doi:10.3772/j.issn.1002-0470.2019.06.005

基于最弱前置条件的程序正确性分析①

郭莎莎② 侯春燕③ 王劲松

(天津理工大学计算机科学与工程学院 天津 300384)

摘 要 随着软件的不断更新迭代,软件正确性检测的必要性愈加凸显,软件正确性检测的处理时间直接决定软件的维护成本。动态测试的断言编写和静态分析的符号执行均针对程序正确性进行优化完善,但分析结果易出现路径缺失甚至错误无法识别等问题。现有验证方法在路径扩展时易生成较多无用路径,且针对性不强,因此有必要研究一种更为可靠的方案。本文采用最弱前置条件对软件可行性加以分析,对程序执行语义正确建模,使用程序切片技术预处理程序代码,并根据层级结构存储节点及其子程序。实验结果表明,该方法可以有效减小静态分析对程序状态抽象操作带来的验证精度损耗,且能够遍历求解出程序的所有可能路径,并通过分组标出条件表达式的结论真假值,以此验证路径正确性,同时可对高复杂的程序代码进行有效的正确性分析。

关键词 程序正确性, 最弱前置条件, 静态分析, 路径扩展, 程序切片技术

0 引 言

程序正确性对一个程序或者程序系统至关重要,现有的程序分析重视程序的正确性,但完全的正确性验证仍旧难以实现[1]。依靠人工方法进行软件错误定位无法满足正确性测试需求,所以软件测试技术中的自动分析技术成为必需。

最弱前置条件是表示实现程序执行结果的最基本输入条件[2]。它来源于三公理语义,是一种后推度量标准[3]。程序中的最弱前置条件可以正常执行程序执行结果的最基本的输入条件,这样可有效满足相关后置条件中限制最小的前置条件[4]。此前有使用程序切片技术讨论小范围的问题,运用最弱前置条件去推论程序的潜在错误[5-7],还有使用缺陷模式及其有限状态机描述程序属性进行软件测试[8]。此外,还有采用汇编程序检查器寻找简单程序的错误[9],如果一条路径中有很多不同的谓词,那么这条路径不可行的概率比较大。最弱前置条件也应用于静态分析技术中的误报消除,对程序抽象过程中的误报发现具有可靠性[10]。杨宇等人[11]在文中概括了静态分析常用的多种策略,并对当前的静态分析现状进行了比较。程序正确性验证也使用半自动化工具寻找程序断言并与满足断言的条件进行比较来验证程序正确性[12]。虽然程序正确性分析已经有很多研究,但是处理大程序的分析方法仍然不够精确高效,并且软件中的各类缺陷也是导致错误频发的关键所在。

程序设计过程中未知的错误出现概率很大,排除输入错误和语义语法错误仍然无法杜绝错误的发生,这成为程序正确性解决的一大难题。因此,本文在程序中加入最弱前置条件,通过检测潜在错误和导致未知结果的传播路径来保证程序的正确性。文章将研究重点放在基于 C 语言的简单线性分配程

① 国家自然基金(61402333, 61272450),天津市自然科学基金(18JCZDJC30700)和赛尔网络下一代互联网技术创新项目(NGII20160121)资助。

② 女,1992 年生,硕士生;研究方向:软件测试; E-mail: 13072232371@163.com

③ 通信作者,E-mail: chunyanhou@tjut.edu.cn

(收稿日期:2018-09-06)

序,侧重判断 C 语言程序的最弱前置条件和可行路径分析,注重考虑选择结构和 for 循环结构的嵌套,并加入程序执行次数的统计功能和层级结构,通过对其可行路径的判断来验证程序的正确性。

1 理论模型

最弱前置条件的多路径是从一个或多个选择结构、循环结构和它们的嵌套结构中扩展得到。罗旭等人[2]描述了一个序列结构,其中任何赋值语句或模块的后置条件都可以直接对应当前语句或模块的最弱前置条件。文中定义了几种演绎形式来描述这些语句或模块。

1.1 赋值语句

赋值语句是将一个特定的值赋给一个变量,它的最弱前置条件由式(1)推导。对赋值语句的处理是结构的第一步,因为非赋值语句与后置条件的求解无关。

$$\begin{cases} WP(x:=E,Q)=Q_E^x,\ x\in \mathrm{var}(Q) \\ WP(x:=E,Q)=Q,\ x\notin \mathrm{var}(Q) \\ WP(skip,Q)=Q \end{cases} \tag{1}$$

赋值语句处理的一大难点是表达式含有未知数时如何确定未知数的位置。直接调用数学软件去解决会造成运行时间长和稳定性不足的问题。所以本文将算数表达式调整为赋值表达式。因为未知数被设置为最高优先级,所以判断除未知数以外的其他算术优先级时,即使这些算术优先级包含未知数,也可通过递归调用方法将这些具有较高算术优先级的位置看作单个变量。

1.2 if-else 选择结构

if 子句、else-if 子句和 else 子句的程序节点之间具有并列关系,所以本文在处理字符串时将 if 语句和 else 语句看作一个整体字符串来截取。然后进行循环找出各分支关键字并对各字符串进行截取。最后将所有并列的程序节点存储在设定的节点序列的二维数组的同一行中。而 else 的条件不等式可以通过 if 的条件不等式求反得到。本文讨论变量的范围并得到最弱前置条件,因此不能将 if 选择结构的 2 个分支路径的语句块做并集,这可能导致不完整的路径覆盖。仅包含一组 if-else 的选择结构可以根据路径扩展特性将 2 条路径扩展为 4 条路径。if 选择结构的最弱前置条件由式(2)推导出。

$$WP_d((if\ P\ S_1;else\ S_2),Q) = P\wedge WP(S_1,Q)\vee\neg P\wedge WP(S_2,Q) \tag{2}$$

1.3 switch 选择结构

switch 选择结构中主要考虑 case 关键字和 default 关键字。当程序路径分析到 case 语句时,上一节点的后置条件适用于每一条 case 语句的前置条件,分别生成各自路径。为防止程序路径出现相互污染现象,case 语句子程序间或者对应的后置条件相同时,每条语句仍单独分析。所以 switch 结构的路径根据 case 语句条数决定。

1.4 循环结构

当 for 循环结构只具有赋值语句时,可以将循环体中的赋值语句按照提前确定的循环次数倍增,直接将循环结构改为赋值语句处理。但是在 for 循环体中加入嵌套结构后,子程序复杂化,上述方法不再适用。由于 for 语句的循环次数可以从语句后的表达式中求解,所以 for 语句的 predicate 属性存储括号中的初始化语句、条件判断语句和迭代语句,在分析最弱前置条件时单独分析,以此来获得 for 循环语句的具体循环次数。通过调用函数判断中间表达式符号,同时在链表中保存 for 条件的有效信息。在迭代次数已知的情况下,循环体可以展开,同顺序语句处理方式相同。另外,环结构还需要考虑循环条件无法执行的问题。其最弱前置条件可由式(3)得出。

$$WP(for\ P\ S,\ Q) = WP(Fun_{S|P},\ Q) \tag{3}$$

2 算法实现

手动输入程序代码后,使用程序切片技术等方法对代码进行处理和执行。使用最弱前置条件验证程序正确性,即使用程序已有节点求解得到最后一个最弱前置条件,并将其作为当前程序节点的后置条件,然后将当前程序节点中包含的代码语句作为参数进行最弱前置条件计算。按照此方法对整个程序内的所有程序节点进行后推分析,直到后推至首

节点时结束,据此得到的首节点的最弱前置条件即为整个程序的最弱前置条件。

2.1　预处理程序代码

图 1 是对代码预处理过程。

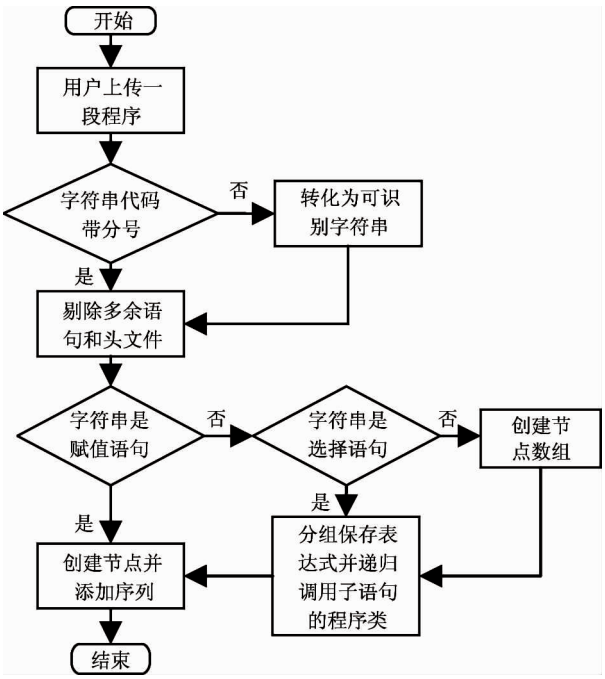

图 1　预处理程序代码

上传测试程序后进行解析,判断上传程序是否为带有分号的字符串代码,如果不符合条件,将其转化为可识别的字符串。然后剔除字符串中的多余语句和头文件。再根据字符串关键字判断字符串是否为赋值语句。如果是就截取到其后的第一个分号为止,创建该字符串的赋值语句节点,并且添加到节点序列中;如果关键字是选择结构或循环结构,截取字符串到子语句末端,并创建程序类,运用递归调用方法处理。字符串是选择语句时,按照分支分组保存表达式并递归调用子语句的程序类;字符串是循环语句时,需创建节点数组,该数组可看作动态列长的二维数组,数组的行数代表程序的总体执行长度,列数代表当前行所代表的程序节点的分支数量。最后遍历程序节点并存储到节点序列中。

图 2 中的程序是处理后得到的主要字符串。本章剩余小节均使用该段代码进行分析。

2.2　确定程序入口

程序的第一个待处理模块称为程序的入口。根据路径扩展特性和最内连接特性[2],将图 2 所示代码的程序入口定为 else 选择分支的最后一条语句。将没有后置条件和前置条件的节点复制到路径节点序列中进行后置条件分析并存储条件信息。如果 if 选择结构中的 else 选择分支未在程序中列出,则自动添加 else 分支条件,将子语句设定为空。图 3 表示嵌套结构的分支分析和程序入口的位置判断。

```
int main()
{a=1;b=2;
    for(int i=2; i<=4;i++)
        {a=3×i+1;
          if(a<11)
              (b+1)/4-2=i;
          else
              b×2-1=a;
        }
}
```

图 2　处理后的代码结果图

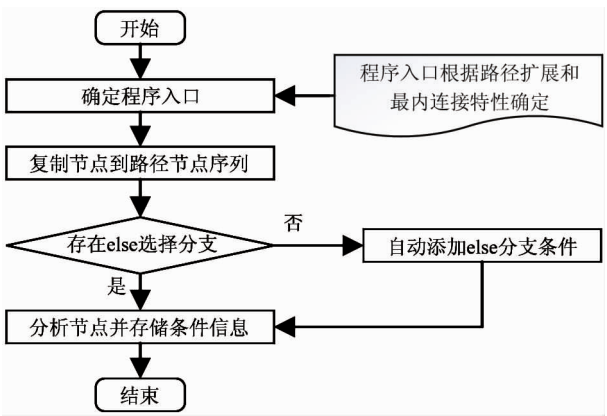

图 3　嵌套结构的分支分析和程序入口的位置判断

2.3　推导前置条件

创建 WPforAssign 类去单独提取待处理的某一赋值语句节点,防止节点之间相互影响和污染。如图 4 所示,每条赋值语句的后置条件是其对应的上一条语句的前置条件,所以系统读取该后置条件,并且运用简单的数学运算将表达式形式做出更改,再将后置条件代入字符串中。此外,需要分析赋值语句的等号中是否包括已有的未知变量,或未知变量表达式是否存在于后置条件的条件不等式中,如果存在,则需要化简赋值语句并将与变量相等的表达式代入后置条件且化简;否则该赋值语句将被跳过,仅判断该不等式的真假并存储后置条件。最后根据字符串表达式推出该语句的前置条件。

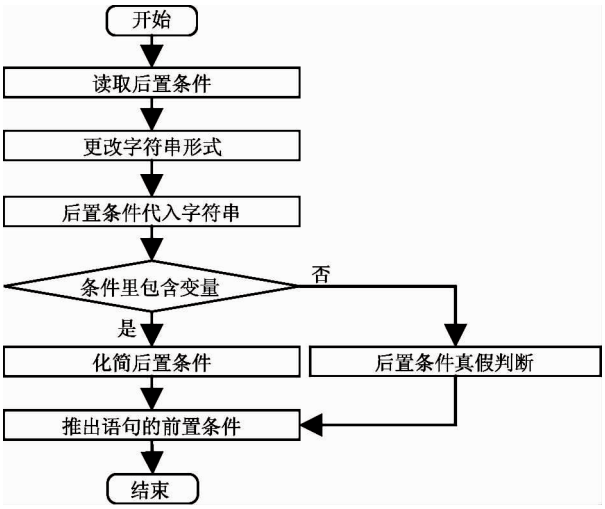

图 4　处理赋值语句的后置条件并求解前置条件

2.4　推导最弱前置条件和扩展路径

根据上节方法求解得到每一条语句的前置条件和后置条件,最终推导出程序的最弱前置条件以及扩展的路径。具体步骤如下:

步骤 1　依据路径扩展特性,找到程序节点关键字 for,然后通过定义 predicate 属性提前推出循环次数。

步骤 2　定义 addsubpaths( )方法,将 for 循环体按照循环次数拆分为若干个循环一次的循环结构,区分出主路径和依附路径。通过节点和后置条件生成新路径并且将新路径加入路径序列。其中依附路径以进入分支所需要满足的条件表达式作为后置条件。此外,当一组路径中所有前置条件判断为真时,这组路径中的主路径就是可行性路径。if结构的条

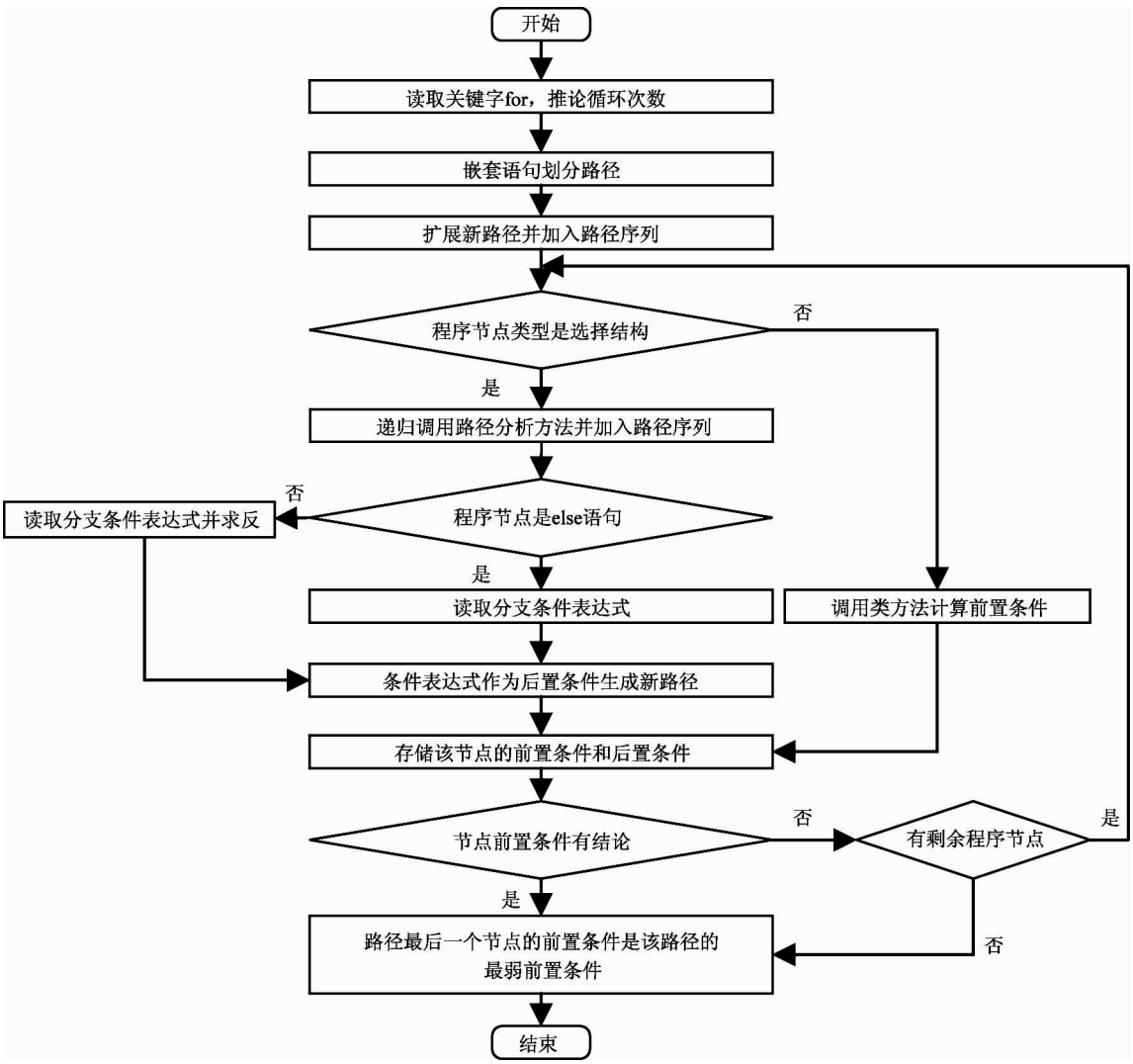

图 5　程序最弱前置条件求解和路径扩展

件不等式可作为当前节点的父节点的后置条件生成另一条依附于该路径的新路径，以此作为当前条件选择分支是否能够被执行的判断条件。若这条路径的前置条件为假，那么即使被依附的路径可以一直分析下去，并且得出真的前置条件，也不能将其视作可行性路径。

步骤 3　递归调用程序路径分析方法，将当前分析节点、对应后置条件以及新的路径需要分析的起始节点作为形式参数生成一条新的程序路径。根据程序节点类型是否为 else 选择分支语句判断读取进入该分支的条件表达式是否需要求反。然后将条件表达式作为后置条件生成新路径。

步骤 4　所有语句的后置条件和前置条件推出后，存储在相应的路径节点序列中。然后判断最后一个节点的前置条件是否可以得出结论。如果得出结论，这个前置条件是此路径的最弱前置条件。如果不能得出，需要判断程序节点是否有剩余。剩余就继续遍历处理，重复此节步骤直至所有节点均处理。此时得到的最后一个节点是该路径的最弱前置条件。

图 5 表示程序最弱前置条件推出和路径扩展执行顺序。第一个待处理节点的后置条件是程序运行后的最后输出结果，此结果通过在界面中手动输入实现。

3　性能分析

3.1　包含嵌套的 for 循环结构的结果分析

本节分析图 2 中预处理后的程序代码。系统处理代码后会根据层级结构存储节点和节点的子程序。图 6 中，系统将 2 条赋值语句和 for 循环结构分为 3 大模块并属同一层级，而 if 选择结构的 2 条分支为第二层级，对于选择条件以及其求反语句为第三层级。这验证了系统可以对较为复杂的程序代码正确分析程序部分并且正确截取。

第一个节点的后置条件“b < 7”手动输入后，根据循环次数 3 次和嵌套的 if 选择结构分析，该程序的路径会扩展为 8 条主路径和 56 条依附路径，并且每一条主路径都有2条相关的依附路径，可行路径

```
0->a=1
1->b=2
2->for(int i=2;i<=4;i++) {a=3×i+1;if(a<11)
      0->a=3×i+1
      1_0-ifbranch->a<11
            0->(b+1)/4-2=i
      1_1-elsebranch->a>=11
            0->b×2-1=a
```

图 6　程序节点序列层级展示图

在主路径中推出。扩展路径的推论顺序按照图 7 展示的层级依次逆推。图 7 中路径 1 是可行路径，通过对选择条件作为后置条件时得到的前置条件是否为真得出的路径，依照层级逆推可正确得到程序的最弱前置条件。该条路径讨论循环次数 $i=4$ 时选择条件求反后作为后置条件进入表达式分析并且循环次数为$i=2$，3时选择条件直接作为后置条件进入表达式分析的情况。路径 2 是测试结果中的一条不可行路径。图中箭头表示方向即为程序执行走向，每一条表达式的右侧不等式（组）为其对应的后置条件，左侧不等式（组）为求取的前置条件，上一条表达式的前置条件和下一条表达式的后置条件相同。前置条件判断路径真假值在图中用文字表示。其他不可行路径与路径 2 展示的不可行路径判断方式相同，通过测试可知其余不可行路径在求解程序前置条件时均有假值的情况。

表 1 表示图 7 结果图中的层级关系。从表中可知，不同路径下的层级关系是一样的，验证了图 7 中程序节点序列层级分布的正确性。

表 1　不同路径分层

	路径 1	路径 2
第一层级	第 0 步	第 0 步
	第 27 ~ 29 步	第 27 ~ 29 步
第二层级	第 1 ~ 2 步	第 1 ~ 2 步
	第 6 ~ 11 步	第 6 ~ 11 步
	第 15 ~ 21 步	第 15 ~ 21 步
	第 24 ~ 26 步	第 24 ~ 26 步
第三层级	第 3 ~ 5 步	第 3 ~ 5 步
	第 12 ~ 14 步	第 12 ~ 14 步
	第 21 ~ 23 步	第 21 ~ 23 步

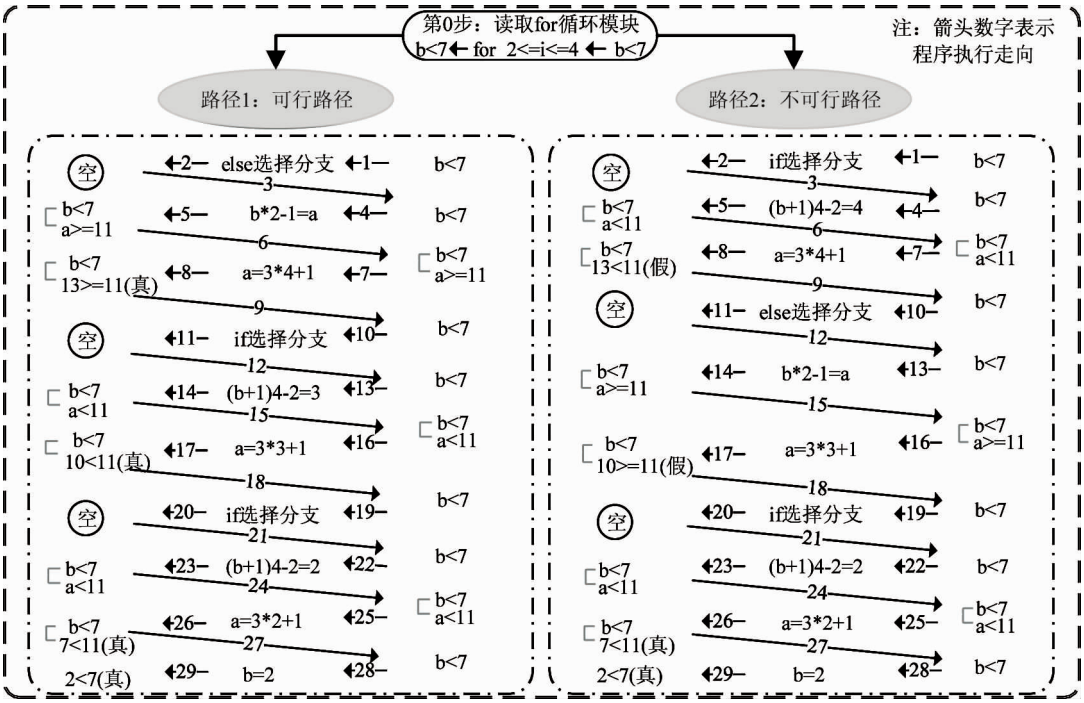

图 7　程序可行路径和一条不可行路径结果图

4.2　两种选择结构和 while 循环结构分析

图 8 所示测试用例包含 2 种选择结构和 while 循环结构,其中 if 选择结构包含嵌套部分。该测试用例路径可扩展为 8 条主路径和 32 条依附路径,图 9展示可行路径和主路径中的一条不可行路径。因为 while 循环结构的循环体无法提前推出,所以将循环条件处理为表达式,并将循环结构做一次循环处理,并根据循环条件将循环结构分为两条路径,循环条件求反的情况循环体设为空值。switch 选择结构的 case 语句中,将常量表达式的值赋值到变量中,并作为表达式进行处理。用例中 case 语句有 2 条,所以将路径扩展为 2 条。

```
int main ()
{ int b ,a;  b=20,a=10;
 if(a>20){ b+=10; if(b>10) {a=a-15;} else {a+=1;} }
 else {b-=10;}
 while (b>5) { b+=2; }
 switch (b) { case 10: b=b×2;
              case 30: b=b/2; }
return ; }
```

图 8　包含选择结构和 while 循环结构的测试用例

4.3　代码覆盖程度比较

推导线性分配程序最弱前置条件的半自动建模方法(semi-automatic modeling methods for deducing the weakest precondition of linear assignment program,以下简称 SAM 方法)[2] 研究建模方法和设计算法来推导线性分配程序的最弱前置条件,以找出预期的或未知的执行结果的原因。表 2 详细列举了 SAM 方法和本文方法对于代码覆盖程度的差异。指出本文所用方法的改进和扩展,代码覆盖程度得到提升;程序执行次数的统计功能和层级结构的加入,使代码模块的质量得到优化。该方法的提升减少了软件测试的成本,并提高了程序测试的正确性。本文在实现原方法所包含内容的基础上,对 switch 选择结构和包含嵌套的 for 循环结构也进行了测试,得到本章两个小节所展示的所有测试结果。

5　结　论

长期以来,程序正确性一直受到计算机科学界和工业界的广泛关注。程序正确性和程序结构设计密不可分。软件系统投入运行前提高程序正确性会大幅缩减后期支出。本文采用最弱前置条件对程序进行分析和预测,提高了程序正确性,减少软件测试的成本。但是程序路径在分支节点数增加时会呈指

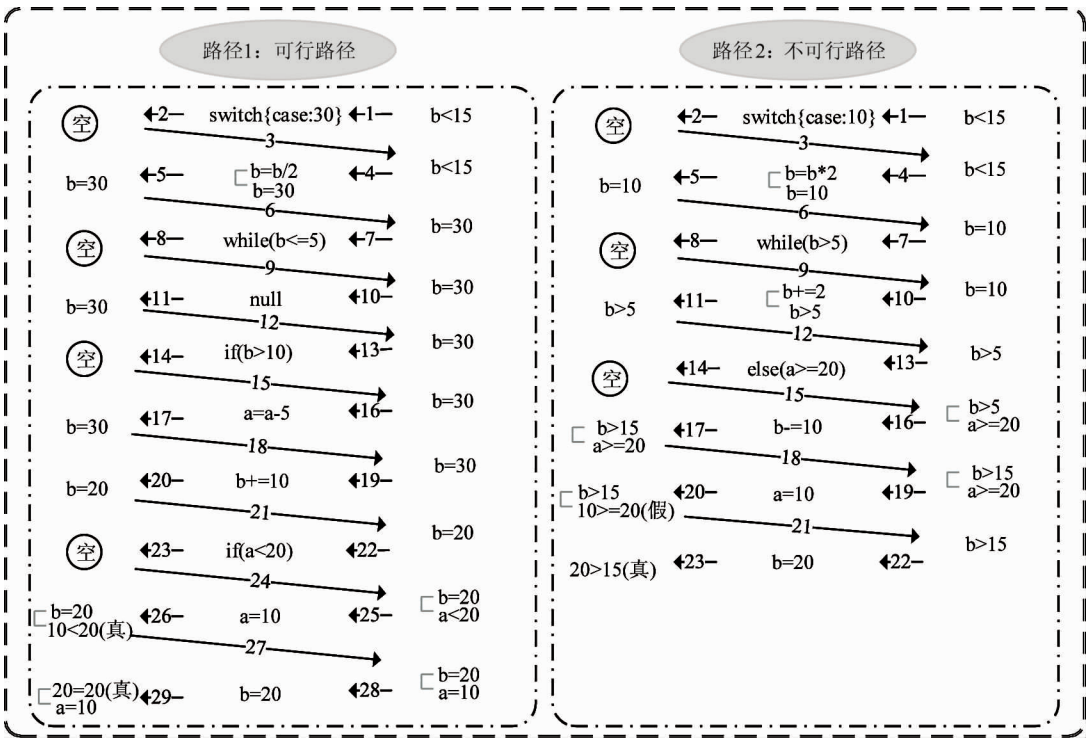

图 9　两种选择结构和 while 循环结构的测试结果图

表 2　基于最弱前置条件的程序代码覆盖程度的比较

	赋值语句	多分支的 if 选择结构	包含嵌套的 if 选择语句	switch 选择语句	一般形式的 for 循环结构	包含嵌套的 for 循环结构	while 循环结构
SAM 方法	√	√	√	×	√	×	√
本文方法	√	√	√	√	√	√	√

数增长,即使引入最弱前置条件会减缓部分现状,可是对系统占用率较大的问题仍然无法解决。在以后的研究过程中,我们将进一步优化循环结构条件未知时如何推导最弱前置条件,并提升兼容性问题。

参考文献

[1] 张健. 精确的程序静态分析[J]. 2008, 31(9): 1549-1553

[2] Luo H, Liu X, Chen X, et al. Software reliability analysis using weakest preconditions in linear assignment programs[J]. *IEEE Transactions on Software Engineering*, 2016, 42(9): 866-885

[3] Hoare C A. An axiomatic basis for computer programming [J]. *Communications of The ACM*, 1969, 12(10): 576-580

[4] 郭曦,王盼,王建勇,等. 基于 k 近邻最弱前置条件的程序多路径验证方法[J]. 计算机学报, 2015, 38(11):2203-2214

[5] Weiser M. Program slicing[C]. In: Proceedings of the 5th International Conference on Software Engineering, San Diego, USA, 1981. 439-449

[6] Tip F. A survey of program slicing techniques[J]. *Journal of Programming Languages*, 1995, 3:121-189

[7] Wang J, Yi X, Yang X. Towards a framework for scalable model checking of concurrent C programs[C]. In: Proceedings of the 2006 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos, Cyprus, 2006. 355-362

[8] 肖庆,杨朝红,宫云战. 提高静态缺陷检测精度方法[J]. 计算机辅助设计与图形学学报, 2010,(11): 2037-2044

[9] Flanagan C, Rustan K, Leino M, et al. Extended static checking for Java[C]. In: Proceedings of the ACM Con-

ference on Programming Language Design and Implementation, Berlin, Germany, 2002. 234-245

[10] 陈杰. 基于最弱前置条件的静态分析误报消除技术[J]. 计算机工程与应用,2012,48(33):1-4 +33

[11] 杨宇,张健. 程序静态分析技术与工具[J]. 计算机科学, 2004(2):171-174

[12] 刘杰,余童兰. 基于断言的程序正确性检测工具[J]. 电脑与信息技术, 2007(5):14-16 +21

Program correctness analysis based on the weakest preconditions

Guo Shasha, Hou Chunyan, Wang Jinsong

(School of Computer Science and Engineering, Tianjin University of Technology, Tianjin 300384)

Abstract

With the constant iteration of software, the necessity of software correctness detection accumulatively highlights and the time of software correctness processing detection directly determines the maintenance cost. The assertion of dynamic analysis and the symbolic execution of static analysis are optimized for the correctness of program, but the analysis results are prone to problems such as missing paths or unrecognizing errors. The existing verification methods easily generate more useless paths when the paths are extended, and the pertinence is weak, so it is neccesary to study a more reliable solution. The weakest preconditions are used to analyze the software feasibility. Then model program execution semantics correctly and preprocess program codes by program slicing technology. In addition, the nodes and their subroutines are stored according to the hierarchical structure. The experiment results show that the proposed method can effectively reduce the loss of verification accuracy caused by static analysis on the program state for abstraction operation, and traverse all possible paths of the program. Then by grouping and marking the true and false values of the conditional expressions the method can verify path correctness and analyze efficiently correctness of highly complex program codes.

Key words: program correctness, the weakest precondition, static analysis, path extension, program slicing technology