ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.9, September 2007, pp.2130−2140
DOI: 10.1360/jos182130

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于Assume-Guarantee搜索复用的C程序验证方法*

易晓东[1+], 王 戟[1,2], 杨学军[1,2]

[1](国防科学技术大学 计算机学院,湖南 长沙 410073)

[2](并行与分布处理国防科技重点实验室,湖南 长沙 410073)

Verification of C Programs Using Assume-Guarantee Reuse of Searching

YI Xiao-Dong[1+], WANG Ji[1,2], YANG Xue-Jun[1,2]

[1](College of Computer, National University of Defense Technology, Changsha 410073, China)

[2](National Defense Laboratory for Parallel and Distributed Processing, Changsha 410073, China)

+ Corresponding author: E-mail: yi_xiao_dong@sohu.com

Yi XD, Wang J, Yang XJ. Verification of C programs using assume-guarantee reuse of searching. *Journal of Software*, 2007,18(9):2130−2140. http://www.jos.org.cn/1000-9825/18/2130.htm

Abstract: This paper presents a novel method, namely Assume-Guarantee reuse of searching, to verify C programs with respect to temporal safety properties. Its idea is to introduce a conservative assume condition for each program location, and to assume that every path starting from the program location will never violate the property if the evaluation of its variables at that location satisfy the assume condition. All the possible execution paths are traversed based on the assume conditions, and the temporal safety property is checked on the fly. If some assume condition is too weak, it will be continually strengthened based on the spurious counterexamples. The presented verification method can try to adopt the weak assume conditions so as to let more execution paths satisfy the conditions and to reuse the searching efforts. Therefore, a significant reduction of verification cost can be achieved. The verification method has been used to verify the initial handshake process of SSL protocol based on the C source code of openssl-0.9.6c. The experimental results demonstrate that the method is both effective and practical.

Key words: Assume-Guarantee reuse of searching; variable abstraction; approximated program semantics; partial strongest post-condition

摘 要: 提出了一种基于 Assume-Guarantee 搜索复用的验证方法,对 C 程序源代码进行验证.其思想是,在程序的每点处都引入一个保守假设条件,并假设从任意点出发,变量取值满足该点假设条件的所有执行路径都不会违背给定性质,然后根据这些假设条件遍历所有可能的执行路径以验证给定的时序安全性质,并在遍历的过程中验证这些假设条件是否满足,如果不满足,则不断对其精化和加强.验证方法总是在保证假设条件可靠的前提下尽量使用较弱的条件,使得大量的执行路径由于满足假设条件而可以搜索复用,从而降低验证代价.应用该方法验证了 Linux 操作

* Supported by the National Natural Science Foundation of China under Grant No.60233020 (国家自然科学基金); the National High-Tech Ressearch and Development Plan of China under Grant No.2006AA01Z429 (国家高技术研究发展计划(863)); the Program for New Century Excellent Talents in University under Grant No.NCET-04-0996 (新世纪优秀人才支持计划)

Received 2006-05-28; Accepted 2006-10-10

系统中 SSL 协议的实现程序 openssl-0.9.6c 满足 SSL 协议的初始握手规范.实验结果表明,该方法具有良好的实用性和可扩展性.

关键词: Assume-Guarantee 搜索重用;变量抽象;程序近似语义;部分最强后置条件

中图法分类号: TP311 **文献标识码**: A

近年来,直接面向程序源代码的验证方法由于其方便直接、无须建模和简洁实用等优点而受到了越来越多的关注,特别是随着谓词抽象(predicate abstraction)[1]理论的提出,基于该理论的SLAM[2,3],BLAST[4],MAGIC[5,6]及ComFoRT[7,8]等多个研究工作使得面向程序源代码的验证方法取得了显著进展.但是,由于软件系统的复杂性,特别是程序的状态空间爆炸问题,使得对实用程序的验证仍面临代价过高的困难,降低验证代价是面向程序源代码的验证技术实用化的关键.本文的研究将Assume-Guarantee的思想应用于对C程序的验证,通过对验证信息充分复用,能够显著降低面向C程序源代码的验证开销,从而增强验证的可扩展性和实用化程度.

Assume-Guarantee 推理(assume-guarantee reasoning)广泛地应用于程序模块化组合验证,并取得了良好的效果,如文献[7,9,10]等.其基本思想是,在验证多个程序模块或多个并发进程时,可以对这些模块或进程的上下文或输入、输出参数进行假设,基于这些假设条件分别对各个单独的模块或进程进行验证,然后验证这些假设条件得到满足即可,这种分而治之的方法往往能大幅度地降低验证开销.本文借鉴了 Assume-Guarantee 推理的思想,在顺序 C 程序源代码的验证中,通过“搜索复用”降低验证开销,提出了一种基于 Assume-Guarantee 搜索复用的验证方法(本文以下简称为 Assume-Guarantee 搜索复用方法).该方法先在程序源代码的每一点处引入一个保守的假设条件,并假设从该点出发,变量取值满足假设条件的所有执行路径都不会违背给定性质,然后根据这些假设条件遍历程序的所有可能执行路径.我们保证程序任意一点处的假设条件总是比在该点处变量实际满足的条件要弱(weak),这样可以保证遍历过程中不会漏掉任何一条程序执行路径.但同时,太弱的假设条件也会导致我们遍历一些在实际程序中并不存在的执行路径,如果某些这种实际不存在的执行路径违背了给定性质,我们就对假设条件进行精化和加强.如果程序各点处的假设条件都被证明是可靠的,且满足这些假设条件的执行路径都不违背给定性质,那么程序也被证明满足给定的性质.

在保证假设条件可靠的前提下,上述Assume-Guarantee搜索复用方法总是试图为每个程序点引入尽可能弱的假设条件,这是由于假设条件越弱,就会有越多的程序路径执行到该点时其变量取值满足假设条件,如果假设条件是可靠的,这些路径就不需要继续遍历下去,从而更多地减小验证代价.为了找到尽可能弱的假设条件,我们引入了一种新的抽象方法,即变量抽象(variable abstraction),其思想与传统的静态程序切片(program slicing)[11]相似,即根据程序中每条语句所包含的变量决定该语句是否与待验证的性质相关,但它省去了依赖关系计算,其代价要比程序切片小得多.如果只考虑变量抽象下的相关语句的行为,就得到了程序的近似语义.程序的近似语义刻画了这样一个事实,即验证过程中人们所关注的性质往往只涉及到程序中的一小部分变量和行为,特别是在实用程序和大规模程序中更是如此,这个事实也用于基于程序切片的程序辅助调试方法中,并得到了广泛的应用.我们也成功地将程序的近似语义与传统的符号执行相结合,提出了切片执行(slicing execution)[12,13]的概念,并将其用于对C程序源代码的验证,取得了很好的效果.

近似语义下的最强后置条件,定义为部分最强后置条件(partial strongest post-condition),只考虑程序的部分行为和变量,因此比传统精确语义下的最强后置条件要弱.我们使用部分最强后置条件作为每个程序点的假设条件,并使用反例指导的抽象精化(counter-example guided abstraction refinement,简称CEGAR)[14]方法,根据验证过程中发现的违背性质、但实际不存在的路径对变量抽象的抽象准则(abstraction criterion)进行精化,然后根据精化的抽象准则对部分最强后置条件进行精化和加强,以得到精化和加强的假设条件.

我们使用 Assume-Guarantee 搜索复用方法对 SSL 协议的初始握手过程进行了验证,验证所使用的 C 程序来自于 Linux 操作系统下 SSL 协议的实现程序 openssl-0.9.6c.该验证案例也被基于谓词抽象的验证工具 BLAST 和 MAGIC 以及基于切片执行的验证工具验证过,我们列举了它们各自的验证结果,并与本文的搜索复用方法的验证结果进行了比较和分析.结果表明,Assume-Guarantee 搜索复用方法具有良好的实用性和可扩

展性.

本文第 1 节给出程序的近似语义,变量抽象、部分最强后置条件等概念也在该节中给出.第 2 节详细介绍 Assume-Guarantee 搜索复用方法,并对其进行理论分析与证明.第 3 节介绍验证工具的实现,以及对实验结果的比较和分析.第 4 节讨论相关工作.最后是结束语.

1 程序的近似语义

与BLAST和MAGIC一样,我们假设C程序不含递归调用和非直接跳转.经过被调用函数内嵌(inline)、while/for/switch等语句的改写后,可以假设C程序中只包含 4 种语句,即赋值语句、if-then-else分支语句、goto语句和return语句.分支语句"if(c) then A else B;"中的分支条件if(c)根据其后继语句是A还是B分别被替换为assume语句$assume(c)$和$assume(\neg c)$.注意,此处的assume语句是用来替换C程序中的分支语句if,以直观地描述分支条件的,并不是本文Assume-Guarantee搜索复用方法中的Assume条件.程序的精确语义可以由最强后置条件(strongest post-condition)[15]来定义,赋值语句和assume语句的最强后置条件分别定义为[3,15]

$$SP(x:=e)=\lambda f.\exists x'.f[x'/x]\wedge(x=e[x'/x]) \tag{1}$$

$$SP(assume(c))=\lambda f.f\wedge c \tag{2}$$

其中,$f[x'/x]$和$e[x'/x]$分别是将公式f和表达式e中的所有x的自由出现都使用x'替换后得到的公式和表达式.根据文献[3]中的方法,我们可以通过在计算过程中引入Skolem常量的方法来消除最强后置条件中的存在量词($\exists$).对于赋值语句$x:=e$和assume语句$assume(c)$,如果表达式e或c中引用了未定义的变量y,则引入一个Skolem常量θ_y来符号化地表示y此刻的值,并在上述语句之前插入一条赋值语句$y:=\theta_y$.借鉴文献[3],我们使用二元组$\langle\Omega,\Phi\rangle$来表示和计算最强后置条件,其中,$\Omega$是一个部分函数,将程序变量映射到由Skolem常量组成的表达式,即$\Omega:Vars\mapsto Exp$ ($Vars$和Exp分别代表程序变量和表达式的全集).我们可以按通常的方法将Ω提升到表达式集合,即$\Omega:Exp\mapsto Exp$;Φ是一个集合,存储了所有assume语句引入的布尔表达式.基于$\langle\Omega,\Phi\rangle$的赋值语句和assume语句的最强后置条件的计算方法定义如下[3]:

$$SP(x:=e)=\lambda\langle\Omega,\Phi\rangle.\langle\Omega[x\rightarrow\Omega(e)],\Phi\rangle \tag{3}$$

$$SP(assume(c))=\lambda\langle\Omega,\Phi\rangle.\langle\Omega,\Phi\cup\Omega(c)\rangle \tag{4}$$

其中,函数$\Omega[x\rightarrow e]$定义为

$$\Omega[x\rightarrow e](y)=\begin{cases}\Omega(y), & \text{if } y\neq x\\ e, & \text{if } y=x\end{cases} \tag{5}$$

经典的最强后置条件给出了程序的精确语义,即描述了执行某个程序语句后变量取值所满足的条件.但是在验证过程中,待验证的性质所涉及的程序变量往往较少,我们只需考虑少量的程序变量,从而对程序行为进行抽象.在本文中,我们通过引入变量抽象和部分最强后置条件对程序语义进行保守近似(over-approximation)[12,13].变量抽象只关注变量集合V中的部分程序变量,相应地产生部分最强后置条件(partial strongest post-condition),记为SP_V.

变量抽象与传统的程序切片(program slicing)的思想比较类似,但目标和计算方法不同.变量抽象的目标和计算方法是基于一个抽象准则(abstraction criterion,一个变量集合V)来判断一条程序语句是否是与性质验证相关的语句,判断的方法是比较程序语句中包含的变量是否是抽象准则变量集合V的子集.我们可以看出,因为它省去了依赖关系的计算,从而其计算开销比传统程序切片要小得多.定义函数$Vars(e)$返回表达式e中包含的所有变量的集合,例如$Vars(x>y)=\{x,y\}$.对于一条程序路径$p=s_1,s_2,\ldots,s_n$,定义

$$Vars(p)=\bigcup_{1\le i\le n}Vars(s_i).$$

定义 1(赋值语句的相关性). 赋值语句 $x:=e$ 在抽象准则 V 下的相关性定义如下:

- 如果 $Vars(x)\subseteq V$ 并且 $Vars(e)\subseteq V$,则称 $x:=e$ 是抽象准则 V 下的完全相关赋值语句;
- 如果 $Vars(x)\subseteq V$ 但 $Vars(e)\not\subseteq V$,则称 $x:=e$ 是抽象准则 V 下的部分相关赋值语句;

- 如果 $Vars(x)\not\subseteq V$,则称 $x:=e$ 是抽象准则 V 下的无关赋值语句.

定义 2(Assume 语句的相关性). *assume* 语句 $assume(c)$在抽象准则 V 下的相关性定义如下:

- 如果 $Vars(c)\subseteq V$,则称 $assume(c)$是抽象准则 V 下的相关 *assume* 语句;
- 如果 $Vars(c)\not\subseteq V$,则称 $assume(c)$是抽象准则 V 下的无关 *assume* 语句.

基于变量抽象,我们定义部分最强后置条件作为程序的近似语义.

定义 3(赋值语句的部分最强后置条件). 给定抽象准则V,赋值语句$x:=e$在抽象准则V下的部分最强后置条件$SP_V(x:=e)$定义为

- 如果$x:=e$是V下完全相关的赋值语句,那么$SP_V(x:=e)=SP(x:=e)$;
- 如果$x:=e$是V下部分相关的赋值语句,那么$SP_V(x:=e)=SP(x:=\theta_e)$;
- 如果$x:=e$是V下无关的赋值语句,那么$SP_V(x:=e)$是恒等函数.

如果$x:=e$是部分相关的赋值语句,由于没有考察表达式e的某些变量,我们无法确知e的值,故而引入一个Skolem常量θ_e,表示表达式e可取任意值.

定义 4(Assume语句的部分最强后置条件). 给定抽象准则V,*assume*语句$assume(c)$在抽象准则V下的最强后置条件$SP_V(assume(c))$定义为

- 如果$assume(c)$是V下相关的*assume*语句,那么$SP_V(assume(c))=SP(assume(c))$;
- 如果$assume(c)$是V下无关的*assume*语句,那么$SP_V(assume(c))$是恒等函数.

一条路径$p=s_1,s_2,\ldots,s_n$的部分最强后置条件$SP_V(p)$定义为$SP_V(p)=SP_V(s_n)\circ SP_V(s_{n-1})\circ\ldots\circ SP_V(s_1)$,其中函数合成符号"$\circ$"定义为从右向左组合,即$g\circ h=\lambda x.g(h(x))$.部分最强后置条件的定义基于传统最强后置条件给出,我们同样基于二元组$\langle\Omega,\Phi\rangle$来计算部分最强后置条件.通过定义如下函数$h$,可将$\langle\Omega,\Phi\rangle$转换为一阶逻辑表达式:

$$h(\langle\Omega,\Phi\rangle)=\exists\theta_1\ldots\theta_m.\left(\left(\bigwedge_{\omega\in\Omega}e2b(\omega)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right) \tag{6}$$

其中,$\theta_1,\ldots,\theta_m$是在计算过程中引入的所有Skolem常量,函数$e2b(\omega)$将部分函数Ω的每个变量或表达式映射转换为一个布尔表达式,例如:$e2b(x\to 1)=(x==1)$.

2 基于 Assume-Guarantee 搜索复用的 C 程序验证方法

2.1 Assume-Guarantee搜索复用方法

本文使用有限状态自动机(finite state automata,简称 FSA)来描述待验证的性质.程序入口语句对应于性质自动机的初始态,程序中的特定语句、特定操作或特定事件作为性质自动机的状态迁移事件,例如加锁、通信、对变量的某种操作或到达程序的某种状态等.我们规定:如果程序的某条执行路径使得性质自动机的状态进入其接收态,那么性质被违背.换言之,只有程序的所有执行路径都不能使得性质自动机的状态进入接收态时,性质才得到满足.

为了验证C程序是否满足性质自动机,我们将部分最强后置条件从二元组$\langle\Omega,\Phi\rangle$扩展到三元组$\langle\Omega,\Phi,\varepsilon\rangle$,其中,$\varepsilon$记录了性质状态机的状态.相应地,我们也将部分最强后置条件的计算$SP_V(s)$扩展到$\langle\Omega,\Phi,\varepsilon\rangle$上,定义$\langle\Omega',\Phi',\varepsilon'\rangle=SP_V(s)(\langle\Omega,\Phi,\varepsilon\rangle)$,其中,$\langle\Omega',\Phi'\rangle=SP_V(s)(\langle\Omega,\Phi\rangle)$,且$\varepsilon'$是性质状态机从状态$\varepsilon$经过语句$s$迁移到的新状态(如果$s$是性质状态机的一个状态迁移事件).需要注意的是,无论语句s是否为相关语句,只要它是性质状态机的状态迁移事件,就需要更新$\langle\Omega,\Phi,\varepsilon\rangle$中的$\varepsilon$.

定义 5(Assume条件). 给定抽象准则V,对于一条从程序入口语句到某程序语句s的路径p,定义部分最强后置条件$SP_V(p)$(True)是在语句s处的Assume条件,其中,True定义为$\langle\Omega_0,\Phi_0,\varepsilon_0\rangle$,$\Omega_0=\Phi_0=\varnothing$, ε_0 为性质状态机的初始态.

定义 6(Assume条件的冗余性). 程序语句s处的某个Assume条件$\langle\Omega,\Phi,\varepsilon\rangle$相对于$s$处的某个其他的Assume条件集合$PSP$是冗余的(记为$\langle\Omega,\Phi,\varepsilon\rangle\Rightarrow PSP$),当且仅当存在$PSP$的一个子集$\{\langle\Omega_1,\Phi_1,\varepsilon_1\rangle,\ldots,\langle\Omega_n,\Phi_n,\varepsilon_n\rangle\}\subseteq PSP$,使得$\varepsilon_1=\ldots=\varepsilon_n=\varepsilon$且式(7)成立,其中,$v_1,\ldots,v_m$是抽象准则$V$中的所有变量,函数$h$的定义见式(6).

$$\forall v_1...v_n.\left(h(\langle \Omega,\Phi\rangle) \Rightarrow \bigvee_{1\le i\le n} h(\langle \Omega_i,\Phi_i\rangle)\right) \tag{7}$$

$\langle \Omega,\Phi,\varepsilon\rangle \Rightarrow PSP$可以直观地理解为,$PSP$描述了比$\langle \Omega,\Phi,\varepsilon\rangle$更弱的Assume条件.换言之,$\langle \Omega,\Phi,\varepsilon\rangle$描述的所有可能的变量取值都包含在$PSP$描述的合法变量取值范围之内,从而$PSP\cup\{\langle \Omega,\Phi,\varepsilon\rangle\}$描述的Assume条件等价于$PSP$,因此称$\langle \Omega,\Phi,\varepsilon\rangle$相对于$PSP$是冗余的.如果$\langle \Omega,\Phi,\varepsilon\rangle \Rightarrow PSP$成立,记$\Rightarrow_{\langle \Omega,\Phi,\varepsilon\rangle}(PSP)$为$\langle \Omega,\Phi,\varepsilon\rangle$所蕴含的$PSP$的子集,即$PSP$的上述子集$\{\langle \Omega_1,\Phi_1,\varepsilon_1\rangle,\ldots,\langle \Omega_n,\Phi_n,\varepsilon_n\rangle\}$.需要指出的是,该子集可能并不是唯一的,在实践中我们往往选择一个尽可能小的集合.

定义 7(Assume 条件的可靠性). 设$\langle \Omega,\Phi,\varepsilon\rangle$是在程序语句 s 处的一个 Assume 条件,如果基于任意满足$h(\langle \Omega,\Phi\rangle)$的变量取值从$s$开始执行,都不能使性质状态机从状态$\varepsilon$迁移到其接收态,则称$\langle \Omega,\Phi,\varepsilon\rangle$是语句$s$处的一个可靠(sound)的 Assume 条件.

如果$\langle \Omega,\Phi,\varepsilon\rangle$是语句 s 处一个可靠的 Assume 条件,那么满足它的所有变量取值和性质状态机当前状态的、从 s 开始的执行路径都不会违反所给性质.

定义 8(Assume条件之间的依赖关系). 称程序语句s_1处的某个Assume条件$\langle \Omega_1,\Phi_1,\varepsilon_1\rangle$依赖于程序语句$s_2$处的某个Assume条件$\langle \Omega_2,\Phi_2,\varepsilon_2\rangle$(记为$\langle \Omega_1,\Phi_1,\varepsilon_1\rangle \rightarrow \langle \Omega_2,\Phi_2,\varepsilon_2\rangle$),是指$\langle \Omega_1,\Phi_1,\varepsilon_1\rangle$的可靠性依赖于$\langle \Omega_2,\Phi_2,\varepsilon_2\rangle$的可靠性.

换言之,如果$\langle \Omega_2,\Phi_2,\varepsilon_2\rangle$是$s_2$处的一个可靠的Assume条件,则$\langle \Omega_1,\Phi_1,\varepsilon_1\rangle$也是$s_1$处的一个可靠的Assume条件.Assume条件之间的依赖关系是传递的,我们定义 $psp \xrightarrow{*} psp'$ 当且仅当存在$psp_1,\ldots,psp_n$,使得

$$psp \rightarrow psp_1 \rightarrow \ldots \rightarrow psp_n \rightarrow psp'.$$

Assume-Guarantee 搜索复用方法的基本思想是,使用某个特定的初始抽象准则(如性质自动机中出现的变量集合),从程序的入口语句遍历程序的执行路径,将到达每个程序点的部分最强后置条件作为该点处的 Assume 条件,并在后续的遍历过程中验证这些 Assume 条件的可靠性.如果在遍历时找到一条执行路径不满足待验证的性质,就首先分析该路径是否是一条真实可行(feasible)的执行路径,如果是,则找到了一条反例路径;否则是由于某个 Assume 条件不可靠而造成的伪反例(spurious counterexample)路径.造成 Assume 条件不可靠的原因是由于变量抽象的抽象准则太粗糙,因此我们推断出必要的程序变量加入抽象准则变量集合,并对不可靠的 Assume 条件进行精化,直到证明了所有的 Assume 条件都是可靠的.

2.2 基于Assume条件的执行路径遍历

基于Assume-Guarantee搜索复用方法的验证过程如图 1 所示,输入参数是程序控制流图$CFG=\langle S,E\rangle$和性质自动机FSA,其基本方法是基于控制流图按深度优先方式遍历程序的所有执行路径.过程基于一个先进后出的栈*WorkStack*进行,栈中的每个元素都是形如π的结构,该结构中包含 3 个元素:$s\in S$是当前的程序语句;$succ\subseteq S$是s的后继语句;$psp\in PSP$是一个形如$\langle \Omega,\Phi,\varepsilon\rangle$的Assume条件.栈*WorkStack*定义了 4 个常规操作:push,pop,top和empty,分别表示数据进栈、数据出栈、取得栈顶数据和判断栈是否为空.*AssumeMap*记录了程序的每条语句所对应的Assume条件,初始为空(过程第 13 行).V是作为抽象准则的变量集合,初始时,V为一个初始变量集合.过程第 14 行、第 15 行往栈中放置了一个初始元素,其中,s_0为程序的入口语句,函数$Succ(s_0)$返回s_0中控制流图中的所有后继语句,$\langle \Omega_0,\Phi_0,\varepsilon_0\rangle$表示初始条件True,其中,$\Omega_0=\Phi_0=\varnothing$,$\varepsilon_0$为性质状态机的初始态.随后,过程进入循环,如果栈不为空(第 16 行),就得到栈顶元素(第 17 行),然后判断栈顶元素的Assume条件$\langle \Omega,\Phi,\varepsilon\rangle=\pi.psp$中性质状态机的状态$\varepsilon$是否是性质状态机的接收态(第 18 行、第 19 行),如果是,则表明性质被违背,栈中存储的是一条可能的反例路径,我们需要检查其可行性,并根据不可行的伪反例路径精化抽象准则,其详细讨论将在第 2.3 节给出,本节中我们只考虑性质不被违背的情况.如果性质没有被违背,则首先判断当前语句是否还有未被遍历的后继语句(第 29 行),如果有,则通过第 31 行调用ForwardSearch对其进行处理;否则在第 32 行调用BackTrack回溯到栈中的前一条语句.

图 1 所示过程在执行时还需要动态构建Assume条件之间的依赖关系"→"(第 5 行、第 12 行),如果某个Assume条件被证明为不可靠,那么依赖于它的所有Assume条件也不可靠,需要将它们从*AssumeMap*中去除(第

26 行、第 27 行).先看ForwardSearch函数,它根据当前语句的参数结构π构建其后继语句s'的参数结构π',并在必要时将其推入栈中.过程第 1 行设置π'结构的s和$Succ$两个成员,第 2 行根据π计算出语句s'的Assume条件$\pi'.psp$,设$\pi'.psp\rightarrow\langle\Omega,\Phi,\varepsilon\rangle$.注意,性质状态机的状态$\varepsilon$也被更新.过程第 3 行判断$s'$新的Assume条件$\pi'.psp$相对于$s'$已有的*Assume*条件$AssumeMap(s')$是否是冗余的,如果是,则根据冗余*Assume*条件的定义,我们不需要基于它继续遍历下去,因此在这种情况下,不会将s'的参数结构π'推入栈中.但这么做的前提条件是$AssumeMap(s')$中的所有*Assume*条件都是可靠的,因此,过程第 4 行、第 5 行建立$\pi'.psp$到$\pi'.psp$所蕴含的$AssumeMap(s')$的子集$\Rightarrow_{\pi'.psp}(AssumeMap(s'))$中每个元素$\langle\Omega,\Phi,\varepsilon\rangle$之间的依赖关系.如果$\pi'.psp$相对于$AssumeMap(s')$不是冗余的,过程在第 7 行将$\pi'.psp$作为$s'$的一个新的*Assume*条件加入$AssumeMap(s')$,并在第 8 行将$\pi'$推入栈中,以便进一步遍历.如果当前语句的所有后继都被遍历完成(第 32 行的else分支),则需要调用BackTrack回溯.回溯时,先在第 9 行将当前语句的参数结构π弹出堆栈,再取得当前语句的前驱语句的参数结构π'(第 10 行),并对$\pi.psp$所依赖的所有Assume条件$\langle\Omega,\Phi,\varepsilon\rangle$,设置$\pi'.psp\rightarrow\langle\Omega,\Phi,\varepsilon\rangle$.换言之,当一个语句的Assume条件从栈中弹出时,它应该将它的依赖关系传递给它的前驱.

```
/* Global variables */
π,π': struct{s∈S; succ⊆S; psp∈PSP;};
WorkStack: A list of struct π;
AssumeMap:  S ↦ 2^PSP ;
V: Abstraction criterion (A set of variables);

ForwardSearch(π,s'){
1    let π'.s=s', π'.succ=Succ(s');
2    let π'.psp=SP_V(π.s)(π.psp);
3    if (π'.psp⇒AssumeMap(s')){
4      For all ⟨Ω,Φ,ε⟩∈(⇒_{π'.psp}(AssumeMap(s')))
5        set π'.psp→⟨Ω,Φ,ε⟩;
6    } else {
7      AssumeMap(s')=AssumeMap(s')∪{π'.psp};
8      WorkStack.push(π');
     }
}
BackTrack(π){
9    WorkStack.pop();
10   let π'=WorkStack.top();
11   For all ⟨Ω,Φ⟩ such that π.psp→⟨Ω,Φ⟩
12     set π'.psp→⟨Ω,Φ⟩;
}
```

```
main(CFG=⟨S,E⟩,FSA){
13   or all s∈S set AssumeMap(s)=∅;
  RESTART:
14   et π.s=s0, π.succ=Succ(s0),π.psp=⟨Ω0,Φ0,ε0⟩;
15   WorkStack.push(π);
16   while (!WorkStack.empty()){
17     let π=WorkStack.top();
18     let ⟨Ω,Φ,ε⟩=π.psp;
19     if (ε is an accepting state of FSA){
20       let p=π0.s,…,πn.s is the path in WorkStack;
21       if (p is feasible)
22         Report p as a counter-example and EXIT;
23       else{
24         Refine the abstraction criterion V;
25         For all π'.psp∈WorkStack
26           For all ⟨Ω',Φ',ε'⟩ →* π'.psp
27             Remove ⟨Ω',Φ',ε'⟩ from AssumeMap;
28         Clear WorkStack and goto RESTART;
         }
29     } else if (∃s'∈π.succ){
30       Remove s' from π.succ;
31       ForwardSearch(π,s');
32     } else BackTrack(π);
     }
}
```

Fig.1 Verification procedure based on Assume-Guarantee framework

图 1 基于 Assume-Guarantee 框架的验证过程描述

定理 1. 如果图 1 所示过程弹出了某个栈元素π(过程第 9 行),设 Assume 条件$\pi.psp$ 所依赖("$\rightarrow$")的所有 Assume 条件都是可靠的,那么$\pi.psp$ 也是语句$\pi.s$ 的一个可靠的 *Assume* 条件.

证明:图 1 所示过程会遍历一条语句的所有后继语句,除非某条后继语句s'的Assume条件$\langle\Omega,\Phi,\varepsilon\rangle$相对于$AssumeMap(s')$是冗余的,此时,过程会建立$\langle\Omega,\Phi,\varepsilon\rangle$到$\Rightarrow_{\pi'.psp}(AssumeMap(s'))$中所有Assume条件的依赖关系,并在回溯时将这些依赖关系传递到其所有前驱.由定理的条件我们知道,$\Rightarrow_{\pi'.psp}(AssumeMap(s'))$中的所有的Assume条件都是可靠的,且被$\langle\Omega,\Phi,\varepsilon\rangle$所蕴含,因此,过程虽然不遍历$s'$,但可以保证以$\langle\Omega,\Phi,\varepsilon\rangle$遍历$s'$也不会违背所给性质.$\pi$从栈中被弹出,意味着$\pi.s$的所有后继都被遍历完成或不需要遍历,由于$\pi.s$的所有后继也都已经从栈弹出了,我们可以归纳得到,从$\pi.s$出发、变量取值满足$\pi.psp$的所有程序执行路径都已被遍历完成,这些执行路径都不违背所给性质.另外,作为Assume条件的部分最强后置条件是对程序语义的保守近似,基于其遍历的程序执行路径

中包含了所有实际的程序执行路径,从而定理得证. □

推论 1. 如果图 1 所示过程因栈 *WorkStack* 为空而退出(过程第 16 行),且执行过程中未出现性质自动机被违背的情况(即过程第 19 行的 if 条件不为真),那么被验证的性质得到满足.

证明:如果过程退出时栈为空,就意味着所有语句s对应的*AssumeMap*(s)中的所有Assume条件都是可靠的.特别地,程序入口语句对应的Assume条件True($\Omega_0=\Phi_0=\varnothing$)也是可靠的.也就是说,无论程序输入是什么,都不会违背性质,故推论得证. □

2.3 反例检查与抽象准则精化

由于部分最强后置条件是对程序语义的保守近似,基于部分最强后置条件除了会遍历所有实际的程序执行路径之外,还会遍历一些在程序实际执行中不会存在的路径.因此,对于一条基于部分最强后置条件产生的反例路径,我们必须检查其是否是真实可行的(feasible).如果反例路径不可行(infeasible),我们需要根据它精化抽象准则,这些工作对应于图 1 过程中的第 20~第 27 行.

对于栈*WorkStack*中的一条可能的反例路径$p=\pi_0.s,\ldots,\pi_n.s$,我们通过计算其传统的最强后置条件来判断其是否可行.令$\langle\Omega,\Phi\rangle=SP(p)$(True),如果$h(\langle\Omega,\Phi\rangle)$=False,则表明路径$p$是不可行路径.在实际中,常使用如下与$h(\langle\Omega,\Phi\rangle)$=False等价的公式:

$$\bigwedge_{\phi\in\Phi}\phi=\text{False} \tag{8}$$

我们也可以利用基于测试的方法,例如,基于指定路径的程序测试数据生成技术,来判断p是否可行[16,17].一条路径p不可行,是由它的assume语句之间的冲突造成的,如果在计算路径p的部分最强后置条件时,由于抽象准则太粗而将这些冲突的assume语句抽象为无关语句,就会造成根据部分最强后置条件判断出p可行的错误.因此,我们需要找出这些冲突的assume语句,然后使用它们作为切片请求(slicing request),对路径p进行动态切片[18],找出所有依赖语句,并将这些语句中所包含的变量加入到抽象准则变量集合V中去,这样,在使用精化后的抽象准则计算p的部分最强后置条件时,就可以判断出p为不可行路径.

给定一条执行路径p和一个变量集合V作为切片请求[18],动态程序切片能够计算所有相关的语句和变量.当前的动态切片算法(如文献[18])具有高度的可扩展性,并能很好地支持指针和变量别名.令$\langle\Omega,\Phi\rangle=SP(p)$(True),为了进行过程第 24 行的抽象准则精化,我们先找出Φ的满足式(8)的极小子集$\Phi_{\min}$,即$\Phi_{\min}$是导致p不可行的assume语句的极小集合.我们再以$Vars(\Phi_{\min})$为动态切片请求计算出相关的语句集合$DS(Vars(\Phi_{\min}))$,然后再将抽象准则由V精化为$V\cup Vars(DS(Vars(\Phi_{\min})))$.

定理 2. 对于一条不可行的路径p,如果$SP_V(p)$(True)≠False,设根据上述方法将抽象准则由V精化为$V\cup V'$,其中$V'=Vars(DS(Vars(\Phi_{\min})))$,那么一定有$SP_{V\cup V'}(p)$(True)=False.

证明:基于精化后的抽象准则$V\cup V'$对路径p进行的变量抽象,将把$DS(Vars(\Phi_{\min}))$中的所有语句抽象为相关语句,由这些语句的最强后置条件可以判断出p是不可行的. □

定理 2 表明,如果基于某抽象准则 V 的部分最强后置条件判断出某不可行路径 p 是可行的,那么,基于精化抽象准则 $V\cup V'$的部分最强后置条件一定可以判断出 p 不可行.

经过图 1 中第 24 行的抽象准则精化后,我们保守地假设栈中所有元素π'的 Assume 条件$\pi'.psp$ 都由于抽象准则过于粗糙而不可靠,同时,所有依赖于$\pi'.psp$ 的 Assume 条件$\langle\Omega',\Phi',\varepsilon'\rangle\xrightarrow{*}\pi'.psp$ 也都不可靠.过程第 25~第 27 行将所有不可靠的 Assume 条件全部从 *AssumeMap* 中去掉.对不依赖于堆栈中所有$\pi'.psp$ 的其他 Assume 条件,根据定理 1 可知它们仍是可靠的.随后,过程在第 28 行清空堆栈,并跳转到 *RESTART* 重新开始对程序执行路径的遍历.在新的遍历过程中,由于可以复用那些仍然可靠的 Assume 条件,从而可以有效减少要遍历的执行路径的数量,降低遍历代价.在遍历程序的执行路径时,由于程序中循环的存在,图 1 给出的验证过程可能不终止.我们可以借鉴 BLAST 和 MAGIC 的方法,最多只将循环展开一定的次数以确保终止性.需要指出的是,本文方法能够很好地处理实用程序中的大部分循环,例如,在实验程序中我们并未引入循环展开次数限制,但对所有实验程序的验证均能够终止.

3 验证工具与实验结果

我们在MAGIC[5]的基础上实现了基于Assume-Guarantee搜索复用方法的C程序验证工具,工具使用Simplify[19]作为定理证明工具,使用Das的流不敏感的指向图算法[20]来处理指针和变量别名.该工具支持对函数指针、变量别名、结构和数组的处理,其局限是不支持递归函数调用和非直接跳转如setjump/longjump等,这些限制目前在实验程序中尚不存在.

我们通过使用与BLAST,MAGIC和切片执行相同的测试用例来检验本文所提出的Assume-Guarantee验证方法的有效性和实用性.所有的测试用例都取自于openssl-0.9.6c的程序源代码,它共有 2 000 多行C代码,实现了在Internet上进行安全信息传输的SSL协议.与上述验证工具一样,我们的验证目标是SSL协议的初始握手协议.客户端和服务器端实现握手协议的代码都是大约 350 行,代码结构为如图 2 所示的状态机形式,其switch语句共有大约 35 个case语句,用于根据当前状态决定所做的动作以及决定其后续状态.待验证的性质是某些特定的状态转换序列不会在握手过程中出现,例如$S_3 \rightarrow S_5 \rightarrow S_{33} \rightarrow S_1$.我们共验证了 4 个客户端性质和 16 个服务器端性质,分别使用*ssl-clnt*和*ssl-srvr*来标识它们,待验证性质状态机的状态数量为 6 个~22 个.在性质状态机的每个状态,需要遍历 35 个case语句,以选择合适的分支并执行该分支对应的代码.如果性质状态机的状态数量为 6,就相当于要把上述约 350 行代码组合 6 次,因此,测试用例对应的验证任务具有相当的验证规模,BLAST, MAGIC和我们的实验结果也证明了这一点.与其他验证工具的结果一样,所有性质都被程序满足.

我们的实验平台是 1.6GHz AMD Athlon XP 的 CPU 和 1GB 内存,软件环境是 Windows 2000 和 CygWin 2.427.实验结果见表 1,表中 Properties 为待验证的性质,Name 为性质的名字,States 为性质状态机的状态数.Assume-Guarantee reuse of searching 为本文给出的验证过程,CE Paths 表示在验证过程中找到的伪反例路径(spurious counter-example paths)的数量,Vars 为变量抽象的最终抽象准则中变量的数量,Thm Calls 为验证过程中调用定理证明工具的次数,Time 为验证所用的时间,单位是秒.表 1 中也给出了 BLAST,MAGIC 和切片执行的验证结果,其中,BLAST 和 MAGIC 的数据来自文献[21],其实验平台是 1.6GHz AMD Athlon XP 的 CPU 和 900MB 内存,软件环境是 Linux.表中"*"表示验证时间超过 3 个小时,最好的验证结果在表中表示为粗体.需要说明的是,虽然我们的实验平台比文献[21]的实验平台多了 100MB 的内存,但除了两个性质 srvr-6 和 srvr-14 外,其他性质的验证都只使用了不到 200MB 内存.

Table 1 Experimental results

表 1 实验结果

Properties		Assume-Guarantee reuse of searching				BLAST	MAGIC	Slicing execution	
Name	States	CE paths	Vars	Thm calls	Time	Time	Time	Thm calls	Time
ssl-clnt-1	6	8	9	4 362	**12**	348	156	32 518	28
ssl-clnt-2	6	5	8	877	**8**	523	185	8 632	10
ssl-clnt-3	6	6	8	1 133	**10**	469	195	45 327	39
ssl-clnt-4	6	8	9	4 365	**12**	380	191	37 966	34
ssl-srvr-1	6	5	8	5 020	**24**	2 398	226	111 495	85
ssl-srvr-2	5	6	9	5 821	**65**	691	216	84 791	66
ssl-srvr-3	5	7	10	267 314	439	1 162	200	86 018	**67**
ssl-srvr-4	5	6	9	9 448	**35**	284	170	87 000	68
ssl-srvr-5	9	14	17	111 077	405	1 804	205	211 516	**145**
ssl-srvr-6	22	6	9	1 424 958	2552	*	**359**	894 003	698
ssl-srvr-7	9	5	8	8 868	**76**	359	196	221 230	150
ssl-srvr-8	11	5	8	5 584	**25**	*	211	279 124	207
ssl-srvr-9	9	5	8	11 164	**80**	337	316	213 075	145
ssl-srvr-10	8	7	10	265 409	444	8 289	241	169 836	**127**
ssl-srvr-11	9	5	8	8 899	**76**	547	356	211 821	145
ssl-srvr-12	13	13	17	99 695	374	2 434	301	385 410	**280**
ssl-srvr-13	9	4	7	2 209	**135**	608	436	210 169	144
ssl-srvr-14	16	6	9	1 422 838	2529	10 444	406	531 295	**405**
ssl-srvr-15	14	7	10	273 767	476	*	**179**	409 549	305
ssl-srvr-16	19	13	17	102 904	397	*	**356**	700 580	536

从表 1 的实验结果我们可以看到,在 20 个验证用例中有 12 个都是Assume-Guarantee搜索复用方法的验证

时间最短,而且某些验证用例(如srvr-1,srvr-4,srvr-7,srvr-8,srvr-9,srvr-11 等)的验证效果要比其他 3 种工具或方法好很多,这说明了本文所提出方法的有效性和实用性.但同时我们也注意到,某些验证用例,如srvr-6 和srvr-14 的验证时间大大长于MAGIC和切片执行.经过仔细分析我们发现,造成这种现象的原因有 3 个,如何解决这些问题也是我们后续的研究工作:一是由于Assume-Guarantee搜索复用方法是基于栈实现的,如果过程一开始就选择了一个不会出现错误的分支,那么它就必须等该分支的所有后继被遍历完成后才会选择其他分支;二是使一条伪反例路径不可行的变量可能有很多组,我们为精化抽象准则而选择使用的某一组变量未必是最佳选择;三是由于某些变量未被及时加入变量抽象的抽象准则而导致其取值范围被放大,从而增加大量额外的程序执行路径.例如,ssl_srvr程序中有如图 3 所示的代码,上述两个验证时间过长的性质都是由于没有及时将变量($s\rightarrow s_3$)→tmp.next_state加入抽象准则,而导致$s\rightarrow$state的取值范围在执行case S_m后被放大到可以取任意值,从而在下一次循环中$s\rightarrow$state被认为可能进入所有的case.

```
while (1){
    switch(s→state){
        case S1: …; s→state=NEXT(S1);
break;
        case S2: …; s→state=NEXT(S2);
break;
        …
        case Sn: …; s→state=NEXT(Sn);
```

Fig.2 The code structure of SSL Handshake routine

图 2 SSL 初始握手协议的代码结构

```
while (1){
  switch(s→state){
    case Sm: …; s→state=(s→s3)→tmp.next_state; break;
    …
    case Sn: …; (s→s3)→tmp.next_state=8576; break;
  }
}
```

Fig.3 A code fragment of *ssl_srvr* routine

图 3 *ssl_srvr* 例程中的一段代码

4 相关工作

Assume-Guarantee推理的基本思想已被VeriSoft[9],ComFoRT[7]等基于源代码的验证工具用来对构件化和并发程序的验证.文献[10]论述了如何基于Assume-Guarantee推理实现对既包含串行模块又包含并发进程的程序进行验证.Assume-Guarantee推理除了用于验证的目的以外,还可以用于程序测试,文献[22]给出了这方面的研究.本文提出的基于Assume-Guarantee的验证方法在思路上与上述研究有相通之处,不同之处在于,本文的思路是针对顺序程序搜索中模块化复用,其假设的粒度更细,是在程序语句级.从另一方面来看,本文提出的方法与模块或进程级的Assume-Guarantee推理工作在程序的不同级别,可以综合运用.

本文提出的方法可用于对并发 C 程序进行验证,不同之处在于,并发 C 程序的每个"并发程序位置"是由所有并发进程的一条程序语句组合而成的.Assume-Guarantee 搜索复用方法为每个"并发程序位置"引入一个假设条件,并以深度优先的方式遍历从该位置出发的所有交迭执行路径,以验证假设条件的可靠性,参见文献[23].

变量抽象与部分最强后置条件的概念最初是在文献[13]中作为切片执行的理论基础提出来的,切片执行可看作是一种轻量级的符号执行,它首先基于某个抽象准则,使用变量抽象将程序语句划分为相关和无关语句,然后计算相关语句的部分最强后置条件,并基于部分最强后置条件对程序路径进行保守的遍历,并同时验证性质是否得到满足.基于切片执行的 C 程序验证方法也是一个迭代的过程,每次迭代过程中产生的伪反例路径被用来精化抽象准则,直到性质被验证或找到一条真实的反例路径为止.该方法的每一次迭代过程中的计算都是按不动点的方式进行的,即从待计算的语句集合中任取一个,计算其部分最强后置条件后将其后继语句加入待计算语句集合,直到达到一个不动点为止.由于该过程可以有选择地遍历最有可能出错的语句分支,因此它不会像 Assume-Guarantee 验证方法那样,由于选择错误的语句分支而显著增加验证时间(见表 1 的实验结果).但同时,由于一个新的迭代无法使用之前迭代的信息,一般来说,其验证代价比本文的验证方法要大.

给定一组谓词,谓词抽象[1]可以自动地从程序源代码中抽象出程序的有限自动机模型,它是SLAM[2,3],BLAST[4],MAGIC[5,6]及ComFoRT[7,8]等工具和项目的理论核心.这些工具的验证过程与基于切片执行的方法一样也是迭代进行的,每次迭代分 3 个步骤,即谓词抽象-模型检验-反例检查与反例指导的抽象精化(CEGAR),直到性质满足或找到一条真实反例为止.本文提出的Assume-Guarantee验证方法沿用了反例指导的抽象精化的思

路,通过不断地根据伪反例路径精化和加强不可靠的假设条件,充分利用已有的验证信息,从而有效地降低了验证代价.

其他面向C程序源代码的验证方法包括:静态分析方法,如Meta-level Compilation[24],ESP[25]及MOPS[26]等,其优点是可扩展性很强,但验证过程往往存在误报的问题;抽象解释方法如ASTREE[27]及C Global Surveyor(CGS)[28]等,当前它们只能验证C程序的运行时错误,而不支持一般的时序安全性质;动态模型检验方法,如CMC[29]及VeriSoft[9]等,其优点是无须抽象,保证了精确性,但往往不能遍历程序的所有执行路径,因此常用于查错而非验证.与上述方法相比,本文的方法能够面向复杂的时序安全性质,在保证精确验证的前提下提高可扩展性.

5 结束语

本文提出了一种基于 Assume-Guarantee 搜索复用的验证方法,用于对 C 程序源代码的验证,其基本思想是,在程序的每条语句处引入保守的假设条件,再在后续的验证过程中验证每个假设条件的可靠性,如果某个假设条件不可靠,就基于产生的反例路径对其进行精化和加强,直到性质得到证明或找到一条真实的反例路径为止.每个假设条件都是根据程序的保守近似语义得出的对程序行为较弱的保守假设,从而能够大量地缩减需要遍历的程序执行路径数量,有效降低了验证开销.本文的方法用于验证 Linux 操作系统下 SSL 协议的实现程序 openssl-0.9.6c 满足 SSL 协议的初始握手规范,实验结果表明,该方法具有良好的实用性和可扩展性.

References:

[1] Graf S, Saidi H. Construction of abstract state graphs with PVS. In: Grumberg O, ed. Proc. of the Computer-aided Verification (CAV'97). LNCS 1254, Haifa: Springer-Verlag, 1997. 72−83.

[2] Ball T, Majumdar R, Millstein T, Rajamani SK. Automatic predicate abstraction of C programs. In: Proc. of the Programming Language Design and Implementation (PLDI 2001). Snowbird: ACM Press, 2001. 203−213. http://portal.acm.org/citation.cfm?id=381694.378846

[3] Ball T, Rajamani SK. Generating abstract explanations of spurious counterexamples in C programs. Technical Report, MSR-TR-2002-09, Microsoft Research, Microsoft Corporation, 2002. http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2002-09

[4] Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with BLAST. In: Proc. of the 10th Int'l SPIN Workshop (SPIN 2003), LNCS 2648, Portland: Springer-Verlag, 2003. 235−239. http://www-cad.eecs.berkeley.edu/tah/Publications/software_verification_with_blast.pdf

[5] Chaki S, Clarke E, Groce A. Modular verification of software components in C. In: Clarke LA, Dillon LK, Tichy W, eds. Proc. of the 25th Int'l Conf. on Software Engineering (ICSE 2003). Portland: IEEE Computer Society. 2003. 385−395.

[6] Chaki S, Ouaknine J, Yorav K, Clarke E. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In: Proc. of the 2nd Workshop on Software Model Checking (SoftMC). 2003. http://www.cs.cmu.edu/~chaki/publications/SOFTMC-2003.pdf

[7] Chaki S, Clarke E, Sinha N, Thati P. Automated assume-guarantee reasoning for simulation conformance. In: Etessami K, Rajamani SK, eds. Proc. of the Computer Aided Verification (CAV). LNCS 3576, Edinburgh: Springer-Verlag, 2005. 534−547.

[8] Chaki S, Ivers J, Sharygina N, Wallnau K. The ComFoRT reasoning framework. In: Etessami K, Rajamani SK, eds. Proc. of the Computer Aided Verification (CAV). LNCS 3576, Edinburgh: Springer-Verlag, 2005. 164−169.

[9] Dingel J. Computer-Assisted assume/guarantee reasoning with VeriSoft. In: Clarke LA, Dillon LK, Tichy W, eds. Proc. of the 25th Int'l Conf. on Software Engineering (ICSE 2003). Portland: IEEE Computer Society, 2003. 138−148.

[10] Henzinger TA, Minea M, Prabhu V. Assume-Guarantee reasoning for hierarchical hybrid systems. In: Benedetto MDD, Sangiovanni-Vincentelli AL, eds. Proc. of the Hybrid Systems: Computation and Control 4th Int'l Workshop HSCC 2001. Springer-Verlag, 2001. 275−290.

[11] Weiser M. Program slicing. IEEE Trans. on Software Engineering (TSE), 1982,SE-10(4):352−357.

[12] Yi XD, Wang J, Yang XJ. Slicing execution for model checking C programs. Int'l Journal of Software Engineering and Knowledge Engineering (IJSEKE), 2006,16(5):747−768.

[13] Yi XD, Wang J, Yang XJ. Verification of C programs using slicing execution. In: Proc. of the 5th Int'l Conf. on Quality Software (QSIC 2005). Melbourne: IEEE Computer Society Press, 2005. 109−116. http://ieeexplore.ieee.org/iel5/10545/33358/01579126.pdf?tp=&isnumber=&arnumber=1579126

[14] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-Guided abstraction refinement. In: Emerson EA, Sistla AP, eds. Proc. of the CAV. LNCS 1855, Springer-Verlag, 2000. 154−169.

[15] Gries D. The Science of Programming. Springer-Verlag, 1981.

[16] Zhang J, Wang XX, A constraint solver and its application to path feasibility analysis. Int'l Journal of Software Engineering & Knowledge Engineering, 2001,11(2):139−156.

[17] Shan JH, Wang J, Qi ZC. Improved method to generate path-wise test data. Journal of Computer Science and Technology, 2003, 18(2):235−240.

[18] Zhang XY, Gupta R, Zhang YT. Precise dynamic slicing algorithms. In: Clarke LA, Dillon LK, Tichy W, eds. Proc. of the IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Portlandz: IEEE Computer Society, 2003. 319−329.

[19] Detlefs D, Nelson G, Saxe J. Simplify: A theorem prover for program checking. 2003. http://research.compaq.com/src/esc/simplify.html

[20] Das M. Unification-Based pointer analysis with directional assignments. In: Lam M, ed. Proc. of the PLDI 2000: Programming Language Design and Implementation. ACM Press, 2000. 35−46.

[21] Chaki S, Clarke E, Groce A, Strichman O. Predicate abstraction with minimum predicates. In: Proc. of the 12th Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME). LNCS 2860, Springer-Verlag. 2003. 19−34

[22] Blundell C, Giannakopoulou D, Pasareanu CS. Assume-Guarantee testing. In: Proc. of the SAVCBS 2005. 2005. 7−14 http://www.cs.iastate.edu/~leavens/SAVCBS/2005/SAVCBS05.pdf

[23] Wang J, Yi XD, Yang XJ. Towards a framework for scalable model checking of concurrent C programs. In: Proc. of the IsoLA 2006. 2006.

[24] Ashcraft K, Engler DR. Using Programmer-Written Compiler Extensions to Catch Security Holes. IEEE Security and Privacy, 2002. 143−159.

[25] Das M, Lerner S, Seigle M. ESP: Path-Sensitive program verification in polynomial time. In: Proc. of the PLDI 2002: Programming Language Design and Implementation. Berlin: ACM Press, 2002. http://www.cs.cornell.edu/courses/cs711/2005fa/papers/dls-pldi02.pdf

[26] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In: Proc. of the 9th ACM Conf. on Computer and Communications Security (CCS). 2002. 235−244.

[27] Cousot P, Cousot R, Feret J, Mauborgne L, Mine A, Monniaux D, Rival X. The ASTREE analyzer. In: Proc. of the ESOP 2005: The European Symp. on Programming. LNCS 3444, Springer-Verlag, 2005. 21−30.

[28] Venet A, Brat G. C global surveyor: Designing a static analyzer for NASA flight software. In: Proc. of the VMCAI 2005. LNCS 3385, Paris: Springer-Verlag, 2005.

[29] Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. In: Proc. of the 5th Symp. on Operating System Design and Implementation (OSDI). 2002. 75−88. http://www.stanford.edu/~engler/osdi2002.pdf

易晓东(1978－),男,湖北松滋人,博士生,主要研究领域为模型检验,高可信软件技术,软件工程.

王戟(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高可信软件技术,软件方法学,软件工程.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行与分布处理,信息安全,软件工程.