软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(10):2655−2664 [doi: 10.3724/SP.J.1001.2012.04196]

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

基于 SMT 求解器的路径敏感程序验证*

何炎祥[1,2], 吴 伟[2+], 陈 勇[2], 徐 超[2]

[1](武汉大学 计算机学院,湖北 武汉 430072)

[2](软件工程国家重点实验室(武汉大学),湖北 武汉 430072)

Path Sensitive Program Verification Based on SMT Solvers

HE Yan-Xiang[1,2], WU Wei[2+], CHEN Yong[2], XU Chao[2]

[1](School of Computer, Wuhan University, Wuhan 430072, China)

[2](State Key Laboratory of Software Engineering (Wuhan University), Wuhan 430072, China)

+ Corresponding author: E-mail: whuwuwei@126.com

He YX, Wu W, Chen Y, Xu C. Path sensitive program verification based on SMT solvers. *Journal of Software*, 2012,23(10):2655−2664 (in Chinese). http://www.jos.org.cn/1000-9825/4196.htm

Abstract: With the rapid increase in size and complexity of software, more and more attention is paid to the software's trust. Verifying whether programs satisfy the properties described by assertions is a common method to guarantee trust of the software. Since path sensitive program verification cannot traverse all paths, it needs merge the path information, which causes a loss of precision. The study proposes a program verification method using SMT solvers, which can reduce the path search space and improve the precision at the same time. The method's main sprit is impacting the cycle path by using maximal strongly connected component and slicing the CFG according to the aim assertion. The study abstracts the path space using Boolean formulas and verifies the path by combining abstract interpretation and symbolic execution. The study has conducted experiments based on the F-Soft program verification platform and SMT solver Z3, and results show that this method performs well based on precision and effect.

Key words: path sensitive; program verification; abstract interpretation; symbolic execution; SMT solvers

摘 要: 随着软件规模的不断扩大以及复杂度的不断增长,人们越来越关注软件的可信性问题.验证程序是否满足断言所描述的性质,是保证软件可信性的一种常见方法.路径敏感的程序验证由于不可能遍历所有的路径,需要合并路径信息,因此造成精度上的损失.提出一种基于 SMT 求解器的路径敏感程序验证方法,在保证精确度的前提下,有效减少路径搜索空间.其基本思想是,利用最大强连通分量压缩循环路径,然后根据目标断言对控制流图进行切片.使用一种布尔表达式方法对路径空间进行抽象,结合抽象解释和符号执行技术对路径进行验证.结合 F-Soft 平台和 Z3 工具对该方法进行了实验验证,结果表明,该方法在验证的精确度和效率上都有较好的效果.

关键词: 路径敏感;程序验证;抽象解释;符号执行;SMT 求解器

中图法分类号: TP311 **文献标识码**: A

* 基金项目: 国家自然科学基金(90818018, 91018009)
收稿时间: 2011-08-03; 修改时间: 2011-11-02; 定稿时间: 2012-02-15

软件已经应用到社会的各个领域,软件的不可信会给人们的生命财产安全带来重大隐患[1].另外,即使最有经验的程序员,在编程的过程中也会出现错误.尽管有语法和类型检查工具,但是仍然消除不了语义层次的缺陷.对程序进行验证,是保证软件可信的重要手段,主要方法有模型检测、定理证明等.基于霍尔逻辑的程序验证方法由于依靠可靠的数学推理,受到越来越多人的重视[2].但是由于定理证明器的能力限制、循环不变式难以求解以及程序指针等数据结构的复杂性等原因,导致这种方法大部分需要人工完成,因此对程序的高效且精确的验证仍然有待改进.近几年来,SMT 求解器(SMT solver)[3]的快速发展,给大规模程序更加深层次性质的验证带来了新的机遇.SMT 求解器是对 SAT 求解器的扩展,SAT 求解器是判断布尔可满足性问题的工具,判定对象属于命题逻辑范畴.而命题逻辑的表达能力相对较弱,一阶逻辑在其基础上补充了量词和变量,与之对应的可满足性问题称为 SMT 问题(satisfiability modulo theory,简称 SMT).SMT 求解器就是求解 SMT 问题的自动化工具,它处理的对象是包含一些特定理论的一阶逻辑公式,典型的理论包括固定长度的位向量、数组、未解释函数、线性算术运算、差分逻辑等.SMT 求解器已成为许多程序分析和验证工具的基础组件.本文利用 SMT 求解器对路径进行编码和遍历.

路径敏感通常是验证程序安全性质的关键要求[4],因为对程序的每个分支进行单独验证,可以减少误报的数量,提高验证的准确性.然而,如果要准确遍历程序控制流图(CFG)[5]中的每条路径,会带来搜索空间爆炸问题,限制了验证复杂程序的能力.因此,如何减少搜索空间而又不影响验证的准确性,是路径敏感方法要解决的关键问题.本文提出一种利用 SMT 求解器的路径敏感程序验证方法来解决这个问题.首先,考虑到循环会造成路径空间的无限性,利用寻求最大强连通分量的方法对控制流图中的循环路径进行压缩;其次,在验证程序某个给定的性质时,通常程序中许多路径与性质无关,尽管它们影响程序的执行状态.我们利用切片技术[6,7]去掉控制流图中的无关边;最后,我们利用一种布尔表示式的方法对路径集合进行抽象以提高路径遍历的效率.对路径进行验证时,我们采用抽象解释[8]和符号执行[9]的方法获取路径的可达条件,然后利用 SMT 求解器来判定可达条件的满足性,以此来验证程序中断言的正确性.若程序中所有的断言都被验证通过,则可断定程序是安全的.

本文的主要贡献是:

(a) 结合多种方法,包括循环路径压缩、CFG 切片和路径编码,有效缓解了路径搜索空间爆炸问题,提高了程序精确验证的效率;

(b) 提出的方法充分利用了 SMT 求解器的优势,实现了对程序更加复杂性质的验证,提高了程序验证的自动化程度和准确度.

本文第 1 节利用一个例子对我们的方法进行一个总体的概述.第 2 节介绍路径空间缩减和路径编码方法.第 3 节介绍对路径的验证方法.第 4 节和第 5 节分别介绍实验结果和相关工作.最后对本文工作进行总结并对未来的工作进行展望.

1 基于 SMT 求解器的路径敏感程序验证方法概述

本文所提验证方法的主要算法流程如图 1 所示.我们首先通过一个例子来大致描述该算法的执行过程,算法中的几个关键步骤将在下面几节中详细描述.图 2 表示的是一个 C 语言函数示例程序,这个函数通过循环对变量 x,y 进行赋值.利用本文方法对这个程序进行验证的目的就是验证 12 行的断言成立.与 BLAST[10],SLAM[11] 等代码验证工具一样,本文的方法首先获得程序的控制流图 $\mathbb{P}_{cfg}$(算法第 1 行),控制流图类似于通常编译原理课本中所描述的,但是本方法采用的 CFG 有一点不同是:节点表示程序当前的执行状态[5],类似程序计数器的概念;边表示程序执行时的状态转移及转移条件或行为.另外,将程序中断言语句的命题取反,通过一条边连接到一个错误节点.示例程序的控制流图如图 3(a)所示.CFG 中存在着大量不影响目标断言验证的边,通过对 CFG 进行切片(算法第 2 行),取出无关的边,减少验证的工作量.接着对切片后的控制流图进行分析,找出图中最大强连通分量(算法第 3 行),将连通分量中所有节点合并成一个点形成新的压缩后的控制流图 $\mathbb{P}_{mscc}$.图 3(b)就是对图 3(a)所示控制流图进行压缩后的控制流图,从图中可以看出,节点 3~节点 5、节点 8、节点 11 合并成了一个新的

节点.通过压缩处理后,$\mathbb{P}_{mscc}$ 中就不存在循环路径了,减少了路径搜索空间.为了证明断言成立,实际上就是验证在任何初始状态下,没有到达错误节点的执行路径.我们使用布尔公式抽象的方式来编码起始节点到错误节点之间的所有路径集合Π_p(算法第 4 行),然后从集合中选取一条未被验证过的路径(算法第 6 行),验证这条路径是不可行路径.验证的过程中只考虑当前路径上的节点和边,可采用各种不同的验证方法(算法第 7 行).如果所选路径验证未通过,就将该路径加入到集合Π_r中,作为反例路径输出(算法第 8 行).从路径集合Π_p中除去当前路径,然后继续选取未被验证路径加以验证,直到所有路径都被验证过(算法第 10 行).

Algorithm 1. Verifying program based on SMT solvers.

Input: $\mathbb{P}$: program source, Ψ: $\langle l_n,\phi\rangle$ assertion to be verified;

Output: Π_r: set of path which is fail to be verified.

begin

1 $\mathbb{P}_{cfg}$:=*ConstructCFG*($\mathbb{P}$)

2 $\mathbb{P}_{sli}$:=*SliceCFG*($\mathbb{P}_{cfg}$)

3 $\mathbb{P}_{mscc}$:=*CompactCFG*($\mathbb{P}_{sli}$)

4 Π_p:=*EncodePaths*($\mathbb{P}_{mscc}$,l_0,l_n)

5 **while** (*SAT*(Π_p)) **do**

6 π :=*GetPath*(Π_p)

7 **if** (not *VerifyingPath*(π)) **then**

8 Π_r:=$\Pi_r\cup\{\pi\}$

9 **endif**

10 Π_p:=$\Pi_p\wedge$*ExcludeEncode*(π)

11 **endwhile**

end

Fig.1 Verifying algorithm

图 1 验证算法

```
0:  void example(·) {
1:    int x, y;
2:    x=20; y=0;
3:    while (x>0) {
4:      if (x>10) {
5:        x=x−1;
6:        y=y+1;
7:      } else {
8:        x=x−1;
9:        y=y−1;
10:     }
11:   }
12:   ASSERT(y=0);
13: }
```

Fig.2 Example program

图 2 示例程序

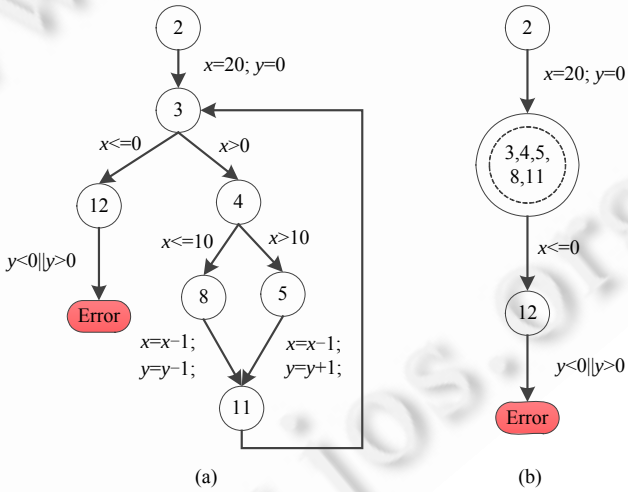

Fig.3 CFG of example program and its compact form

图 3 实例程序的 CFG 及其压缩形式

本文的验证方法是路径敏感的,具有较好的精确性.但是路径敏感方法通常遇到的问题就是路径空间太大,影响验证的效率和规模.示例程序的路径只有几条,但在实际中,一个控制流图通常有几百条边,路径的数量就急剧上升.因此,采用显示的方法遍历路径集合是不可行的.为了解决这个问题,我们使用了 3 种手段来提高验证效率:

- 首先,通过对 CFG 进行切片,除去大量无关路径;

- 其次,通过寻找最大强连通分量,我们压缩了切片后控制流图中的循环和递归节点,生成无环的控制流图,使得因循环导致的无限路径空间变成有限的路径空间;
- 最后,我们采用布尔逻辑对路径集合进行抽象编码.这种编码方式可以有效支持以下操作:
  1) 使用布尔公式对控制流图中任意两个节点间的所有路径进行编码,并且布尔公式的复杂度与控制流图的大小呈线性关系;
  2) 给定未遍历的路径集合抽象表达式Π_p与一个边集合 E,从Π_p中剔除所有包含 E 中边的路径.剔除是通过向Π_p中添加剔除布尔逻辑子句而实现的;
  3) 确定路径集合是否遍历完,即判定Π_p是否可满足,或者产生一条未遍历的路径,即求解Π_p.

我们采用 SMT 求解器来完成上述操作.近几十年来,SMT 求解器取得了巨大的进步,可以求解包含多达上千个变量和子句的公式.因此,对于大规模的程序验证,当前一些流行的求解器,如 Z3[12],MathSAT[13],Yices[14],Simplify[15],CVC3[16]等,可以有效支持我们的验证方法.与采用固定顺序搜索 CFG 的方法相比,如符号执行采用的深度优先搜索或模型检测采用的广度优先搜索,基于布尔表达式抽象的遍历方法具有简洁、高效的优势.

当选定一条路径后,如果路径不包含压缩节点,我们采用基于符号执行的前向分析来获取路径的可达条件φ;如果路径中包含压缩节点,意味着路径中包含循环,如图 3(a)所示的路径 2→{3,4,5,11}*→12→Error,验证这类路径的难点是获取循环的归纳不变式.如果程序中以某种方式给出了循环不变式,则验证是平凡的;否则,我们采用抽象解释方法来获得循环不变式.抽象域可以根据需要灵活选择[17,18],如谓词抽象域、同余域、多面体域、区间域等.与采用有限展开循环的方法相比,本方法具有较高的可靠性和较低的漏报率.因此,在验证包含压缩节点的路径时,我们同样采用前向分析方法.当碰到压缩节点时,就利用获取的循环不变式继续分析,直到路径的终点.在获取路径的可达条件φ后,就利用 SMT 求解器来判定φ是否可满足:如果φ不可满足,则返回验证成功;否则,返回验证失败.

2 控制流图压缩及路径编码

本文提出的方法主要面向命令式程序代码,验证的范围包括内存、类型等方面的安全问题,也可以对添加的断言进行检测.该方法适用于包含多个函数的程序验证,验证过程以单个函数为基本单位.对于函数中的函数调用,可以通过克隆、函数摘要等方式进行变换.我们的方法可以视为一个可扩展的验证框架,在给出了面向对象形式规格说明方法及相应的对象不变式说明后,可以用于面向对象程序的代码验证.我们使用控制流图(CFG)或控制流自动机(CFA)来表示程序[5].一个类似 C 的命令式程序可以很容易地转换成 CFG.

定义 2.1(控制流图). 控制流图 CFG 是一个有向根图,包含有限节点和边,可以形式化地表示为一个元组:

$$\langle L,L_e,l_0,E,V,op,\phi_0\rangle.$$

- L:程序位置有限集合,即 CFG 中节点集合;
- L_e:$L_e\subseteq L$,待验证的节点集合,即错误节点集合;
- l_0:$l_0\in L$,程序入口节点,即 CFG 的根节点;
- E:CFG 中边集合,表示程序的控制流.$E\subseteq L\times op\times L$,我们使用$(l,op,l')$或者$l\xrightarrow{op}l'$表示从节点 l 到节点 l' 的有向边,并且边被标记为 op;
- V:程序变量有限集合;
- op:在 V 上的操作集合,表示程序执行动作.op 由赋值语句或者条件语句组成,其定义如下:

$$\begin{aligned} cst &::= c\in\mathbb{Q}\\ var &::= v\in V\\ expr &::= cst \mid var \mid expr\oplus expr,\ \text{with}\ \oplus\in\{+,-,*\}\\ guard &::= expr\otimes expr,\ \text{with}\ \otimes\in\{<,\leqslant,=,\neq,\geqslant,>\}\\ op &::= guard \mid var := expr \end{aligned}$$

- ϕ_0:程序的起始状态,即程序变量在起始状态所满足的条件.

在程序变量满足起始状态ϕ_0的条件下,程序从 CFG 的起始位置开始执行,直到运行到最终位置.验证程序 l 处的对程序变量的断言ϕ成立,就需要验证程序以任何路径执行到 l 处ϕ都成立.

遍历 CFG 的所有路径以验证程序,对于大规模的程序来说是不可行的.另外,有许多无关路径是不可达的或者不影响验证结果.因此,验证过程并不需要遍历所有路径.对 CFG 进行切片,就是根据给定的切片标准,即一个变量的集合,判定一条边是否是验证所需要的相关边.如果不是相关边,就从 CFG 中去掉.假设 $vars(\phi)$表示断言ϕ中所出现的变量集合,$vars(op_e)$表示边 e 上的转换动作 op 所包含的变量集合,如果 $vars(\phi)$和 $vars(op_e)$中的变量存在数据相关性,则我们称边 e 是验证断言ϕ的相关边,否则,就是无关边.我们采用的切片判断准则简单、高效,因此适合大规模程序的验证.

为了进一步减少切片后 CFG 的路径空间,我们对$\mathbb{P}_{sli}$进行处理获得其分支图.为了获取分支图,首先要求出$\mathbb{P}_{sli}$中的最大强连通分量,然后将最大强连通分量抽象为一个节点,就得到了$\mathbb{P}_{sli}$的分支图.最大强连通分量和分支图的形式化定义如下:

定义 2.2(最大强连通分量). 设有向图 G':$\langle N',E'\rangle$是有向图 G:$\langle N,E\rangle$的子图,如果 G'满足:

(1) 对于任意的节点 $n_1,n_2\in N'$,存在一条 n_1 到 n_2 的路径,且存在一条 n_2 到 n_1 的路径;

(2) 图 G 中没有以 G'为子图的强连通分量,

则称 G'为图 G 的最大强连通分量.

定义 2.3(分支图). 设 G^{scc}:$\langle V^{scc},E^{scc}\rangle$为有向图,如果 G^{scc}满足:

(1) V^{scc} 包含的每一个节点对应于 G 的每个强连通分支;

(2) 如果在图 G 的对应于节点 u 的强连通分量中,某个节点和对应于 v 的强连通分量中的某个节点之间存在一条有向边,则边$(u,v)\in E^{scc}$.

我们就称 G^{scc} 为有向图 G 的分支图.

G^{scc} 是一个有向无回路图,因此避免了无限长度路径,便于后面的路径敏感验证.获取分支图的关键是获得强连通分量,求一个图的强连通分量可以使用线性算法,如 Kosaraju 算法或 Tarjan 算法,因此可以高效地处理较大程序的控制流图.

根据算法 1,在获取控制流图的分支图$\mathbb{P}_{mscc}$后,需要遍历路径进行验证.我们提出一种符号化的路径表示和遍历方法.由于任意两个节点之间的路径都是无环简单路径,因此适合采用布尔表达式的方式对路径集合进行编码.对于$\mathbb{P}_{mscc}$中任何一条路径 p,布尔变量 b_e 表示边 e 在路径 p 中.对于节点 $l\in L_{mscc}$,$out(l)$表示节点 l 的出边集合,$in(l)$表示节点 l 的入边集合.对于边 $e\in E_{mscc}$,用 $src(e)$和 $tgt(e)$分别表示边 e 的始节点和末节点.设边集合 $E_s\subseteq E_{mscc}$,谓词 $onlyone$ 表示路径中只有一条路径来自 E_s,定义如下:

$$onlyone(E_s)=(\vee_{e\in E_s}b_e)\wedge(\wedge_{e,f\in E_s,e\neq f,}b_e\rightarrow\neg b_f).$$

控制流图中到达某个节点的执行路径的布尔表达式Π_p可以用如下布尔约束的交来表示.$SAT(\Pi_p)$的每个解即表示一条路径:

(a) 经过起始节点 l_0,布尔约束表示为 $onlyone(out(l_0))$;

(b) 经过某个目标节点 l_e,布尔约束表示为 $onlyone(in(l_e))$;

(c) 路径中所有节点只有一条入边,除了起始节点 l_0,布尔约束表示为

$$\wedge_{e\in E_{mscc},src(e)\neq l_0}b_e\rightarrow onlyone(in(src(e)));$$

(d) 路径中所有节点只有一条出边,除了目标节点 l_e,布尔约束表示为

$$\wedge_{e\in E_{mscc},tgt(e)\neq l_e}b_e\rightarrow onlyone(out(tgt(e))).$$

定理 2.1. 布尔表达式Π_p的所有解表示了分支图$\mathbb{P}_{mscc}$中从起始节点 l_0 到目标节点 l_e 的所有路径.

证明:定理 2.1 的证明很直接,因为布尔表达式Π_p是条件(a)~条件(d)的并,因此它的解必须都满足这 4 个条件,而满足这 4 个条件即描述了一条合法的路径.另外,一条合法的路径,则必须满足这 4 个条件.

当验证完一条路径时,需要把这条路径从待验证路径集合中去掉.当用布尔表达式表示路径集合时,可以通

过添加约束条件来去掉某些路径.谓词 *ExcludeEncode* 表示不访问某条路径π, $ExcludeEncode(\pi)=\wedge_{e\in E_\pi}\neg b_e$.公式 $ExcludeEncode(\pi)\wedge \Pi_p$ 就可以表示将路径π从Π_p所表示的路径集合中去掉.

3 路径验证

当选定验证的路径上包含循环压缩节点时,我们首先利用抽象解释技术来获取节点的归纳不变式,即程序在循环路径上的状态空间所满足的约束断言.抽象解释是一种基于格的程序语义可靠近似理论,为各类程序分析与验证方法建立了一个统一的形式化框架.

设程序的具体状态表示为$\mathcal{S}$,抽象状态空间表示为$\mathcal{A}$,我们使用程序变量上的约束断言来表示抽象状态.

断言公式可以使用命题逻辑及算术逻辑等可判定的逻辑公式.每个断言$\varphi\in\mathcal{A}$表示了一个具体状态集合,即$[\![\varphi]\!]\in 2^{\mathcal{S}}$.路径上压缩节点代表了 CFG 上一个最大强连通分量,即程序中的循环路径.设这个循环的节点集合为 L,入口节点为 l_{in},出口节点为 l_{out}.如图 3 中的循环节点集合为{3,4,5,8,11},其入口节点为 3,出口节点也为 3.程序运行到 l_{in} 处的抽象状态 $\varphi_{l_{in}}$ 称为循环初始条件.映射ρ:$L\to\mathcal{A}$将 L 中的节点映射到程序抽象状态空间.

定义 3.1(**归纳映射**). 如果映射ρ满足如下两个条件,则称为归纳映射:

(1) $\varphi_{l_{in}}\vDash\rho(l_{in})$;

(2) $\forall e=(l_i,op,l_j),\rho(l_i)\wedge op\vDash\rho(l_j)$.

求循环压缩节点的归纳不变式就是求$\rho(l_{out})$,下面介绍基于抽象解释框架以求解归纳映射.

显然,逻辑蕴涵关系$\vDash$是抽象域$\mathcal{A}$上的偏序关系,因此,偏序集$\langle\mathcal{A},\vDash\rangle$是一个完备格,其底元和顶元分别是$\bot$和$\top$.同理,$\langle 2^{\mathcal{S}},\subseteq\rangle$ 是具体状态空间上的完备格.抽象算子 $\alpha:2^{\mathcal{S}}\to\mathcal{A}$ 和具体算子 $\gamma:\mathcal{A}\to 2^{\mathcal{S}}$ 构成了 $2^{\mathcal{S}}$ 和$\mathcal{A}$上的 Galois 连接.利用传统的抽象解释求解不动点的方法即可求出归纳映射.我们从起始映射ρ_0 开始不停地迭代,产生映射序列$\rho_1,\rho_2,\ldots$,过程如下:

$$\rho_0(l)=\begin{cases}\varphi_{l_{in}}, & l=l_{in}\\ \bot, & \text{else}\end{cases},\ \rho_{i+1}(l_n)=\vee_{(l_m,op,l_n)}sp(\rho_i(l_m),op),$$

其中,*sp* 表示本文后面提到的最强后置条件.如果迭代第 k 次时满足$\forall l\in L.\rho_{k+1}(l)\vDash\rho_k(l)$,则停止迭代.由于$\langle\mathcal{A},\vDash\rangle$满足上升链条件,因此迭代过程最终会收敛.

定理 3.1. 设程序通过任何路径执行到 l_{in} 的状态满足 $\varphi_{l_{in}}$, 则程序通过任何路径执行到 l_{out} 的状态满足$\rho(l_{out})$.

证明:定理 3.1 的正确性是基于抽象解释理论.利用抽象解释求解的归纳映射,实际上描述了每个循环节点上程序可能的运行状态,是对精确状态的抽象.$\rho(l_{out})$抽象描述了循环出口处所有可能的程序运行状态,因此定理 3.1 成立. □

当一条路径上循环节点的归纳不变式获得以后,采用符号执行的方法来验证路径是否可行.符号执行是指在不执行程序的前提下,用符号作为值赋给变量,对程序的路径进行模拟执行.符号执行过程中的状态包括 3 部分:变量到其符号值的映射、程序计数器和路径条件.路径条件是一个关于程序输入变量的符号值的约束,一组输入值使得程序沿着某条路径执行当且仅当这组输入值满足这条路径的路径条件.我们利用符号执行获取路径条件,然后利用 SMT 求解器验证路径条件是否满足.与传统的符号执行不同,我们使用最强后置条件技术来获得路径条件,这样可以提高精确度,减少误报率.对于边(l,op,l'),设程序在 l 处的状态空间满足φ,则记 $sp(\varphi,op)$为程序从 l 运行到 l'处后对应的最强条件,它满足两个性质:

(1) 如果程序的状态空间满足φ,则执行完语句 *op* 后的状态空间必定满足 $sp(\varphi,op)$;

(2) 如果执行完语句 *op* 后的状态空间不满足 $sp(\varphi,op)$,则可以断定执行前的状态空间不满足φ.

op 由赋值语句和条件语句组成,其最强后置条件定义如下:

- $sp(\varphi,x:=e)=\exists x'.\varphi[x'/x]\wedge(x=e[x'/x])$;
- $sp(\varphi,\varphi')=\varphi\wedge\varphi'$,

其中,$\varphi[x'/x]$与 $e[x'/x]$表示将表达式中所有的自由变量 x 替换为 x'后得到的表达式.

设 $l_0 \xrightarrow{op_0} l_1,\ldots,l_n \xrightarrow{op_n} l_e$ 为 CFG 中一条路径,程序的起始状态满足ϕ_0,则这条路径的最强后置条件为

$$sp(\ldots sp(sp(\phi_0,op_0),op_1),\ldots op_n).$$

利用 SMT 求解器判定其是否可满足,即可对路径进行验证.

定理 3.2. 设控制流图中路径 l 的起点为 l_{in},终点为 l_{out},l_{out} 处的待验证断言为ϕ.利用上述方法求出路径 l 的验证条件φ如果不可满足,则任何经过 l 的程序执行到 l_{out} 时使得ϕ成立.

证明:由算法 1 可知,最后求取的验证条件φ是由程序在 l_{out} 处的抽象状态逻辑表示(记为 s)与$\neg\phi$的并,即 $s\wedge\neg\phi$.验证条件φ不成立,则意味着$\neg(s\wedge\neg\phi)$成立.由命题逻辑可知,$\neg(s\wedge\neg\phi)$与 $s\rightarrow\phi$等价,所以 $s\rightarrow\phi$成立.即,任何经过 l 的程序执行到 l_{out} 时使得ϕ成立. □

4 实验与分析

我们利用 F-Soft[19]程序验证平台对本文的方法进行了实现.F-Soft 可以对 ANSI-C 程序进行验证,验证的性质包括缓冲区溢出、字符串 API 的使用、空指针引用、用户自定义的类型状态属性、内存泄漏等.我们利用 F-Soft 的前端生成 CFG,然后使用数据流分析对 CFG 进行切片.我们使用 Z3[12]作为 SMT 求解器,因为 Z3 支持位向量、数组、结构以及有理数的关系、算术表达式等,Z3 还支持自定义公理,为验证复杂的程序性质提供了支持.F-Soft 实现了几种不同抽象域的抽象解释框架[19],我们利用它来获取循环不变式.实验的验证主要基于算法 1 的描述,实现包括前端和后端,其示意图如图 4 所示.前端中的预处理是利用函数克隆技术对函数调用进行处理,然后对每个函数构建控制流图,基于待验证的属性对控制流图进行切片,然后压缩控制流图中的循环路径.最后,利用数值抽象域如常量、区间、多边形等对循环路径进行抽象解释分析,获得循环不变式.在前端中,构造 CFG、切片和抽象解释利用了 F-Soft 平台提供的功能.前端得到的包含循环不变式的 CFG 交给后端处理,后端实现的主要部分是对路径进行编码和符号执行.在获取路径的过程中,我们使用 Z3 求解可满足性问题,在符号执行和路径验证中,我们使用 Z3 对相关约束进行判定.

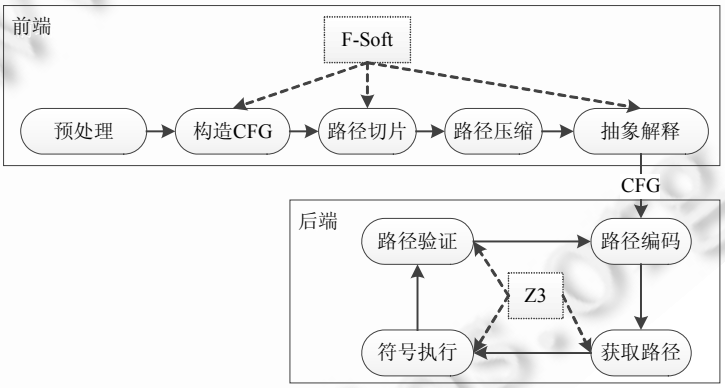

Fig.4 Experimental setup

图 4 实验具体实现示意图

为了验证本文方法的有效性,我们与同样是基于 SMT 求解器的程序验证工具 SMT-CBMC[20]进行比较.SMT-CBMC 利用边界模型检测技术对 C 程序进行验证.我们从一些标准的基准测试包里选取典型的测试用例进行测试.实验结果见表 1,我们从 SMT-CBMC 模型检测工具[20]的基准测试包中选取前 6 个测试用例,这几个测试用例的主要特点是包括数组和循环.其中,BubbleSort,SelectionSort 分别实现了桶排序和选择排序,BellmanFord 程序实现了带权图的最短路径算法,StrCmp,SumArray 和 MinMax 分别实现了字符串比较、数组求和以及求最大最小值算法.接下来两个用例来自 SNU-RT 基准包[21],分别实现插入排序和 Fibonacci 函数.第 9 个测试用例来自 MiBench 基准包,它实现了立方方程的根求解.第 10 个测试用例来自 CBMC 手册[22],它利

用位操作实现两个数的乘法.第 11 个测试用例来自 High Level Synthesis 基准包[23],其中用到了大量的位操作,是典型的嵌入式软件代码.前面 11 个测试用例所验证的属性、使用到的数据结构比较全面,可以很好地检验验证方法的有效性.另外,我们从基准测试包 SNU-RT 和 NXP[24]里选取了几个规模较大的程序进行实验,进一步验证本文方法的有效性.验证失败的原因主要是超时或者内存不足.我们的实验环境为 Intel Core2 T6500 2.1GHz 的 CPU,内存大小为 1G,操作系统为 Ubuntu 11.04.从实验结果可以看出:本文方法的验证时间有所减少;同时,验证的成功率也有所提高.这是因为,SMT-CBMC 对循环进行了展开处理,同时不支持 ANSI-C 的一些结构,如位操作、浮点数算术、指针算术等.

Table 1 Experimental results

表 1 实验结果

测试用例	代码行数	断言个数	时间(秒)		失败个数	
			Our	SMT-CBMC	Our	SMT-CBMC
BubbleSort	43	17	1.07	1.87	0	0
SelectionSort	34	17	0.72	0.70	0	1
BellmanFord	49	33	0.3	0.89	1	2
StrCmp	14	6	23.1	26.3	0	1
SumArray	12	7	4.01	3.45	0	0
MinMax	19	9	4.67	7.24	0	0
InsertionSort	86	17	1.45	3.11	1	2
Fibonacci	83	4	9.20	13.12	0	0
Cubic	66	5	0.14	1.78	0	0
BitWise	18	1	21.78	30.20	0	1
Adpcm_encode	149	12	4.11	6.83	0	4
sensor	603	167	211.34	302.41	0	0
jfdctint	374	331	178.01	221.17	4	13
exStbHwAcc	1432	113	430.82	493.51	0	5

5 相关工作

近几十年来,程序验证[25]一直被认为是提高软件可信性的有效方法,但是对程序的完全正确性验证仍然面临精度和效率上的困难.在验证程序满足某方面的性质方面,许多研究者做了大量研究并取得了许多有益的成果.Strom 和 Yemini[26]很早就通过类型状态分析来提高软件可信性.在发现软件缺陷方面出现了很多典型工具,如:Metal[27]是一个错误检测工具,它的核心是一个局部的数据流分析,但是路径不敏感的;PREfix[28]是一个通过遍历程序路径进行符号求值的错误发现工具,它在许多大规模程序中发现了很多缺陷.PREfix 使用有界路径遍历减少路径空间;LCLint[29]结合一个扩展的类型系统和数据流分析实现对程序的检测,并且可以利用函数的断言注释.程序验证的一个重要方法是使用定理证明来保证软件的可信,典型的工具是 ESC-Java[30],它使用定理证明的方法来验证 Java 函数是否满足给定的前置和后置断言.ESC-Java 需要程序员在代码中添加注释来完成局部分析,Flanagan 等人研究了如何自动添加注释[31].与之相比,我们的方法自动化程序更高一些.模型检测[32]实际上也是路径敏感的程序验证方法,SLAM[11]就是一个通过迭代求精的模型检测工具,它开始对程序进行粗略的近似,然后以目标制导的方式对抽象不断求精.BLAST[10]工具则是使用惰性抽象和基于反例的抽象求精技术.有界模型检测技术(BMC)[33]也是用了 SAT 或 SMT 求解器对程序进行验证,这类工具包括 CBMC[22],F-Soft[19],Saturn[34]等.与我们的方法不同,他们对循环进行了有限的深度展开.另外一些路径敏感的分析工具包括 ESP[4],trace partitioning[35],elaborations[36]等,这些工具使用一种启发式方法来缩小路径搜索空间.本文中对路径进行编码的技术借鉴了 Beyer 等人[37]的工作,他们也使用布尔公式对控制流进行抽象,但是没有对路径搜索空间进行压缩.

6 结 论

路径敏感的程序验证是一种较为精确的验证方法,但是由于路径爆炸问题,限制了它的使用范围和规模.本文提出一种基于 SMT 求解器的验证方法,使用循环路径压缩和 CFG 切片对路径空间进行缩减,同时利用布尔表达式抽象方式对路径空间进行遍历.在验证的过程中,使用 SMT 求解器对逻辑公式加以证明或对约束进行求

解.本文的方法综合使用了多种技术,确保了验证的实用性和扩展性,实验结果表明,与类似方法相比,本文的方法在精确度上有明显提高,同时效率上也有所改进.

空间爆炸问题一直是路径敏感程序分析和验证所面临的主要挑战,虽然本文使用了多种方法进行优化,但是仍然有改进的空间.我们接下来的工作主要研究如何利用已验证过的路径寻找不必要验证的路径,这主要是考虑到 CFG 中大部分路径的可达性只取决于路径中某些边的转移条件.另外一个继续研究的方向是,对循环路径进行更加精确的处理、对不同的验证目标使用不同的抽象域、对深度比较浅的循环路径使用边界模型检测方法进行验证.

References:

[1] Liu K, Shan ZG, Wang J, He JF, Zhang ZT, Qin YW. Overview on major research plan of trustworthy software. Bulletin of National Natural Science Foundation of China, 2008,22(3):145−151 (in Chinese with English abstract).

[2] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969,12(10):576−580. [doi: 10.1145/363235.363259]

[3] Lahiri SK, Qadeer S. Back to the future: Revisiting precise program verification using SMT solvers. ACM SIGPLAN Notices (Special Issue on Proc. of the POPL 2008), 2008,43(1):171−182. [doi: 10.1145/1328897.1328461]

[4] Das M, Lerner S, Seigle M. ESP: Path-Sensitive program verification in polynomial time. ACM SIGPLAN Notices, 2002,37(5):57−68. [doi: 10.1145/543552.512538]

[5] Henzinger TA, Necula GC, Jhala R, Sutre G, Majumdar R, Weimer W. Temporal-Safety proofs for systems code. Lecture Notes in Computer Science, 2002,2404:526−538. [doi: 10.1007/3-540-45657-0_45]

[6] Tip F. A survey of program slicing techniques. Journal of Programming Languages, 1995,3(3):121−189.

[7] Jhala R, Majumdar R. Path slicing. ACM SIGPLAN Notices (Special Issue on Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation), 2005,40(6):38−47. [doi: 10.1145/1064978.1065016]

[8] Cousot P, Cousot R. Abstract interpretation frameworks. Journal of Logic and Computation, 1992,2(4):511−547. [doi: 10.1093/logcom/2.4.511]

[9] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394. [doi: 10.1145/360248.360252]

[10] Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with BLAST. Lecture Notes in Computer Science, 2003,2648:235−239. [doi: 10.1007/ 3-540-44829-2_17]

[11] Ball T, Rajamani SK. Automatically validating temporal safety properties of interfaces. Lecture Notes in Computer Science, 2001, 2057:102−122. [doi: 10.1007/ 3-540-45139-0_7]

[12] Moura LD, Bjørner N. Z3: An efficient SMT solver. Lecture Notes in Computer Science, 2008,4963:337−340. [doi: 10.1007/978-3-540-78800-3_24]

[13] Bruttomesso R, Cimatti A, Franzén A, Griggio A, Sebastiani R. The MathSAT 4 SMT solver. Lecture Notes in Computer Science, 2008,5123:299−303. [doi: 10.1007/978-3-540-70545-1_28]

[14] Dutertre B, Moura LD. A fast linear-arithmetic solver for DPLL(T). Lecture Notes in Computer Science, 2006,4144:81−94. [doi: 10.1007/11817963_11]

[15] Detlefs D, Nelson G, Saxe JB. Simplify: A theorem prover for program checking. Journal of the ACM, 2005,52(3):365−473. [doi: 10.1145/1066100.1066102]

[16] Barrett C, Tinelli C. CVC3. Lecture Notes in Computer Science, 2007,4590:298−302. [doi: 10.1007/978-3-540-73368-3_34]

[17] Cousot P. Abstract interpretation based formal methods and future challenges. Lecture Notes in Computer Science, 2001,2000:138−156. [doi: 10.1007/3-540- 44577-3_10]

[18] Cousot P, Cousot R. Basic concepts of abstract interpretation. In: Renè J, ed. Proc. of the Int'l Federation for Information Processing (IFIP). Boston: Springer-Verlag, 2004. 359−366. [doi: 10.1007/978-1-4020-8157-6_27]

[19] Ivanăić F, Yang ZJ, Ganai MK, Gupta A, Shlyakhter I, Ashar P. F-Soft: Software verification platform. Lecture Notes in Computer Science, 2005,3576:301−306. [doi: 10.1007/11513988_31]

[20] Armando A, Mantovani J, Platania L. Bounded model checking of software using SMT solvers instead of SAT solvers. Lecture Notes in Computer Science, 2006,3925:146−162. [doi: 10.1007/11691617_9]

[21] SNU real-time benchmarks. http://www.cprover.org/goto-cc/examples/snu.html

[22] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. Lecture Notes in Computer Science, 2004,2988:168−176. [doi: 10.1007/978-3-540- 24730-2_15]

[23] High level synthesis benchmark suite. 2000. http://mesl.ucsd.edu/spark/benchmarks.shtml

[24] NXP. High definition IP and hybrid DTV set-top box STB225. 2009. http://www.nxp.com/

[25] Nelson G, Oppen DC. Simplification by cooperating decision procedures. ACM Trans. on Programming Languages and Systems (TOPLAS), 1979,1(2):245−257. [doi: 10.1145/357073.357079]

[26] Strom RE, Yemini S. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. on Software Engineering, 1986,12(1):157−171.

[27] Engler D, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. of the 4th Conf. on Symp. on Operating System Design & Implementation. New York: USENIX Association, 2000. 1−1. http://dl.acm.org/citation.cfm?id=1251229&picked=prox&CFID=158599953&CFTOKEN=20224954

[28] Bush WR, Pincus JD, Sielaff DJ. A static analyzer for finding dynamic programming errors. Software Practice and Experience, 2000,30(7):775−802. [doi: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H]

[29] Evans D. Static detection of dynamic memory errors. ACM SIGPLAN Notices, 1996,31(5):44−53. [doi: 10.1145/249069.231389]

[30] Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended static checking for Java. ACM SIGPLAN Notices, 2002,37(5):234−245. [doi: 10.1145/543552.512558]

[31] Flanagan C, Leino KRM. Houdini, an annotation assistant for esc/Java. Lecture Notes in Computer Science, 2001,2021:500−517. [doi: 10.1007/3-540-45251-6_29]

[32] Clarke EM. Model checking. Lecture Notes in Computer Science, 1997,1346:54−56. [doi: 10.1007/BFb0058022]

[33] Biere A, Cimatti A, Clarke EM, Zhu YS. Symbolic model checking without BDDs. Lecture Notes in Computer Science, 1999,1579: 193−207. [doi: 10.1007/ 3-540-49059-0_14]

[34] Xie YC, Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability. ACM Trans. on Programming Languages and Systems (Special Issue on Proc. of the POPL 2005), 2007,29(3):1−43. [doi: 10.1145/1232420.1232423]

[35] Mauborgne L, Rival X. Trace partitioning in abstract interpretation based static analyzers. Lecture Notes in Computer Science, 2005,3444:5−20. [doi: 10.1007/ 978-3-540-31987-0_2]

[36] Sankaranarayanan S, Ivančić F, Shlyakhter I, Gupta A. Static analysis in disjunctive numerical domains. Lecture Notes in Computer Science, 2006,4134:3−17. [doi: 10.1007/11823230_2]

[37] Beyer R, Henzinger TA, Majumdar R, Rybalchenko A. Path invariants. ACM SIGPLAN Notices (Special Issue on Proc. of the 2007 PLDI Conf.), 2007,42(6):300−309. [doi: 10.1145/1273442.1250769]

附中文参考文献:

[1] 刘克,单志广,王戟,何积丰,张兆田,秦玉文.“可信软件基础研究”重大研究计划综述.中国科学基金,2008,22(3):145−151.

何炎祥(1952－),男,湖北应城人,博士,教授,博士生导师,CCF 高级会员,主要研究领域为可信软件,并行分布处理,嵌入式系统.

陈勇(1986－),男,博士生,主要研究领域为嵌入式软件,绿色优化,可信软件.

吴伟(1985－),男,博士生,主要研究领域为可信软件,嵌入式系统,编程语言.

徐超(1980－),男,讲师,CCF 学生会员,主要研究领域为可信软件,嵌入式系统.