第37卷 第14期 计 算 机 工 程 2011年7月
Vol.37 No.14 Computer Engineering July 2011

·软件技术与数据库· 文章编号：1000—3428(2011)14—0068—02 文献标识码：A 中图分类号：TP311

基于条件值的C/C++预处理测试算法

娄坚波，刘久富，李金奎，王 伟

(南京航空航天大学自动化学院，南京 210016)

摘 要：传统C/C++代码的预处理分析利用符号执行推断预处理中自由变量条件表达式的值，但是该算法的时间复杂度是指数型的。为降低时间复杂度，提出一种快速符号执行算法。源代码通过词法分析器得到预处理变量和路径条件，为预处理变量建立节点，把路径条件转化为条件表达式，通过符号执行算法将两者整合为条件值 c-value 的形式，最终显示预处理结束后每一个预处理变量的条件值。实例结果表明，该算法能免去传统符号执行中对于路径可行性的分析，从而降低时间复杂度。

关键词：符号执行；预处理变量；条件值；条件表达式

Test Algorithm of C/C++ Preprocessing Based on Conditional Value

LOU Jian-bo, LIU Jiu-fu, LI Jin-kui, WANG Wei

(College of Automation Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

【Abstract】Traditional C/C++ code relying on preprocessing can be quite complex to analyze. Symbolic execution can be used to infer these expressions using the free variables, but its time complexity is exponential. In order to reduce the time complexity, this paper puts forward a method of fast symbolic execution. By using lexical analyzer, preprocessing variables and path conditions from source code are gotten. Next is building node for each preprocessing variable and converting path conditions into conditional expressions, they are merged into conditional values(c-values) through the algorithm of symbolic execution. The result is to display each preprocessing variable's c-value after pretreatment. Experimental results show that the path feasibility analysis of traditional symbolic evaluation can be omitted. So it can reduce the time complexity by using symbolic execution algorithm.

【Key words】symbolic execution; preprocessing variable; conditional value; conditional expression

DOI: 10.3969/j.issn.1000-3428.2011.14.021

1 概述

软件测试是保证软件质量的重要手段。静态测试是指不执行程序代码而寻找程序代码中可能存在的缺陷或评估程序代码的过程[1]。

符号执行通常也被理解为符号预测[2]，属于静态测试方法的一种，其基本思想是：用抽象的符号表示程序中变量的值来模拟程序的执行。符号执行的结果往往是一系列符号表达式，每一个符号表达式表示一个最终输出变量的值。

通常在C/C++代码的自动分析过程中，都是以假设预处理都已完成为前提的。但事实上，预处理过程中的自由预处理变量必须被赋予特定的值后，程序才能正确调用相应的.cpp文件。在大型软件系统中，广泛存在着文件的相互包含、条件式编译和各种宏之间相互关联和高度交错。如果没有足够的自动化工具，解析复杂的头文件将成为一件冗余乏味的工作。

本文采用一种基于条件值的快速符号执行算法，对C/C++的预处理过程，主要是条件编译进行预测，有效解决程序中每行代码的可达性、预处理变量的赋值及其条件值这些最主要的问题，保证软件测试高效稳定地运行[3]。

2 预处理变量与条件值

2.1 预处理变量

预处理指令是指那些以“#”开始的代码行，“#”后的标识符便是预处理指令名，C语言中的预处理指令集[4]如下：

#assert、#cpu、#define、#elif、#else、#error、#ident、#if、#ifdef、#ifndef、#import、#include、#include_nex、#line、#machine、#pragma、#system、#unassert、#undef、#warning。

#define 预处理指令通过记号列表绑定了一个预处理变量。假如定义的形式是参数化的，那么这就是“宏”；假如记号列表是空的，例如#define *Y*，则用符号∓表示*Y*的值，这种情况下的预处理变量是受约束的。相应的，若 *Y* 没有在#define 预处理指令中出现过，那么这类预处理变量是不受约束的，用符号⊥表示。在利用符号执行进行代码预测前，所有预处理变量的值是未知的，因此，具体将使用符号 *xi* 代替变量 *x* 的值，当然 *xi* 可能是∓、⊥或者记号列表中的具体值。

通过使用#if 预处理指令可以实现条件编译，例如图1中通过改变预处理变量的值改变预处理的编译行号。只有当 *X* 的值为2时，第5行中的预处理代码才会被编译。

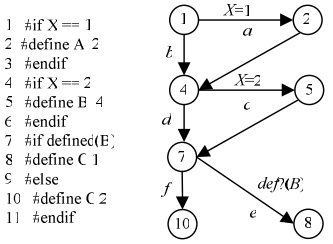

图1 一个C源程序及其控制流图

基金项目：南京航空航天大学基本科研业务费专项科研基金资助项目(NS2010069)

作者简介：娄坚波(1986－)，男，硕士研究生，主研方向：软件静态测试；刘久富，博士；李金奎、王 伟，硕士研究生

收稿日期：2011-01-14 **E-mail：**chongchunidechuang@163.com

2.2　条件值

本文用到一种新的符号替代方式，称为条件值或者 c-value，表示格式为：$c\rightarrow e1\diamond e2$。其中，$c$ 是一个条件表达式；$e1$ 和 $e2$ 可以是一个条件值也可以是普通表达式。条件值的含义为：当 c 为真时，那么相应预处理变量的值取为 $e1$，否则取为 $e2$。

如图 1 所示，在代码行 11 处，预处理变量 C 的值可以用条件值表示为 $def?(B)\rightarrow 1\diamond 2$，表明若 B 已被定义过，那么 C 的值为 1，否则为 2。采用条件值表示法后可以免去对于路径的组合分析，从而无需进行可执行路径判断和回溯等步骤，降低了符号执行算法的时间复杂度。同时条件值之间是可以进行嵌套的，利用条件值可以正确表示出代码中特定行中预处理变量的当前值。

2.3　条件值的简化公式

许多情况下条件值是带嵌套的，因此，需要相应的公式对其进行一定的简化。以下将介绍一些简化公式：

$$c\rightarrow e1\diamond e2\equiv c\wedge e1\vee\sim c\wedge e2 \quad (1)$$

其中，$e1$ 和 $e2$ 指的是布尔表达式。

$$def?(c\rightarrow e1\diamond e2)\equiv c\rightarrow def?(e1)\diamond def?(e2) \quad (2)$$

$$def?(\perp)\equiv \text{false} \quad (3)$$

$$c1\rightarrow(c2\rightarrow e1\diamond e2)\diamond e3\equiv(c1\wedge c2)\rightarrow e1\diamond(c1\rightarrow e2\diamond e3) \quad (4)$$

式中用到了一些常用的简化符号，如 $\wedge$ 和 $\vee$ 分别代表逻辑与、逻辑或。式(2)是针对 *def?*语句给出的基本规则。式(3)给出了 defined($\perp$)的含义。式(4)给出了带嵌套关系条件值的化简公式。

3　符号执行

3.1　程序设计的要点

本文设计实现了快速符号执行工具(FSE)来完成对目标程序的测试。程序中定义了 2 个较重要的结构体[5]，为预处理变量定义的结构体：

```
struct{
char *name;          变量名称
  TokenTypes type;  变量类型
  CString Ini_value; 初始值
  CString value;     当前值
  CString cond_list; 路径条件
}Preproessnode;
```

用链表的形式实现的栈 S，其栈中元素为：

```
struct{
  CString content;   表
  P_node *NEXT;      下一个节点
}P_node;
```

栈 S 存储了一组表，表是由预处理变量和其条件值共同组成的。通常来说表的形式为：$[A,Y_i==2\rightarrow 3\diamond 1]_1$。其中，[ ]右下角的数字表示其在栈 S 中所在的位置。

3.2　符号执行算法

图 2 给出了符号执行的流程，首先通过词法分析得到首个预处理变量，接着为其创建 Preproessnode 型节点 n 并初始化，将一个空表入栈 S 并调用函数 $E(n,Cc)$。条件表达式 Cc 用来表示当前代码行被执行到所需的条件。

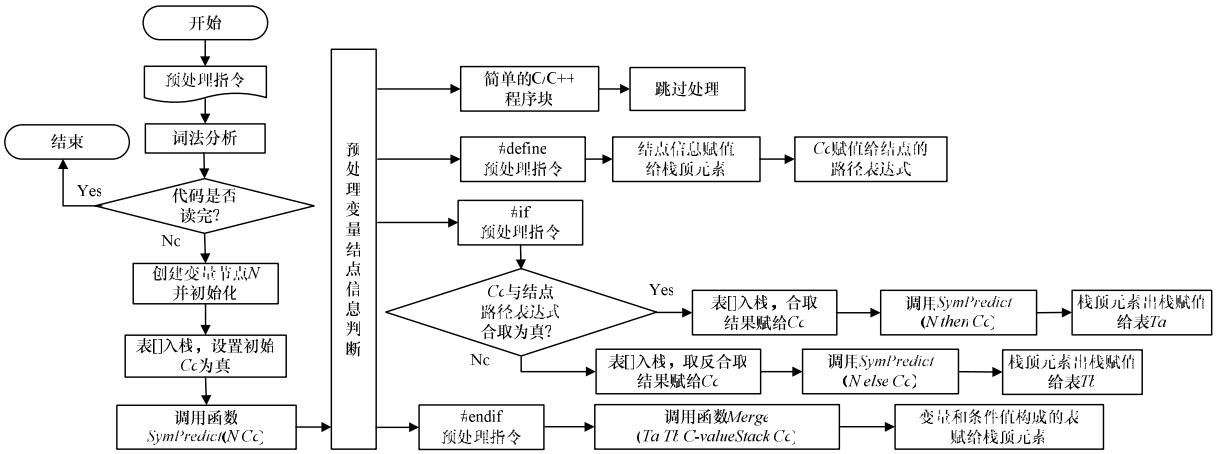

图 2　符号执行流程

在图 2 中，函数 $E(n,Cc)$针对节点 n 分 3 类处理，分别为 C/C++代码、define 型节点、if 型节点。

函数 $Merge(T1,T2,S,Cc)$用于对表 $T1$、表 $T2$ 中所包含的预处理变量进行整合得到条件值，并将表$[x,c\rightarrow vt\diamond vf]$作为预处理变量的最终值入栈。若变量 x 已经不是自由的预处理变量，算法会正确修改变量 x 的 Ini_value，保证条件值的正确。通过符号执行后，在栈 S 中将只会存在一个表，即为最终结果。

4　符号执行算法应用实例

4.1　实例

图 3 是利用工具 FSE 对例 1 进行预测的结果。图中只显示栈顶中表的值。在代码行 3 处，栈顶中表的值为$[(A,3)]$，因此，表 $T1$ 的值为$[(A,3)]$，这里#if 预处理指令之后并没有 else 代码块，因此 $T2$ 的值为空，执行 $Merge(T1,T2,S,Cc)$函数。由于代码行 1 中已经利用#define 预处理指令将变量 A 赋值为 1，因此 A 的 c-value 为 $Yi==2\rightarrow 3\diamond 1$。符号执行的最终结果预处理变量 C 的 c-value 为 $Yi==4\rightarrow(C,Wi==6\rightarrow 7\diamond Ci)\diamond Ci$，利用 2.3 节的公式对其进行简化为$(Yi==4\wedge Wi==6)\rightarrow 7\diamond Ci$。

例 1

```
0.
1. #define   A   1
2. #if   Y==2
3.   #define   A   3
4. #endif
5. #if   Y==4
6.   #define   B   5
7.   #if   W==4
8.       #define   C   7
9.   #endif
10. #endif
```

(下转第 72 页)