Formal Verification for C Program

Article *in* Informatica · January 2007

Source: DBLP

CITATIONS
5

READS
269

2 authors:

Junyan Qian
Guilin University of Electronic Technology
79 PUBLICATIONS **146** CITATIONS

SEE PROFILE

Baowen Xu
Nanjing University
466 PUBLICATIONS **5,187** CITATIONS

SEE PROFILE

INFORMATICA, 2007, Vol. 18, No. 2, 289–304

Formal Verification for C Program *

Junyan QIAN[1,2], Baowen XU[1]
[1]*Department of Computer Science and Engineering, Southeast University*
Nanjing 210096, China
[2]*Department of Computer Science, Guilin University of Electronic Technology*
Guilin 541004, China
e-mail: qjy2000@guet.edu.cn, bwxu@seu.edu.cn

Received: March 2006

Abstract. Iterative abstraction refinement has emerged in the last few years as the leading approach to software model checking. We present an approach for automatically verifying C programs against safety specifications based on finite state machine. The approach eliminates unneeded variables using program slicing technique, and then automatically extracts an initial abstract model from C source code using predicate abstraction and theorem proving. In order to reduce time complexities, we partition the set of candidate predicates into subsets, and construct abstract model independently. On the basis of a counterexample-guided abstraction refinement scheme, the abstraction refines incrementally until the specification is either satisfied or refuted. Our methods can be extended to verifying concurrency C programs by parallel composition.

Key words: program verification, predicate abstraction, model checking.

1. Introduction

Currently code validation falls into two categories: testing and formal verification. Formal verification mainly includes two methods: theorem proving and model checking. *Theorem Proving* requires considerable expertise to guide and assist the verification process, and can not generate counter-examples that are useful for debugging when the verification fails. *Model Checking* (Clarke *et al.*, 1999) is an automatic formal verification technique for a finite state system, where all the states of the system are exhaustively enumerated and the correctness condition checked at each state. Moreover, model checking yields extremely useful counter examples if it fails. It has proven effective in detecting errors in hardware designs. Software model checking could produce major enhancements in software reliability and robustness. However, the state space of software programs is typically so huge that they cannot be directly model checked with conventional model checking methods. Fortunately, applying mathematically abstraction methods might extract a reduced model from a program which makes model checking feasible.

*This work is supported by National Outstanding Young Scientists Foundation of China under Grant No. 60425206, Natural Science Foundation of China under Grant No. 60663005, No. 60633010, The High Technology Research Project of Jiangsu Province of China under Grant No. BG2005032, Guangxi Natural Science Foundation of China under Grant No. 0542036.

Abstraction (Clarke *et al.*, 1994) has been widely used to make model checking more efficient for large systems. Our method is based on *Predicate Abstraction* presented firstly by Graf and Saidi (Graf and Saidi, 1997), which is a special kind of *Abstract Interpretation* (Cousot and Cousot, 1977), where the abstract domain is constructed using a given set of predicates, i.e., a potentially unbound data type is abstracted to a finite set of points.

Predicate abstraction has been popularly and widely applied to systematic abstraction of programs in recent years. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. When model checking of the abstract program fails it may produce a counterexample that does not correspond to a concrete counterexample which is called a spurious counterexample. Consequently, the set of predicates is refined heuristically, and a new abstraction is computed.

Iterative abstraction refinement has emerged in the last few years as the leading approach to software model checking. A framework for software model checking works as follows (Corbett *et al.*, 2000; Saidi, 2000; Ball and Rajamani, 2001).

Step 1 (Abstraction). Create an abstraction $\mathcal{A}$ of the program $\mathcal{C}$ such that $\mathcal{C}$ conforms to $\mathcal{A}$ by construction.

Step 2 (Verification). The abstraction model $\mathcal{A}$ is checked automatically against the desired property ϕ. If $\mathcal{A} \models \phi$, i.e., ϕ is satisfiable in $\mathcal{A}$, then the verification is successful; otherwise, an abstract counterexample is automatically produced, whose spuriousness must be checked. If the counterexample is not spurious, a concrete counterexample is reported and the verification process stop; otherwise go to the next step.

Step 3 (Refinement). Because the chosen set of predicates is not enough to prove program correctness, and results in the failure to concretize the abstract counterexample, new predicates are discovered and added to refine abstract model using the spurious counterexample. Go to Step 1.

Model checking program is the automatic process of deciding whether a program satisfies a given specification or property, and should yield a "yes" or "no" answer. For the sake of simplicity, we focus on the automatic abstraction method for verifying sequential C programs. The method eliminates unneeded variables using program slicing technique, and then automatically extracts an initial abstract model from C source code using predicate abstraction and theorem proving. In order to reduce time complexities, we partition the set of candidate predicates into subsets, and construct abstract model independently. Moreover our method can be extended to verifying concurrency C programs.

Relate work. Predicate abstraction was first introduced by Graf and Saidi in (Graf and Saidi, 1997). During the last years, predicate abstraction (Clarke *et al.*, 2003; Bensalem *et al.*, 1998; Das *et al.*, 1999; Henzinger *et al.*, 2002; Ball and Rajamani, 2001) is being used in software model checking such as Bandera (Corbett *et al.*, 2000), Java PathFinder (Havelund and Pressburger, 2000), SLAM (Ball *et al.*, 2001; Ball and Rajamani, 2001), MAGIC (Chaki *et al.*, 2003a; Chaki *et al.*, 2004) and BLAST (Henzinger *et al.*, 2002), where the first two focus on Java while the last three all deal with C program. The systems are increasingly able to handle industrial software. In SLAM project, an abstract Boolean program is constructed based on the abstraction predicates, then the Boolean program is

model checked to see if error states are ever reachable. In contrast to SLAM which uses symbolic algorithms, BLAST is an on-the-fly reachability analysis tool, and MAGIC uses LTS as specification formalism.

The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm (Clarke *et al.*, 2000), or CEGAR for short. The notion of CEGAR was originally described by Kurshan (Alur *et al.*, 1995) for model checking finite state models. Counterexample guided refinement has even been used with predicate abstraction by Lakhnech et al. (Lakhnech *et al.*, 2001). One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine automatically the abstract program, and the process proceeds until no spurious error traces can be found.

The remainder of this paper is organized as follows. The next section introduces briefly some preliminary notations, while Section 3 gives the pretreatment of C programs before constructing abstract model. Section 4 constructs an initial abstract model from C program. Section 5 gives a method for partitioning the set of candidate predicates into subsets. Section 6 discusses refining abstract model by CEGAR. Finally, Section 7 provides our conclusions.

2. Preliminaries

2.1. *Abstraction*

Abstraction is a general proof technique where a system is first simplified, then the simplified system is analyzed, and the results are transferred back to the original system. Since a simplified system is analyzed, the proof is easier to do. Abstraction has been widely applied in program analysis, compilation and verification. For model checking, a general application of program abstractions is to reduce the complexity of a program model in order to overcome the state-space explosion problem. Abstraction techniques reduce program state space by mapping the set of states of the actual system to a set of abstract states.

When model checking programs using abstraction, the main concern is that the abstractions must be property-preserving. There are two forms of property preservation: *Weak Preservation* and *Strong Preservation*. An abstraction is weak property preserving if a set of properties true in the abstract system has corresponding properties in the concrete system that are also true, while an abstraction is strong preserving if a set of properties with truth values either true or false in the abstract system has corresponding properties in the concrete system with the same truth values.

It is usually difficult and expensive to compute a precise abstraction directly. In order to reduce the complexity, approximation is often used. There are mainly two forms of approximation abstraction: over-approximation and under-approximation. In over-approximation, more behaviors are added in the abstract system than are present in the concrete system. This approach provides a very popular class of weakly preserving abstractions for universally quantified path properties. However, over-approximation often only works well for safety (or invariant) properties. In under-approximation, some

behaviors are removed when going from the concrete to the abstract system. Under-approximation is also often found in the construction of an environment for a system to be checked. In this article, we focus on weak preservation, over-approximation method of abstraction.

2.2. *Notation*

Let S_1 and S_2 be sets of states, and let f be a function mapping the powerset of S_1 to the powerset of S_2, i.e., f: $2^{S_1} \rightarrow 2^{S_2}$. The dual of the function f is defined to be $\widetilde{f}(X) = \overline{f(\overline{X})}$, where the overbar indicates complementation in the appropriate set of states.

DEFINITION 1. Let ρ be a relation over $S_1 \times S_2$ defined in the usual way as a set of pairs, the pre-image function $pre[\rho]$: $2^{S_1} \rightarrow 2^{S_2}$ and the post-image function $post[\rho]$: $2^{S_2} \rightarrow 2^{S_1}$ under the relation ρ are defined as follows:

$$pre[\rho](A) = \big\{s_1 \in S_1 \mid s_2 \in A, \rho(s_1, s_2)\big\},$$
$$post[\rho](B) = \big\{s_2 \in S_2 \mid s_1 \in B, \rho(s_1, s_2)\big\}.$$

The dual function $\widetilde{pre}[\rho](A)$ is defined by $\widetilde{pre}[\rho](A) = S_1 \backslash \widetilde{pre}[\rho](S_2 \backslash A)$.

To reason about a concrete state machine and an abstraction of that machine, we will use the concept of a Galois connection to establish a relationship between the set of concrete states S_1 and the set of abstract states S_2.

DEFINITION 2. Let Id_S denote the identity function on the powerset of S. A Galois connection from 2^{S_1} to 2^{S_2} is a pair of monotonic functions (α, γ), where α: $2^{S_1} \rightarrow 2^{S_2}$ and γ: $2^{S_2} \rightarrow 2^{S_1}$, such that $Id_{S_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq Id_{S_2}$.

For any Galois connection (α, γ) from 2^{S_1} to 2^{S_2}, we have,$\gamma(Y) = \bigcup\{X \in 2^{S_1} \mid \alpha(X) \subseteq Y\}$, $\alpha(X) = \bigcap\{Y \in 2^{S_2} \mid X \subseteq \gamma(Y)\}$. The functions α and γ are often called respectively the abstraction function and the concretization function. The Galois connection that we will be using in this paper is described in the following proposition.

Theorem 1 (Loiseaux *et al.*, 1995). *Given a relation $\rho \subseteq S_1 \times S_2$, the pair $(post[\rho], \widetilde{pre}[\rho])$ is a Galois connection between 2^{S_1} and 2^{S_2}, and denoted by $(\alpha_\rho, \gamma_\rho)$.*

Sometimes, we abbreviate (α, γ) when the relation ρ is clear from the context.

2.3. *Weakest Preconditions*

For a statement *s* and a predicate *p*, let $\mathcal{WP}(s, p)$ denote the weakest precondition (Ball and Rajamani, 2001) of *p* with respect to a given statement *s*. $\mathcal{WP}(s, p)$ is defined as the weakest predicate whose truth before *s* entails the truth of *p* afterwards. Consider an assignment *s* of the form $x = e$, where *x* is a variable and *e* is an expression. Then the

weakest precondition rule says that $\mathcal{WP}(x = e, p)$ is obtained from *p* by replacing all occurrences of *x* in *p* with *e*, denoted $p[e/x]$. For example, $\mathcal{WP}(x = x + 1, x < 5) = (x + 1) < 5 = x < 4$. Therefore, $(x < 4)$ is true before $x = x + 1$ executes if and only if $(x < 5)$ is true afterwards. $\mathcal{WP}$ for assignments is defined as follows:

$$\mathcal{WP}(x = e, p) = p[e/x].$$

Give a statement *s*, a set of predicates $\mathcal{P}$, and predicate $p \in \mathcal{P}$, it may be the case that $\mathcal{WP}(s, p)$ is not in $\mathcal{P}$. For example, suppose $\mathcal{P} = (x < 5), (x = 2)$. We have seen that $\mathcal{WP}(x = x + 1, x < 5) = x < 4$, but the predicate $(x < 4)$ is not in $\mathcal{P}$. Therefore, we need to use theorem prover to limit the weakest precondition to level of an expression over the predicates in $\mathcal{P}$. In the example, we can show that $x = 2 \Rightarrow x < 4$. Therefore if $(x = 2)$ is true before $x = x + 1$, then $(x < 5)$ is true afterwards.

Consider the case of pointers, $\mathcal{WP}(x = e, p)$ is not necessarily $p[e/x]$. For example, $\mathcal{WP}(x = 3, y > 5)$ is not $(*y > 5)$ because if *x* and $*y$ are aliases, then $(*y > 5)$ cannot be true after the assignment to *x*. A similar problem occurs when a pointer dereference is on the left-hand side of the assignment. In order to handle the problems, we consider two cases: either *x* and *y* are aliases, or they are not. If *x* and *y* are aliases, then we replace every occurrence of *y* in *p* with *e*; otherwise, we will leave predicate *p* unchanged. In the example above, we have $\mathcal{WP}(x = 3, *y > 5) = (\&x = y \wedge 3 > 5)(\&x \neq y \wedge *y > 5)$.

3. Pretreatment of C Programs

Before constructing an abstract model of program, we first eliminate unneeded variables using program slicing technique, and then restrict C programs to a simple intermediate form.

3.1. *Program Slicing*

Analogously to the method of (Holzmann, 2000), we remove irrelevant variable for checking a given property using program slicing (Tip, 1995) which is used to reduce a source program to a smaller fragment. Statements of interest, called the slicing criterion, can be obtained by the set of variable names which occur in the temporal logic formula being verified. Slicing algorithms are based on data and control dependency analysis of the program text.

The intention of the program slicing is to identify those parts of the program that are irrelevant with respect to the properties to be proven. While checking a specification ϕ on a program $\mathcal{C}$, we first use slicing to remove the statements of $\mathcal{C}$ that do not affect the satisfaction of ϕ. We require that the specification ϕ hold for program $\mathcal{C}$ if and only if ϕ hold for the slice of program $\mathcal{C}$, i.e., the slice of program $\mathcal{C}$ is both sound and complete with respect to ϕ.

We always apply the program slicing automatically before constructing abstract model using predicate abstraction.

3.2. *Restrictions on C Programs*

Before applying predicate abstraction, we assume that the C program has been translated into a simple intermediate form in which: (1) all intraprocedural control-flow is accomplished with if-then-else statements and goto; (2) all expressions are free of side-effects and short-circuit evaluation and do not contain multiple dereferences of a pointer; (3) a function call only occurs at the top-most level of an expression (for example, $z = x+f(y)$ is replaced by $t = f(y), z = x + t$). We use the CIL compiler infrastructure (Necula *et al.*, 2002) as a front end to parse C programs.

Our method can analyze all C syntactic structure, including pointers, structures and procedures. Without loss of generality, we can assume that there are only six kinds of statements in C: assignments, if-then-else branches, goto, call, return and exit-procedure.

4. Constructing the Abstract Model

In this section we describe the process for creating an abstract framework that can be applied in model checking given a C program and an initial set of predicates $\mathcal{P}$.

4.1. *Control Flow Graph*

Given a sequential program $\mathcal{C}$ consisting of *n* statements and *p* procedures, we assign to each statement a unique index from l_0 to l_{n-1}, and to each procedure a unique index from l_n to l_{n+p-1}. For the sake of simplicity, we assume that both variable's and label's names are globally unique in $\mathcal{C}$. We also assume the existence of a procedure called *main*: it is the first procedure to be executed. As a running example, we use the C program shown in Fig. 1.

The control flow graph CFG is a finite directed graph describing the flow of control in program. The nodes of the CFG are called control locations and correspond to the values of the program counter. The edges denote transfer of control locations. Intuitively, the control flow automaton CFA can be obtained by viewing the CFG of a program $\mathcal{C}$ as an automaton where the states of a CFA correspond to control locations and the transitions between states in the CFA correspond to the control flow between their associated control

```
      int x, even;
l11:  main(){                    l12:  parity(n){
l0:       x=10;                           int i;
l1:       even=1;                l6:       if (n<>0){
l2:       parity(x);             l7:            i=n-1;
l3:       if(even<>1){           l8:            even=-1*even;
l4:            ERROR;}           l9:            parity(i); }
l5:       return; }              l10:      return;}
```

Fig. 1. A simple C program.

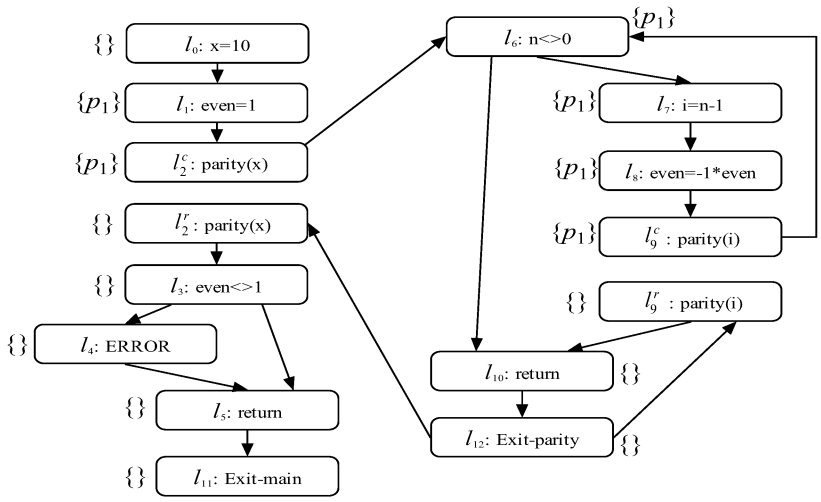

Fig. 2. The CFA of the program shown in Fig. 1. Each location is labelled by the corresponding statement label. The location s are also labelled with inferred predicates when $\mathcal{P} = p_1$ where $p_1 = (x == 0)$.

locations in the program. The CFA can be seen as a conservative abstraction of $\mathcal{C}$'s control flow, i.e., it allows a superset of the possible traces of $\mathcal{C}$. Formally, let *Stmt* be the set of statements of $\mathcal{C}$, a CFA of $\mathcal{C}$ is a 4-tuple (C, I, T, L), where C is a set of control locations, $I \in S$ is an initial location, $T \subseteq S \times S$ is a set of transitions, L: $C \rightarrow Stmt$ is a labelling function. The transitions between control locations reflect the flow of control between their labelling statements.

For describing conveniently, each statement of a procedure call will be splitted into two sub-statements labelled l^c and l^r, where l^c denotes pre-procedure call and l^r denotes after-procedure return. Each procedure has an exit procedure location labelled the exit-<procedure-name>. The CFA of the program in Fig. 1 is shown in Fig. 2. Each location is labelled by the corresponding statement label. Therefore the control locations of the CFA are $l_0, ..., l_{12}$ with l_0 being the initial location, l_{11} and l_{12} being the exit procedure location.

4.2. *Predicate Abstraction*

The basic idea of predicate abstraction is to represent a concrete variable using a Boolean variable whose value depends on an expression over the concrete variable itself. In predicate abstraction, the main challenge is to identify the predicates that are necessary for proving the given property. A set $\mathcal{P} = \{p_1, ..., p_k\}$ of the pure Boolean C expressions called predicates, includes those in the property to be verified, are identified from the concrete program. They also serve as the atomic propositions that label the states in the concrete and abstract transition systems. That is, the set of atomic propositions is $AP = \{p_1, ..., p_k\}$. A state in the concrete system will be labelled with all the predicates it satisfies. Each predicate p_i is associated with a Boolean variable b_i that represents its

truth value. So each abstract state is a valuation of these *k* Boolean variables. An abstract state will be labelled with predicate p_i if the corresponding bit b_i is 1 in that state.

The predicates are also used to define a total function ρ between the concrete and abstract state spaces. A concrete state *s* will be related to an abstract state $s_{\mathcal{A}}$ through ρ if and only if the truth value of each predicate on *s* equals the value of the corresponding Boolean variable in the abstract state $s_{\mathcal{A}}$. Formally,

$$\rho(s, s_{\mathcal{A}}) = \bigwedge_{i\in[1,k]} p_i(s) \Leftrightarrow b_i(s_{\mathcal{A}}).$$

Let ρ be an abstraction function. Based on Section 2.2, the pair of functions $post[\rho]$ and $\widetilde{pre}[\rho]$ generated from relation ρ forms a Galois connection. We will denote this Galois connection by (α, γ).

DEFINITION 3. Assume the set of concrete states $S_{\mathcal{C}}$ and the set of abstract states $S_{\mathcal{A}}$, $s \in S_{\mathcal{C}}, s_{\mathcal{A}} \in S_{\mathcal{A}}$, The abstraction and concretization functions, α: $S_{\mathcal{C}} \rightarrow S_{\mathcal{A}}$ and γ: $S_{\mathcal{A}} \rightarrow S_{\mathcal{C}}$ are defined as, $\alpha(s)(b_i(s_{\mathcal{A}})) = p_i(s), \gamma(s_{\mathcal{A}})(s) = \bigwedge_{i\in[1,k]} p_i(s) \Leftrightarrow b_i(s_{\mathcal{A}})$.

Using this ρ, we can build an abstract model which simulates the concrete model. A valuation of $\mathcal{P}$ is a vector $\vec{v} = v_1...v_k$ of Boolean values, such that v_i expresses the Boolean value of p_i. $\mathcal{V}$ denotes the set of all valuations, i.e., the set of abstract memory states. Intuitively, a valuation typically models many concrete memory states. Given a valuation $\vec{v} = v_1...v_k$, the concretization $\gamma(\vec{v})(s)$ is defined as $\bigwedge_{i\in[1,k]} p_i(s) \equiv v_i$, where $p_i(s) \equiv v_i$ is equal to p_i if v_i is true, and equal to $\neg p_i$ if v_i is false. For example, $\mathcal{P}$ contains a single predicate $(x == 0)$ and there has two valuations 0 and 1, so $\gamma(0) = \neg (x == 0)$ and $\gamma(1) = (x == 0)$. That is, Boolean valuation 0 models all concrete states where the variable *x* is not equal to 0 while Boolean valuation 1 models all concrete states where the variable *x* is equal to 0.

For the construction of abstract model $\mathcal{A}$, we combine the control flow graph and the predicate abstraction to obtain the state space $C \times \mathcal{V}$. A state of $\mathcal{A}$ is a pair $\langle c, \vec{v}\rangle$, where $c \in C$ and $\vec{v} \in \mathcal{V}$. By computing the weakest precondition $\mathcal{WP}$ of a predicate *p* relative to a given statement *s*, we construct the $\mathcal{P}_c$ associated with each control location *c* of the CFA a finite subset of $\mathcal{P}$.

For describing predicate abstraction of procedure call, we define Globals($\mathcal{C}$) as the set of global variables of $\mathcal{C}$. Let $\mathcal{P}^G$ denote the global predicates of $\mathcal{P}$. For a procedure *R*, Let $\mathcal{P}^R$ denote the subset predicates in $\mathcal{P}$ that are local to *R*; F_R is the set of formal parameters of the procedure *R*; L_R is the set of local variables and formal parameters of the procedure *R*, while sL_R is the set of strictly local variables (i.e., without the formal parameters) of the procedure *R*. Finally, $First_R$ is the index of the first statement of procedure *R*, and $Proc(l_i)$ is the procedure belonging to the statement $L(l_i)$.

Firstly, we determine the signature of each procedure. Let $vars(e)$ be the set of variables referenced in expression, and $drfs(e)$ be the set of variables dereferenced in expression *e*. The signature of procedure *R* is a four-tuple $\langle F_R, r_R, \mathcal{P}^R_f, \mathcal{P}^R_r\rangle$, where

input: the set of candidate predicate $\mathcal{P}$
output: $\mathcal{P}_c$ for each CFA location
do
  for each location $c \in C$ **do**
    case c:
      case 1: $L(c)$ is an assignment statement and $L(c')$ is its successor
        for each $p' \in \mathcal{P}_{c'}$ **do** add $\mathcal{WP}(L(c), p')$ to $\mathcal{P}_c$
      case 2: $L(c)$ is a condition statement and $L(c')$ is its successor
        if $L(c) \in \mathcal{P}$ or $\neg L(c) \in \mathcal{P}$, **then** add $L(c)$ to $\mathcal{P}_c$
        $\mathcal{P}_c := \mathcal{P}_c \cup \mathcal{P}_{c'}$
      case 3: $L(c)$ is a goto or after-procedure return statement
        and $L(c')$ is its successor
        $\mathcal{P}_c := \mathcal{P}_c \cup \mathcal{P}_{c'}$
      case 4: $L(c)$ is a pre-procedure call statement and $L(c')$ is its successor
        $\mathcal{P}_c := \mathcal{P}_{c'} \backslash \mathcal{P}^{\text{Proc(c)}}$
      case 5: $L(c)$ is a return statement and $L(c')$ is its successor
        $\mathcal{P}_c := \mathcal{P}_{c'} \cup \mathcal{P}_f^R \cup \mathcal{P}_r^R$
      case 6: $L(c)$ is an exit procedure statement and $L(c')$ is its successor
        $\mathcal{P}_c := \mathcal{P}_c \cup \mathcal{P}_{c'}$
until no $\mathcal{P}_c$ was changed in the for loop

Fig. 3. The algorithm of predicate inference.

F_R is the set of formal parameters of the procedure R; $r_R \in L_R$ is the return variable of the procedure R; $\mathcal{P}_f^R$ is the set of formal parameter predicates, defined as $\{e \in P_R \mid vars(e) \cap sL_R = \emptyset\}$; and $\mathcal{P}_r^R$ is the set of return predicates, defined as $\{e \in \mathcal{P}^R \mid (r_R \in vars(e) \wedge (vars(e) \backslash r_R \cap sL_R = \emptyset)) \vee (e \in \mathcal{P}_f^R \wedge (vars(e) \cap \text{Globals}(\mathcal{C}) \neq \emptyset \vee drfs(e) \cap F_R \neq \emptyset))\}$.

The process of constructing $\mathcal{P}_c$ is known as predicate inference and the algorithm is described in Fig. 3.

The effectiveness of predicate abstraction relies critically on the set of predicates, so we select atomic proposition from given property ϕ as an initial candidate set of predicates. Assume that verify $\mathcal{C} \models \mathbf{AF}(x == 1)$ for C program in Fig. 1, and then $\mathcal{P} = \{(x == 1)\}$. Fig. 2 show the CFA with each location c labeled by $\mathcal{P}_c$. Then the algorithm of predicate inference begin with $\mathcal{P}_{l_6} = \{(x == 1)\}$. Form this it obtains $\mathcal{P}_{l_2^c} = \{(x == 1)\}$, and $\mathcal{P}_{l_9^c} = \{(x == 1)\}$, and lead to $\mathcal{P}_{l_1} = \mathcal{P}_{l_8} = \mathcal{P}_{l_7} = \{(x == 1)\}, \mathcal{P}_{l_0} = \{(10 == 0)\} = \emptyset$, other equal to $\emptyset$. So far we have described a method for computing the CFA and a set of predicates associated with each state of the CFA.

4.3. *Construction Initial Abstract Automaton*

We have described a method for computing the CFA of program and a set of predicates associated with each location of the CFA. The states of the abstract model $\mathcal{A}$ correspond to the various possible valuation of the predicate in each location. However, the generation of the abstract transition is done by calling a theorem prover for each potential

assignment to the current and next state predicates. In order to obtain the most precise transition relation, this requires an exponential number of calls to the theorem prover.

DEFINITION 4. The abstract model $\mathcal{A}$ of concrete program $\mathcal{C}$ is a 3-tuple $(S_{\mathcal{A}}, I_{\mathcal{A}}, T_{\mathcal{A}})$ where:

- $S_{\mathcal{A}} = C \times \mathcal{V}$ is the set of states,
- $I_{\mathcal{A}} = INIT$ is the initial state,
- $T_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ is the transition relation.

It is obvious that each location *c* of the CFA gives rise to a set of states of $\mathcal{A}$, $\{c\} \in \mathcal{V}_c$, where $\mathcal{V}_c$ is the set of all predicate valuations of $\mathcal{P}_c$. In the worst case, the size of $\mathcal{A}$ is exponential in the size of $\mathcal{P}$. In addition, $\mathcal{A}$ has a unique initial state *INIT*.

To compute the abstract transition relation it will often be necessary to determine whether two C expression *e* and e' are mutually exclusive using theorem prover. If we cannot prove that there is no transition between their corresponding concrete states, then a transition between two abstract states must be added. Therefore over-approximation occurs when not enough information is available for the theorem prover to calculate a deterministic next state. We can reduce to the problem of deciding whether $\neg(e \wedge e')$ is valid. The theorem prover return true on $\neg(e \wedge e')$, then *e* and *e*' are provable mutually exclusive, denoted $Mutu\text{-}Excl(e, e')$. Otherwise the theorem prover return false or beyond the capabilities of the theorem prover, then the theorem prover could not prove that *e* and e' are mutually exclusive, denoted $Mutu\text{-}Excl(e, e')$.

The remaining part of this section describes a procedure to compute the abstract transition relation. We can add $(\langle c, \vec{v}_c\rangle, \langle c', \vec{v}_{c'}\rangle)$ to $T_{\mathcal{A}}$ if and only if (c, c) T and one of the following conditions hold:

- $L(c)$ is the assignment statement and $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \mathcal{WP}(L(c), \gamma(\vec{v}_{c'})))$. In other words, $\gamma(\vec{v}'_{c'})$ and $\mathcal{WP}(L(c), \gamma(\vec{v}'_{c'}))$ can be not proved that they are mutually exclusive;
- $L(c)$ is the condition statement, $L(c')$ is its then successor, $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \gamma(\vec{v}'_{c'}))$ and $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), L(c))$;
- $L(c)$ is the condition statement, $L(c')$ is its else successor, $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \gamma(\vec{v}'_{c'}))$ and $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \neg L(c))$;
- $L(c)$ is the goto statement or the after-procedure return statement, $\vec{v}'_{c'} = \vec{v}_c$;
- $L(c)$ is the pre-procedure call statement, $c' = FirstProc(L(c))$ and $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \gamma(\vec{v}'_{c'}))$;
- $L(c)$ is the return statement. Let *c*' be the unique successor location of *c*. Note that $L(c')$ must be the exit procedure statement, then we include $(\langle c, \vec{v}_c\rangle, \langle c', \vec{v}_{c'}\rangle)$ in $T_{\mathcal{A}}$;
- $L(c)$ is the exit procedure statement. There are no outgoing transitions form *c* if the procedure is *main* function. Otherwise $\neg Mutu\text{-}Excl(\gamma(\vec{v}_c), \gamma(\vec{v}'_{c'}))$.

Consider the program described in Fig. 1. The abstract model for the program is shown in Fig. 4.

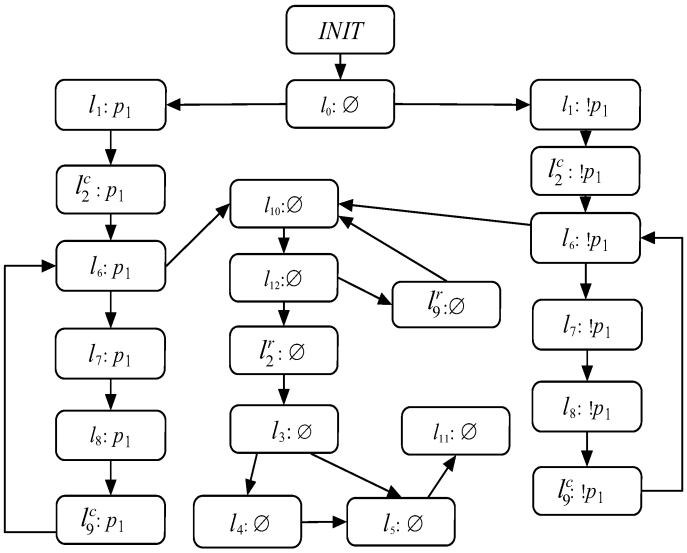

Fig. 4. The abstract model $\mathcal{A}$ of the program shown in Fig. 1.

4.4. *Parallel Composition*

When dealing with complex programs obtained by the parallel composition of simpler programs, the application of this method requires the computation of the corresponding global transition relation from which an abstraction can be computed. The question then arises whether it is possible to compute abstractions of complex programs as the parallel composition of abstractions of their components in order to avoid building the transition relation associated with the complex program. This is guaranteed if the compositionality property,

$$\frac{(\mathcal{C}^1 \prec \mathcal{A}^1) \text{ and } (\mathcal{C}^2 \prec \mathcal{A}^2)}{(\mathcal{C}^1 \parallel \mathcal{C}^2) \prec (\mathcal{A}^1 \parallel \mathcal{A}^2)}$$

holds, where $\parallel$ is a parallel composition operator.

DEFINITION 5 (parallel composition). The parallel composition of transition systems $\mathcal{A}^i = (S^i_{\mathcal{A}}, I^i_{\mathcal{A}}, T^i_{\mathcal{A}})$, $i = 1, 2$, denoted $\mathcal{A}^1 \parallel \mathcal{A}^2$, is the transition system $(S^{\parallel}_{\mathcal{A}}, I^{\parallel}_{\mathcal{A}}, T^{\parallel}_{\mathcal{A}})$, where:

- $S^{\parallel}_{\mathcal{A}} = S^1_{\mathcal{A}} \parallel S^2_{\mathcal{A}}$ is the set of states. $s = \langle c^1, c^2, \vec{v}^1_c, \vec{v}^2_c \rangle \in S^{\parallel}_{\mathcal{A}}$ iff $s^1 = \langle c^1, \vec{v}^1_c \rangle \in S^1_{\mathcal{A}}, s^2 = \langle c^2, \vec{v}^2_c \rangle \in S^2_{\mathcal{A}}$;
- $I^{\parallel}_{\mathcal{A}} = \langle INIT^1, INIT^2 \rangle$ is the initial state;
- $T^{\parallel}_{\mathcal{A}} = T^1_{\mathcal{A}} \parallel T^2_{\mathcal{A}}$ is the transition relation, $(s, s') \in T^{\parallel}_{\mathcal{A}}$ iff $(s, s') \in T^i_{\mathcal{A}}$, for $i = 1$or2.

5. Partitioning the Candidate Predicates

Since the construction of $\mathcal{A}$ from $\mathcal{C}$ involves predicate abstraction parameterized by a set of predicates $\mathcal{P}$, we write $\mathcal{A}(\mathcal{P})$ to denote the abstract model obtained via predicate abstraction from $\mathcal{C}$ using the set of predicates $\mathcal{P}$. For the sake of simplicity, we indicate this explicitly by referring to $\mathcal{A}^i$ as $\mathcal{A}(\mathcal{P}_i)$.

DEFINITION 6 (synchronous composition). The synchronous composition of transition systems $\mathcal{A}^i = (S^i_{\mathcal{A}}, I^i_{\mathcal{A}}, T^i_{\mathcal{A}})$, $i = 1, 2$, denoted $\mathcal{A}^1 \otimes \mathcal{A}^2$, is the transition system $(S^{\otimes}_{\mathcal{A}}, I^{\otimes}_{\mathcal{A}}, T^{\otimes}_{\mathcal{A}})$, where:

- $S^{\otimes}_{\mathcal{A}} = S^1_{\mathcal{A}} \otimes S^2_{\mathcal{A}} \subseteq C \times \mathcal{V}^1 \times \mathcal{V}^2$ is the set of states. $s = \langle c, \vec{v}^1_c, \vec{v}^2_c \rangle \in S^{\otimes}_{\mathcal{A}}$ iff $s^1 = \langle c^1, \vec{v}^1_c \rangle \in S^1_{\mathcal{A}}, s^2 = \langle c^2, \vec{v}^2_c \rangle \in S^2_{\mathcal{A}}$ and $c^1 = c^2 = c$, i.e., the states that have the same control location can be composed;
- $I^{\otimes}_{\mathcal{A}} = INIT^1 \times INIT^2$ is the initial state;
- $T^{\otimes}_{\mathcal{A}} = T^1_{\mathcal{A}} \otimes T^2_{\mathcal{A}} \subseteq S^{\otimes}_{\mathcal{A}} \otimes S^{\otimes}_{\mathcal{A}}, (s, s') \in T^{\otimes}_{\mathcal{A}}$ iff $(s_{|S^i_{\mathcal{A}}}, s'_{|S^i_{\mathcal{A}}}) \in T^i_{\mathcal{A}}$, for *i* = 1, 2, where $s_{|S^i_{\mathcal{A}}}$ denotes the restriction of the state *s* to $S^i_{\mathcal{A}}$.

Theorem 2. *Assume that two transition systems* $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, T_{\mathcal{A}})$ *and* $\tilde{\mathcal{A}} = (\tilde{S}_{\mathcal{A}}, \tilde{I}_{\mathcal{A}}, \tilde{T}_{\mathcal{A}})$ *satisfy* $S_{\mathcal{A}} \subseteq \tilde{S}_{\mathcal{A}}$ *and* $T_{\mathcal{A}} \subseteq \tilde{T}_{\mathcal{A}}$*, then* $\tilde{\mathcal{A}}$ *simulates* $\mathcal{A}$*, i.e.,* $\mathcal{A} \preceq \tilde{\mathcal{A}}$.

Proof. According to the definition of simulation relation, the proposition can be proofed easily.

It is usually computationally expensive to compute predicate abstraction directly with respect to $\mathcal{P}$. In the worst case, the size of the state space in abstract model $\mathcal{A}$ is exponential in the size of $\mathcal{P}$. Computing the transitions $T_{\mathcal{A}}$ between the states requires a theorem prover. Therefore, the worst time complexities for checking validities are exponential as well. Instead of building $\mathcal{A}$ directly, Approximation is often used to reduce the complexity. If a transition systems $\tilde{\mathcal{A}} = (S_{\mathcal{A}}, I_{\mathcal{A}}, \tilde{T}_{\mathcal{A}})$ satisfies $T_{\mathcal{A}} \subseteq \tilde{T}_{\mathcal{A}}$, then we say that $\tilde{\mathcal{A}}$ approximates $\mathcal{A}$, denoted $\mathcal{A} \preceq \tilde{\mathcal{A}}$. Intuitively, if $\tilde{\mathcal{A}}$ approximates $\mathcal{A}$, then $\tilde{\mathcal{A}}$ is more abstract than $\mathcal{A}$, i.e., has more behaviours than $\mathcal{A}$.

In order to make our method effective, the set of candidate predicates $\mathcal{P}$ can be partitioned into a number of subsets $\mathcal{P}_1, ..., \mathcal{P}_n$, abstracted independently. Therefore, we only need to consider the effect of the abstraction of concrete program $\mathcal{C}$ on each subset $\mathcal{P}_i$ separately, instead of the full abstract model on the set of predicates $\mathcal{P}$. If we have *n* candidate predicates and partition them into two sets of n_1 and n_2 elements,respectively, we only have to check for $2^{n_1} + 2^{n_2}$ validities instead of 2^n validities.

We say that two predicates interfere with each other if there exists a variable appearing in both of them. Let $\equiv$ be the equivalence relation over $\mathcal{P}$ which is the reflexive, transitive closure of the interference relation. The equivalence class of a predicate $p \in \mathcal{P}$ is denoted by $[p]$. If two predicate p_1 and p_2 have non-disjoint set of variables, then $[p_1] = [p_2]$. That is, a variable cannot occur in the different equivalence class of predicates. By the equivalence relation $\equiv$, the set of predicates can be partitioned into some subsets.

Assume each $T^i_{\mathcal{A}}$ defines the transition relation for a predicate subset $\mathcal{P}_i$. Then, we apply abstraction to each separately, i.e., $\tilde{T}_{\mathcal{A}} = T^1_{\mathcal{A}} \otimes ... \otimes T^n_{\mathcal{A}}$. Finally, $\tilde{\mathcal{A}}$ is given by $\bigotimes_{i\in[1,n]} \mathcal{A}^i$. In general the abstract system computed using a partitioning has more transitions than the system $\mathcal{A}$ computed without using the partitioning. However, assume that the set $\mathcal{P}$ of candidate predicates can be partitioned into sets $\mathcal{P}_1, ..., \mathcal{P}_n$ such that $T_{\mathcal{A}}$ can be written in the form $T^1_{\mathcal{A}} \otimes ... \otimes T^n_{\mathcal{A}}$, then, $\mathcal{A} = \tilde{\mathcal{A}} = \bigotimes_{i\in[1,n]} \mathcal{A}^i$ holds.

Theorem 3. *Assume that $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, T_{\mathcal{A}})$ denotes the abstract model obtained via predicate abstraction from $\mathcal{C}$ using the set of predicates $\mathcal{P}$, and $\mathcal{A}^i = (S^i_{\mathcal{A}}, I^i_{\mathcal{A}}, T^i_{\mathcal{A}})$ denotes the abstract model obtained using the set of predicates $\mathcal{P}_i$, such that the set of predicates $\mathcal{P}_i$ is a partition of $\mathcal{P}$ (satisfying the cover property $\mathcal{P} = \bigvee_{i\in[i,n]} \mathcal{P}_i$ and the disjoint property $\forall 1 \leqslant i, j \leqslant n, i \neq j \Rightarrow \mathcal{P}_i \cap \mathcal{P}_i = \emptyset$), let $\tilde{\mathcal{A}} = \bigotimes_{i\in[1,n]} \mathcal{A}^i$, then $A = \tilde{\mathcal{A}}$.*

Proof. Let $\tilde{\mathcal{A}} = \bigotimes_{i\in[1,n]} \mathcal{A}^i = (S^{\otimes}_{\mathcal{A}}, I^{\otimes}_{\mathcal{A}}, T^{\otimes}_{\mathcal{A}})$, where $S^{\otimes}_{\mathcal{A}} = \bigotimes_{i\in[1,n]} S^i_{\mathcal{A}}$, $I^{\otimes}_{\mathcal{A}}$ = $INIT_1 \times ... \times INIT_n$, and $T^{\otimes}_{\mathcal{A}} = \bigotimes_{i\in[1,n]} T^i_{\mathcal{A}}$. $\mathcal{A} = \tilde{\mathcal{A}}$ if and only if the following conditions holds:

- According to the definition of $S^{\otimes}_{\mathcal{A}}$, $s = \langle c, \vec{v}^1_c, ..., \vec{v}^n_c\rangle \in S^{\otimes}_{\mathcal{A}} \Leftrightarrow s = \langle c, \vec{v}_c\rangle \in S_{\mathcal{A}}$ such that $\vec{v}_c = \langle \vec{v}^1_c, ..., \vec{v}^n_c\rangle$, so $S_{\mathcal{A}} = S^{\otimes}_{\mathcal{A}}$.
- $I^{\otimes}_{\mathcal{A}} = INIT^1 \times ... \times INIT^n = INIT = I_{\mathcal{A}}$ is the initial state.
- Let $s = \langle c, \vec{v}^1_c, ..., \vec{v}^n_c\rangle$ and $s' = \langle c', \vec{v'}^1_c, ..., \vec{v'}^n_c\rangle$, for all $1 \leqslant i \leqslant n, (\langle c,\vec{v}^i_c\rangle, \langle c', \vec{v'}^i_c\rangle) \in T^i_{\mathcal{A}}$, i.e., $(s_{|S^i_{\mathcal{A}}}, s'_{|S^i_{\mathcal{A}}}) \in T^i_{\mathcal{A}}$, then $(s, s') \in T^{\otimes}_{\mathcal{A}} \Leftrightarrow (s, s') \in T_{\mathcal{A}}$, so $T_{\mathcal{A}} = T^{\otimes}_{\mathcal{A}}$.

6. Refinement

The abstraction model $\mathcal{A}$ is checked automatically for the desired property ϕ. If $\mathcal{A} \models \phi$, then the verification is successful; otherwise, an abstract counterexample is automatically produced, and checked to see whether the counterexample is spurious. In this section we present our approach to checking the validity of an abstract counterexample, and refining our abstraction on the basis of the spurious counterexample.

6.1. *Counterexample Validation*

A trace τ of $\mathcal{A}$ is a finite sequence $\tau = \langle (c_1, \vec{v}_1), ..., (c_n, \vec{v}_n)\rangle$ such that (i) for $1 \leqslant i \leqslant n, (c_i, \vec{v}_i) \in S_{\mathcal{A}}$, (ii) $(c_1, \vec{v}_1) \in I_{\mathcal{A}}$, and (iii) for $1 \leqslant i \leqslant n, ((c_i, \vec{v}_i), (c_{i+1}, \vec{v}_{i+1})) \in T_{\mathcal{A}}$. Each such trace τ gives rise to a corresponding concrete execution path $\gamma(\tau) = \langle L(c_1), ..., L(c_n)\rangle$ in the actual C program. Given τ one can efficiently construct $\gamma(\tau)$ and then check its validity, i.e., whether $\gamma(\tau)$ is actually possible in any execution of the C program.

Given a abstract counterexample τ, there are two approaches to checking whether $\gamma(\tau)$ is a valid trace of $\mathcal{C}$- the backward traversal using weakest preconditions (Chaki,

2003a) and the forward traversal based on strongest postconditions (Ball, 2002). We use the backward traversal approach. Let $\gamma(\tau) = \langle L(c_1), ..., L(c_n)\rangle$. We compute a set of expressions e_i for $1 \leqslant i \leqslant n$, and start with $e_{n+1} = true$ and let e_i be the weakest precondition of e_{i+1} with respect to $L(c_i)$. Then $\gamma(\tau)$ is valid iff e_1 is true.

If the abstract counterexample τ is not spurious, report the concretion counterexample and stop; otherwise discover and add predicates to refine abstract model by the spurious counterexample τ.

6.2. *Refining Abstract Model*

The effectiveness of predicate abstraction relies critically on the set of predicates, so we try to discover suitable sets of predicates that is enough to prove or disprove a safety property ϕ. The process of extracting a finite model from a C program using predicate abstraction can be exponential in the number of predicates used. In order to make model construction effective, one must eliminate redundant predicates and keep the set of predicates as small as possible (Clarke *et al.*, 2003).

If the abstract counterexample τ is spurious, by combining the forward and backward approaches we can compute a minimal spurious sub-path $\langle L(c_i), ..., L(c_j)\rangle$ of $\gamma(\tau)$ similar to (Das and Dill, 2002). In particular, there might be several spurious counterexample paths, and each spurious path could yield several predicates. However some paths might share common spurious sub-paths. In the case we can use the minimal spurious sub-paths to avoid significant redundant computation.

Given a set of spurious counterexamples $(\tau_1, ..., \tau_k)$, let $\mathcal{P}_i$ be the set of e_i discovered while checking for the validity of $\gamma(\tau_i)$. We know that the set $\mathcal{P} = \bigcup_{i=1}^{k} \mathcal{P}_i$ will definitely eliminate each τ_i. Note that a minimal predication has been discussed in (Chaki *et al.*, 2003b).

Finally, we address the termination properties of the refinement process. The process will terminate if one of the two following conditions is met:

- model checking the abstract model shows that $\mathcal{A} \models \phi$;
- a real counterexample path that is feasible in $\mathcal{C}$ is found.

7. Conclusion and Future Work

Our intention is to check whether a sequential program $\mathcal{C}$ satisfies a temporal logic property ϕ. The problem of verifying that a concrete program satisfies a temporal formula ϕ is reduced to the problem of verifying an abstract model derived from the program. If a property holds on the abstract structure, it also holds on the concrete program.

Our method can be extended to verifying concurrency C programs by abstract model parallel composition. There are many interesting research directions for further work: (1) handle concurrency; (2) extend to OO languages like Java and C++.

References

Alur, R., A. Itai, R.P. Kurshan and M. Yannakakis (1995). Timing verification by successive approximation. *Information and Computation*, **118**(1), 142–157.

Ball, T., R. Majumdar, T.D. Millstein and S.K. Rajamani (2001). Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*. pp. 203–213.

Ball, T., and S.K. Rajamani (2001). Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, *LNCS*, vol. 2057. pp. 103–122.

Ball, T., and S.K. Rajamani (2002). Generating abstract explanations of spurious counterexamples in C programs. *Technical Report MSR-TR-2002-09*, Microsoft Research, Redmond.

Bensalem, S., Y. Lakhnech and S. Owre (1998). Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of Computer Aided Verification*, *LNCS*, vol. 1427. pp. 19–331.

Chaki, S., E.M. Clarke, A. Groce, S. Jha and H. Veith (2003a). Modular verification of software components in C. In *Proceedings of ICSE*. pp. 385–395.

Chaki, S., E.M. Clarke, A. Groce and O. Strichman (2003b). Predicate abstraction with minimum predicates. In *Proceedings of CHARME*. pp. 19–34.

Chaki, S., E.M. Clarke, A. Groce, J. Ouaknine, O. Strichman and K. Yorav (2004). Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, **25**(2–3), 129–166.

Clarke, E.M., O. Grumberg and D.E. Long (1994). Model checking and abstraction. In *Proceedings of TOPLAS*. pp. 1512–1542.

Clarke, E.M., O. Grumberg and D.A. Peled (1999). *Model Checking*. MIT Press.

Clarke, E.M., O. Grumberg, S. Jha, Y. Lu and H. Veith (2000). Counterexample-guided abstraction refinement. In *Proceedings of Computer Aided Verification*. pp. 154–169.

Clarke, E.M., O. Grumberg, M. Talupur and D. Wang (2003.). Making predicate abstraction efficient: how to eliminating redundant predicates. In *Proceedings of Computer Aided Verification*, *LNCS*, vol. 2725. pp. 126–140.

Corbett, J.C., M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby and H. Zheng (2000). Bandera: extracting finite-state models from Java source code. In *Proceedings of ICSE*. pp. 439–448.

Cousot, P., and R. Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*. pp. 238–252.

Das, S., D.L. Dill and S. Park (1999). Experience with predicate abstraction. In *Proceedings of Computer Aided Verification*, *LNCS*, vol. 1633. pp. 160–171.

Das, S., and D.L. Dill (2002). Counter-example based predicate discovery in predicate abstraction. In *Proceedings of Formal Methods in Computer-Aided Design*. pp. 19–32.

Graf, S., and H. Saidi (1997). Construction of abstract state graphs with PVS. In *Proceedings of Computer Aided Verification*, *LNCS*, vol. 1254. pp. 72–83.

Havelund, K., and T. Pressburger (2000). Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, **2**(4), 366–381.

Henzinger, T.A., R. Jhala, R. Majumdar and G. Sutre (2002). Lazy abstraction. In *Proceedings of POPL*. pp. 58–70.

Holzmann, G.J. (2000). Software model checking. In *NATO Summer School*, Marktoberdorf, Germany, vol. 180. pp. 309–355.

Lakhnech, Y., S. Bensalem, S. Berezin and S. Owre (2001). Incremental verification by abstraction. In *Proceedings of TACAS 2001*, *LNCS*, vol. 2031. pp. 98–112.

Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem (1995). Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, **6**(1), 11–44.

Necula, G.C., S. McPeak, S.P. Rahul and W. Weimer (2002). CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Compiler Construction*, *LNCS*, vol. 2304. pp. 213–228.

Saidi, H. (2000). Model checking guided abstraction and analysis. In *Proceedings of SAS 2000*, *LNCS*, vol. 1824. pp. 377–396.

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, **3**(3), 121–189.

J. Qian received the MS degrees in computer science and engineering from Guilin University of Electronic Technology, Guilin, China. Now he is a doctor candidate at Southeast University, Nanjing, China. His research area includes software verification and model checking etc.

B. Xu received the PhD degrees in computer science and engineering from Beihang University, Beijing, China, in 2002. He is currently a professor in Department of Computer and Engineering at Southeast University, Nanjing, China. His research area includes program analysis, program design, and software engineering etc. Up to now (1996–2006), more than 50 high quality papers on software engineering have been published in journals.

C programų formalus verifikavimas

Junyan QIAN, Baowen XU

Darbe pateiktas programų C kalboje formalus verifikavimo metodas. Norint nustatyti, ar programa tenkina duotą specifikaciją, pirmiausia iš šios programos yra konstruojamas abstraktus programos modelis. Perėjimų abstrakčiame modelyje konstravime naudojamas teoremų įrodymo metodas. Naudojant modelio tikrinimo algoritmą, nustatoma, ar abstraktus modelis tenkina specifikaciją. Jei gaunamas kontrapavyzdys, tai jis naudojamas abstraktaus modelio tobulinimui.