be executed next); [$\varkappa$ rop c] holds when the value of variable $\varkappa$ at the current node is related to constant c by the relational operator rop. The semantics of atomic proposition is defined with respect to states:

$$[\![[m]]\!](n,\sigma) \triangleq \begin{cases} \text{true if } m = n \\ \text{false else} \end{cases},$$

$$[\![[x \; rop \; c]]\!](n,\sigma) \triangleq \begin{cases} \text{true if } \sigma(x)[\![rop]\!][\![c]\!] \\ \text{false else} \end{cases}$$

The semantics of an LTL formula is defined with respect to an execution trace $\Pi = S_1,...,S_k$:

$$\begin{aligned}
\Pi \models [n] \quad &\text{iff} \quad [\![[n]]\!]s_1 = \text{true} \\
\Pi \models [x \text{ rop } c] \quad &\text{iff} \quad [\![[x \text{ rop } c]]\!]s_1 = \text{true} \\
\Pi \models \neg\varphi \quad &\text{iff} \quad \Pi \not\models \varphi \\
\Pi \models \varphi_1 \wedge \varphi_2 \quad &\text{iff} \quad \Pi \models \varphi_1 \wedge \Pi \models \varphi_2 \\
\Pi \models \varphi_1 \vee \varphi_2 \quad &\text{iff} \quad \Pi \models \varphi_1 \vee \Pi \models \varphi_2 \\
\Pi \models \varphi_1 \Rightarrow \varphi_2 \quad &\text{iff} \quad \Pi \models \varphi_1 \Rightarrow \Pi \models \varphi_2 \\
\Pi \models \Box\varphi \quad &\text{iff} \quad \forall i.\, \Pi^i \models \varphi (i \in \{1,\ldots,k\}) \\
\Pi \models \Diamond\varphi \quad &\text{iff} \quad \exists i.\Pi^i \models \varphi (i \in \{1,\ldots,k\}) \\
\Pi \models \varphi_1 \mathbf{u} \varphi_2 \quad &\text{iff} \quad (\exists i.\Pi^i \models \varphi_2 (i \in \{1,\ldots,k\})) \wedge (\forall j.\Pi^j \models \varphi_1 (j \in \{1,\ldots,i-1\}))
\end{aligned}$$

It should be noted that the logic operators listed here do not include next. Intuitively, the next operator allows one to count states and thus any attempt to reduce state space by compressing transitions in this setting is problematic (Huth and Ryan, 2001). The LTL can describe most of the important program properties such as safety and liveness. Dwyer *et al.* (1999) have developed a system of temporal logic specification pattern that provide templates for commonly used specification structures.

Counter-example analysis is to understand the output of model checker. If the model checker finds that the program's finite state-transition system violates the given property, it will show the path of counter-example which can be translated into statements sequence for understanding the program's wrong behavior.

Demand for state space reduction: In program's state-transition model, the number of states is exponential to the number of program's variables and components (Huth and Ryan, 2001). With the ever increasing complexity of software, the number of program's states could be very large even infinite. But model checker can only verify a limited finite states system. Thus program model checking can only aim at a reduced finite state model M'which is an abstraction of program's original model M with respect to the verified property. M' must be small enough to make the checking process tractable on the existed model checker and be large enough to capture all information relevant to the property being checked. State reduction can be performed on the Kripke structure (state-transition model) and can also on source code. The reduction on source code should always precedes on Kripke structure because the program's state space may be too huge to construct the corresponding Kripke structure.

Safety requirement: Def. 2: The safety of state space reduction: Given a program P and a specification φ, let P_r be the residual program resulted from state space reduction. Let Π be an execution trace of beginning at $(n_{init}, \sigma_{init})$, let Π_r be an execution trace of $P_r$beginning at $(n^r_{init}, \sigma_{init})$. We say P_r is a safe abstraction P of with respect to σ when $\Pi_r \models \varphi$ iff $\Pi_r \models \varphi$.

For a strategy of state space reduction to make sense in program model checking, it must be safe.

PROGRAM CONDITIONING

Program conditioning is a technique of program simplification. The principle of conditioning is that, in a program's execution trace, if a given constraint is imposed on the state of a control point, some statements will never be executed and can be deleted. The simplified program after conditioning has just the same behavior as the original program with respect to the given constraint. Conditioning consists of two phases: symbolic execution and path analysis.

The symbolic execution phase: In traditional analysis of program's semantics, each execution is interpreted as a sequence of states $<S_1, S_2,\ldots, S_n>$ where the value of each program variable in each state is constant. When a decision point (i.e., IF, WHILE statement) is confronted, the evaluation of conditional predicate unequivocally identifies the branch to follow. Symbolic execution is a natural extension of normal execution and normal computation can be seen as a special case of symbolic execution. In symbolic execution, a value is expressed as a symbol, a variable is bound with a symbol expression and computational definitions for the basic operators of the language are extended to accept symbolic inputs and produce symbolic formula as output. The only opportunity to introduce symbolic data objects is as inputs to the program. Each time a new input value of the program is required, it is supplied symbolically from the list of symbols $\{\alpha_1, \alpha_2,\ldots\}$. Program inputs are eventually assigned as values to program variables. The state of a program execution is normally expressed as a pair <variable, value>. By contrast, the state of program's symbolic execution is expressed as a compound tuple <<variable, symbol expression>, path condition> which is named as conditioned state or symbolic state. The Path Condition (PC) is a first-order predicate formula in the form of inequalities and expressions over the symbolic input $\{\alpha_1\}$ s which never contains program variables. The interpretation of conditioned state is that the variables will have the symbolic values if and only if the path condition equals to true.

The main function of symbolic execution is to propagate information of constraints, program's state and path forward to each control point. The symbolic execution of a program starts from the symbolic state $<state_{init}, PC_{init}>$. In $state_{init}$, program variables are bound to the special value under unless explicit initialization is

provided in the declarative part, whenever this is the case such variables are bound to their initial value. PC_{init} equals to true, i.e., no initial assumption is made on values of variables. In the process of symbolic execution, each statement of the program is annotated with symbolic state descriptions. In this study, we take a basic subset of C language as the object. Let <state, PC> be the conditioned state before the execution of statement, where, state = {<X_1, α_1(is a program variable and is an element in the set of symbols); Let be the new conditioned state resulting from the execution of starting from the conditioned state. The effects of symbolic execution of a generic program statement are described as follows:

- **Input/output statement:**

$$\sigma((\text{scanf}(\text{"\%d"},\&X_i),<\text{state},PC>)\triangleq<\text{state}',PC>$$

where, $\text{state}'=\{<X_1,\alpha_1>,...,<X_i,\beta>,...,<X_n,\alpha_n>\}$ and β is a new introduced symbolic value.

$$\sigma((\text{printf}(),<\text{state},PC>)\triangleq<\text{state},PC>$$

- **Assignment statement:**

$$\sigma((X_i:=\text{expr}(X_1,...,X_n)),<\text{state},PC>)\triangleq<\text{state}',PC>$$

where, $\text{state}'=\{<X_1,\alpha_1>,...,<X_i,\text{expr}(\alpha_1,...,\alpha_n)>,...,<X_n,\alpha_n>\}$.

- **Sequence of statements:**

$$\sigma((S_1;S_2;...;S_n),<\text{state},PC>)\triangleq\sigma((S_2;...;S_n),\sigma(S_1,<\text{state},PC>)$$

- **Conditional statement:**

$$\sigma((\text{if C then } S_1 \text{ else } S_2),<\text{state},PC>)\triangleq$$

if PC⇒C then $\sigma(S_1,<PC\wedge C>)$ else if PC⇒¬C, then $\sigma(S_2,<\text{state}, PC\wedge\neg C>)$ else $\sigma(S_1,<PC\wedge C>)\circ\sigma(S_2,<\text{state}, PC\wedge\neg C>)$ where ∘ operator represents composition of conditioned state which is defined as follows. Given two conditioned states $<\text{state}_1, PC_1>$ and $<\text{state}_2, PC_2>$, where, $<\text{state}_1 = \{<X_1,\alpha_1>,...,< X_n,\alpha_n>$, $<\text{state}_2 = \{<X,\beta_1,>,...,< X_n,\beta_n>$ the result of the application of the ∘ operator is: $<\text{state}_1, PC_1> \circ <\text{state}_2, PC_2> = <\text{state}_3, PC_3>$, where:

$$\text{state}_3=\{<X_1,\gamma_1>,...,<X_i,\gamma_i>,...,<X_n,\gamma_n>\}$$

where, $\forall i\in\{1,...,n\}\mid\alpha_i=\beta_i\Rightarrow\gamma_i=\alpha_i=\beta_i$ and

$$\forall i\in\{1,...,n\}\mid\alpha_i\neq\beta_i\Rightarrow\gamma_i$$

is a new symbol; $PC_3=(PC_1\wedge\Omega_1)\vee(PC_2\wedge\Omega_2)$ where,

$$\Omega_1=\bigwedge_{\forall i\in\{1,...,n\}\mid\alpha_i\neq\beta_i}(\gamma_i=\alpha_i),\Omega_2=\bigwedge_{\forall i\in\{1,...,n\}\mid\alpha_i\neq\beta_i}(\gamma_i=\beta_i)$$

- **Loop statement:**

$$\sigma((\text{while C S}),<\text{state},PC>)\triangleq$$

if PC⇒C then σ ((while C S),<state, PC ∧ C >) elseif PC⇒¬C then <state, PC> σ ((while C S), σ ((while C S), <state, PC >)). The conditioning of loop statements is somewhat difficult and the given solution is not very good. A better solution is first to transform loop statements into conditionals which is then conditioned (Hu *et al*., 2004).

- **Assert statement:**

$$\sigma((\text{assert}(C)),<\text{state},PC>)\triangleq$$

if C == ture then <state, PC> elseif C == false then <⊥,false> else <state, PC∧C>, where, ⊥ represents that no program variables have been defined.

There are many strategies proposed to symbolize statements of a program (Coen-Porisini *et al*., 1991; Danicic *et al*., 2005). When the symbolic execution is completed, each statement in the program is associated with the set of all conditional context. At the end of the symbolic execution, the value of the output variable is an expression of introduced symbols and constants and the last path condition is the accumulation of conditions which determines a unique control flow path through the program. As, in the relationship between arithmetic and algebra, the specific computations dictated by the program operator in symbolic execution are generalized and delayed.

The path analysis phase: Based on the conditioned state descriptions annotated in the symbolic execution phase, the function of the path analysis phase is to simplify the program by eliminating those inconsistent conditioned states by means of logic reasoning. The inconsistency means that the associated statement can never be executed and the program is then equivalent to one in which the statement is replaced by the empty statement. In this way, the conditioned program is constructed by deciding which statements have inconsistent path conditions. According to (Coward, 1988), in a sample of

programs, of the 1000 shortest paths only 18 were feasible. We can see that conditioning might be very effective in simplifying programs.

For any control point in a program, there may be many possible execution paths through it and the conditioned state can be expressed as:

$$\{<state_1\ PC_1>,...,<state_n,\ PC_n>\}$$

Path analysis only considers the truth value of the proposition:

$$\exists \vec{v}.(PC_1 \vee PC_2 ... \vee PC_n)$$

which is determined by a theorem prover. We may have three possible answers from the theorem prover: The proposition equals to true, then the associated statement is reachable and should be remained. The proposition equals to false, then the associated statement is unreachable and can be deleted; ? The truth value of proposition is not possible to tell. It could be the case that either path condition is not strong enough or that the theorem prover is not smart enough. A conservative strategy is taken and the associated statement should be remained. This conservatism is safe: if a statement is removed because of the outcome of the theorem proving, then that statement is guaranteed to be unnecessary in all states which satisfy the initial condition. Thus, the simplifying power of the conditioner depends on two factors: the precision of the symbolic executor and the precision of the theorem prover. By using an approximation to a program's semantic using a form of symbolic execution and by being willing to accept approximate results from the theorem prover, conditioning allows us to adopt reasoning that does not require the full force of inductive proofs.

As can be seen from the symbolic semantics, the size of the expressions produced increase for every statement. The number of paths is determined by the number of conditional statements and the number of while loops. For c conditional statements and l loops, the number of paths is $O(2^{c+l})$. The number of atomic propositions within each path is the sum of number of assignment statements on that path and the number of atomic propositions P within the boolean component of each conditional statements, assert statements (represented as s) and loop $O(\alpha+p(c+s+l))$. So the size of the symbolic expression at the final statement of a pragram is $O(n2^n)$ where, n = c+l+s, which is exponential to the number of conditional statements, asserts and loops (Coward, 1988). According to empirical evidence by Damcic *et al.* (2005), conditioning system has low degree polynomial behavior in many cases and get an average reduction in program size of approximately 35%.

CONDITIONING FOR STATE SPACE REDUCTION IN PROGRAM MODEL CHECKING

The scheme of state space reduction through conditioning: The proposed scheme is as follows. Each time a single program property is handled. Let an LTL formula φ be the property to be verified. First, the antecedent of an implication form which may exist in φ is extracted and taken as the conditional predicate of assert statement assert (), which is inserted in P an appropriate place in to form a new program P'. Then P' is processed by a conditioning tool and some statements irrelevant to the satisfaction of φ will be deleted and a simplified program P_r is constructed which is a reduced program model of P with respect to φ.

In the proposed scheme, expression of property is the major concern. In study (Manna and Pnueli, 1994), three classes of properties are considered to cover the majority of properties one would ever wish to verify:

- Invariance, which is expressed as □p in LTL
- Response, which is expressed as p⇒◊q in LTL
- Precedence, which is expressed as p⇒(qur) in LTL

In (Dwyer *et al.*, 1999), another classification is given which divides properties into eight types: Absence, Response, Bounded Existence, Universality, Precedence, Response, Chain Precedence and Chain Response. Also, in (Dwyer *et al.*, 1999), an empirical investigation shows that the response type occupies 50% of the 500 property specifications. There are still other kinds of classification and they can approximately include each other (Dwyer *et al.*, 1999).

We can see from (Manna and Pnueli, 1994; Dwyer *et al.*, 1999) that the most commonly used LTL formula of property is implication form: antededent⇒consequent. In our scheme, the antecedent of the implication form is taken as the conditional constraint of conditioning. As mentioned earlier LTL formula of property is based on states and each state of a program involves two parameters: control point (line number of statement) and variables. Thus there are three possible occasions to handle in the scheme.

- When the constraint only relates to variables [x rop c], we impose the constraint on the initial state of the program, i.e., [x rop c] is taken as the conditional predicate of assert () which is inserted in the program's start point

- when the constraint only relates to control point [n], we first symbolically execute the program and extract the path condition of the n-th statement which is taken as conditional predicate of assert () and insert it in the program's start point
- When the constraint relates to both control point [n] and variables [x rop c], we take [x rop c] as the conditioned predicate of assert () and insert it before the n-th statement in the program

Even if there is no constraint which can be extracted from the property formula, conditioning can eliminate a lot of redundant statements (Danicic *et al*., 2005).

Safety analysis: State space reduction for program model checking is required to fulfill the safety requirement given in earlier. That is, given a property φ and an execution path Π in the original program P and an execution Π_r path in the simplified program P_r, if Π and Π_r have a same initial state, it must be guaranteed that $\Pi \mid = \varphi$ iff $\Pi_r \mid = \varphi$.

We first examine the occasion in which no constraint is extracted from the property formula. According to section 4, Conditioning has considered all possibilities of input. The deleted states by conditioning can never exist in any actual execution paths and the corresponding deleted statements can never contribute to any transitions in execution paths. So, given an initial state, for any execution path Π in the original program P, there is always a corresponding execution path Π_r in the simplified program P_r and Π_r completely equals Π to and of course satisfies $\Pi \mid = \varphi$ iff $\Pi_r \mid = \varphi$.

We now examine the occasion where the property formula is $p \Rightarrow q$ and p is taken as the constraint of conditioning. According to the rule of state deletion in conditioning, (ideally) all the conditioned states whose path condition satisfies p = false and the associated statements will be deleted. Due to the fact that the path condition of the deleted states satisfies p = false, the formula $p \Rightarrow q$ in the labels of these states in the state-transition system of the program must be true. It is not necessary to further check these states and deletion of these states will not influence the satisfaction of property $p \Rightarrow q$.

RESULTS

The aim of the study is to verify that state space reduction through conditioning fulfill the safety requirement of program model checking, i.e., the simplified program preserves the property to be verified of the original program to observe the effectiveness of conditioning in state space reduction.

The experiment consists of four steps:

- Given a program in C language and a property formula in LTL, a constraint is extracted from the property formula according to the scheme described which is taken as the conditional predicate of assert (). The assert () will be inserted in the appropriate place in the program
- The resulted program from step 1 is processed by the gcc compiler, whose output is taken as input of the tool of conditioning CF3 (http://www.fluppet.demon.co.uk/Software/cf3_svc_public/). Run CF3
- Take the output of CF3 and the property formula as input of program model checker blast1.0 (http://www-cad.eecs.berkeley.edu/~tah/blast/). Run blast1.0 and observe the output
- Take the original program and the property formula as input of blast1.0. Run blast1.0 and compare the output to that of step 3.

Samples of program in this experiment can not be complex because the conditioner CF3 is now unable to support construct of pointer and loop. We've collected three samples. The first is from the manual of blast1.0 with the file name of tut2.c and the property is that a public variable must be used in the given order. The associated assert () statements have been inserted in the program by the author of the manual. We have rewritten the while loop in the program in conditional statements. The second sample is by (Danicic *et al*., 2005) which is a program for computing UK's taxation and the property to be verified is $((age = 70) \wedge (income = 6000) \wedge (blind = 0) \wedge (married = 1)) \Rightarrow (tax = 0)$ which is satisfiable in the original program. The third sample is the same program of the second sample but the property to be verified is $((age = 40) \wedge (income = 18000) \wedge (blind = 0) \wedge (married = 1)) \Rightarrow (tax = 0)$ which is not satisfiable in the original program.

The experiment is performed using a 1.53 GHz Pentium processor with 256 M RAM. The OS is RedHat Linux 9.0. The result is shown in Table 1.

The experiment result shows that conditioning does not influence the output of model checking a program

Table 1: Experiment result of conditioning for state space reduction in program model checking

	The original program			The conditioned program		
No. of samples	Length (lines)	Time for checking (sec)	Property satisfied	Lenght (lines)	Time for checking (sec)	Checking result
1	38	0.6	yes	38	0.6	yes
2	96	1.6	yes	9	0.5	yes
3	96	2.5	no	11	0.5	no

property. It is worth noting that in the experiment of the third sample, the counter-example given by model checking of the simplified program is just the same as that of the original program. The effectiveness of conditioning to state space reduction varied from program to program and from property to property. Being limited by the available conditioner, further empirical conclusions can not be drawn from the experiment due to the simplicicity of the samples of programs.

CONCLUSION AND FUTURE WORK

Program model checking is a formal method to automatically find a given kind of fault by means of exhaustively searching all possible behavior of a program. Program model checking is now hindered by the state space explosion problem. Conditioning is a technique of program simplification which is based on deleting the unexecutable statements when a condition of interest holds at some point in a program. In this study, we propose a scheme to apply conditioning to reduce state space for program model checking. Analysis and experiment show that this scheme can effectively reduce a program's state space and at the same time preserve the program's property. Due to limitation of the currently avaible tool of conditioning, programs of industry scale have not been evaluated in our experiment. The conditioning technique used in this study is forward conditioning which considers the effect of propagating state information forward from a condition. The recent study (Fox *et al*., 2001) introduces the concept of backward conditioning which propagates state information backward from the condition point to delete statements which cannot cause execution to satisfy the condition. By (Harman *et al*., 2001) discusses a unified framework of forward and backward conditioning. The future work of ours is to study the combination of forward and backward conditioning to reduce state space for program model checking.

ACKNOWLEDGMENT

A project Supported by Scientific Research Fund of Hunan Provincial Education Department (06B011).

REFERENCES

Bozga, M., J.C. Fernandez and L. Ghirvu, 2003. State space reduction based on live variables analysis. Sci. Comput. Program., 47: 203-220.

Canfora, G., A. Cimitile, A. de Lucia and G.A. Di Lucca, 1994. Software salvaging based on conditions. Proceedings of the International Conference on Software Maintenance, Victoria, BC, Canada, Sept. 20-23, IEEE Computer Society, pp: 424-433.

Canfora, G., A. Cimitile and A. de Lucia, 1998. Conditioned program slicing. Inform. Software Technol., 40: 595-607.

Clarke, E., O. Grumberg and D. Peled, 1999. Model Checking. 1st Edn., Springer Berlin Heidelberg, MIT Press, New York, pp: 54-56.

Coen-Porisini, A., F. de Paoli, C. Ghezzi and D. Mandrioli, 1991. Software specialization via symbolic execution. IEEE Trans. Software Eng., 17: 884-899.

Coward, D., 1988. Symbolic execution systems: A review. Software Eng. J., 3: 229-239.

Danicic, S., M. Daoudi, C. Fox, M. Harman and R.M. Hierons *et al*., 2005. ConSUS: A light-weight program conditioner. J. Syst. Software, 77: 241-262.

Dwyer, M., G. Avrunin and J. Corbett, 1999. Patterns in property specifications for finite-state verification. Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA. USA., May 16-22, ACM Press, pp: 411-420.

Flanagan, C. and P. Godefroid, 2005. Dynamic partial-order reduction for model checking software. Proceedings of the ACM Symposium on Principles of Programming Languages, Long Beach, California, USA., Jan. 12-14, ACM Press, USA., pp: 110-121.

Fox, C., M. Harman, R. Hierons and S. Danicic, 2001. Backward conditioning: A new program specialisation technique and its application to program comprehension. Proceedings of the 9th International Workshop on Program Comprehension, Toronto, Canada, May 12-13, IEEE Computer Society, pp: 89-97.

Harman, M., R.M. Hierons, C. Fox, S. Danicic and J. Howroyd, 2001. Pre/Post conditioned slicing. Proceedings of the IEEE International Conference on Software Maintenance, Florence, Italy, Nov. 6-10, IEEE Computer Society, pp: 138-147.

Hatcliff, J., M.B. Dwyer and H. Zheng, 2000. Slicing software for model construction. J. Higher-Order Symbolic Comput., 13: 315-353.

Hierons, R., M. Harman, C. Fox, L. Ouarbya, M. Daoudi, 2002. Conditioned slicing supports partition testing. Software Test., Verificat. Reliabil., 12: 23-28.

Hu, L., M. Harman, R.M. Hierons and D. Binkley, 2004. Loop squashing transformations for amorphous slicing. Proceedings of the IEEE 11th Working Conference on Reverse Engineering, Delft University, the Netherlands, Nov. 9-12, IEEE Computer Society, pp: 152-160.

Huth, M. and M. Ryan, 2001. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, Cambridge.

Manna, Z. and A. Pnueli, 1994. Temporal verification diagrams. Proceedings of the International Symposium on Theoretical Aspects of Computer Software, Tohoku University, Sendai, Japan, Apr. 19-21, Springer-Verlag, pp: 726-765.

Sistla, A.P. and P. Godefroid, 2004. Symmetry and reduced symmetry in model checking. ACM Trans. Program. Languages, 26: 702-734.

Vasudevan, S. and J.A. Abraham, 2004. Static program transformations for efficient software model checking. Proceedings of the IFIP Congress Topical Sessions, Toulouse, France, Aug. 22-27, Springer, Boston, pp: 257-281.

Visser, W., K. Havelund, G. Brat and S. Park, 2000. Model checking programs. Proceedings of the 15th International Conference on Automated Software Engineering (ASE). Grenoble, France, Sept. 11-15, IEEE Computer Society, pp: 3-11.

Visser, W., K. Havelund, G. Brat and S. Park, 2003. Model checking programs. Automated Software Eng., 10: 203-232.

Yorav, K. and O. Grumberg, 2004. Static analysis for state-space reductions preserving temporal logics. Formal Methods Syst. Des., 25: 67-96.