第 33 卷 第 1 期 计 算 机 工 程 2007 年 1 月
Vol.33 No.1 Computer Engineering January 2007
·软件技术与数据库· 文章编号：1000—3428(2007)01—0086—03 文献标识码：A 中图分类号：TP311.56

软件测试中代码分析与插装技术的研究

刘慧梅，徐华宇
（陕西国防工业职业技术学院，西安 710302）

摘 要：软件测试作为软件工程的重要组成部分对提高软件质量和可靠性起到了不可忽视的作用。软件白盒测试自动化工具 WBoxTool 中对标准 C/C++程序进行了可视化的插装和监视，收集程序动态运行信息，并进行可靠的测试分析。该文介绍了在工具 WBoxTool 中使用的代码分析和插装测试技术，并给出插装测试方法的一些实例应用和分析。

关键词：双向邻接链表法；函数调用树；函数性能分析；控制流插装

Research on Code Analysis and Instrumentation in Software Test

LIU Huimei, XU Huayu
(Shaanxi National Defence Industry Professional and Technical College, Xi'an 710302)

【Abstract】 As an important part of the software engineering, software test has a crucial effect on improving software quality and reliability. In white box test tool, WBoxTool, the program is tested automatically using visual instrumentation and monitoring technology for standard C/C++ programs. The instrumented program can be run automatically, and the program dynamic information which to be analyzed lately can be collected at the same time. The technology of code analysis and instrumentation test which has been used in the WboxTool is described, and some test examples are carried to illustrate this approach.

【Key words】 Bidirectional adjacent list; Function calls tree; Function performance analysis; Control flow instrumentation

1 概述

1.1 自动插装测试

软件插装技术是随着开发技术一同发展起来的技术，已经被用于软件动态转换系统、动态优化、软件安全性检查、不同平台间的二进制代码转换和程序执行监视、程序调试、程序性能调节等目的[4]。近几年，已有不少软件致力于动态修改和控制程序的运行，如Jalapeno[2]和Dynamo[3]动态优化程序利用程序运行时的最新信息进行代码转换；代码修改系统Vulcan[5]利用快速断点对运行的程序插装二进制代码。然而，这些系统都建立在非常特定的目的之上，对于用户来说，从插装内容定位到程序高层行为的可视化信息较少，而在软件测试时，被测程序运行时的高层信息对程序的调试和可视化观察更有价值[1]。因此，在自主开发的Linux平台下的软件白盒测试自动化工具WBoxTool中，使用面向代码级的插装静态插装。

1.2 静态插装的难点

自动进行软件插装的难点主要包括：插入桩代码的定位和方式，代码膨胀。由于大量的插入代码执行事件，使得程序性能下降。本文针对 Linux 平台下开发的软件白盒测试自动化工具 WBoxTool 中使用的插装方式，逐一解决上述问题，并给出相应的测试实验结果。

(1)为了确定插入桩代码的定位，需要对源程序进行词法分析和语法分析。在词法/语法分析阶段，建立程序结构图、一些静态表格以及一些关键插入点信息。

(2)为了防止插装内容过多，对插装内容进行设计，根据不同的需求进行不同的插装，即每次只能针对需要收集的信息进行相应功能段的插入，而不提供一次插装，同时获取多种信息。使得对于插装前后的代码量的控制可以做到允许范围内。

(3)由于在原有程序中插入大量非相关任务的代码，程序性能会有所下降。为了解决这一矛盾，一方面，通过插装功能单一化减少插装内容，另一方面通过对插装代码的严格性能要求和测试，降低对程序性能的影响。

2 被测程序代码分析

为了确定插装的位置，需要对程序源代码进行分解和索引，从而可以标识每一个插装位置。采用词法/语法分析器进行代码分析和信息收集，信息存储借助数据库。

2.1 基于 Lex 与 Yacc 的代码分析

lex 与 yacc 是自动生成编程语言词法和语法分析器的工具。这里，使用包括 C++语法中除模板外的标准 C++文法的 lex 和 yacc 源程序。

yacc 生成的语法分析器自下而上进行语法归约，当归约到不同的语法规则时，设计语义动作进行程序信息（类、函数、程序块等）提取，并存储入库。

2.2 相关静态代码分析

从数据库中获取程序基本信息后，主要进行项目全局类图的分析和局部类图的分析，并把分析结果以 UML 图的形式返回给用户。这里主要介绍全局类图的成图情况。

由于 C++不具备强制单根继承结构，导致如果仅对继承而言，全局类图很可能已经无法用一棵树型结构代表，也就是说，可能存在很多棵树构成的森林；再考虑到加入包含的关系，整个类图的结构只会变得更加复杂，因此必须寻找新的结构来完备地表示软件项目的类图结构。这里提出了图的

作者简介：刘慧梅(1976－)，女，工程硕士，主研方向：软件工程；徐华宇，助工

收稿日期：2006-03-08 **E-mail：**huimei_liu@163.com

一种新的存储方式：双向邻接链表法，如图 1 所示。

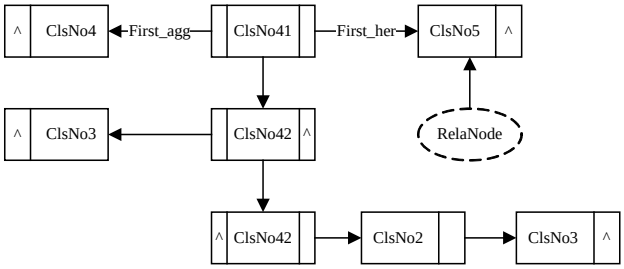


图 1 双向邻接链表

在显示全局类图之前，首先确定每个类的继承层次，层次深的优先访问，优先确定位置坐标。设计算法 dSearch，确定类的层次，同时深度优先遍历已建立好的类图结构 pGloList，确定每个类的继承优先级及属性优先级。为了保证全局类图的完整性，每个类节点设置访问标志。对于非连通类图来说，通过访问标志可以保证类图的每一个分支都得到完整显示。

3 插装类型

这里设计了 3 种类型的插装，运行时可以根据需要选择进行。

(1)函数性能分析插装

在每个函数入口和退出处插入跟踪代码，维护一个计数器，统计函数调用次数；记录本次进入、退出时间，计算本次执行时间；维护该函数调用的 max、min、avg、cumulative 时间，该函数执行时间在总程序中所占比率。

(2)函数调用序列跟踪插装

在每个函数入口和退出处插入跟踪代码，维护一个被调用深度等级，记录函数调用层次。

(3)函数控制流分支跟踪插装

在每个函数入口和退出、在每个分支部分插入跟踪代码。

4 算法设计及实现

4.1 函数调用序列插装

图 2 为函数调用序列模块设计，灰色部分为 instrumentor 在被测程序运行时工作，白色部分为 inserter 进行的准备与分析工作。

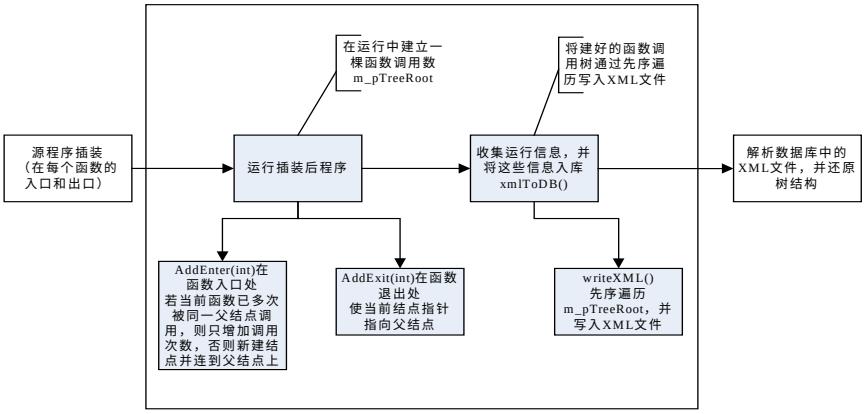
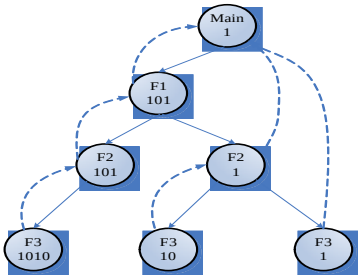


图 2 函数调用序列模块

函数调用序列采用树来收集每个函数的调用。当一个函数被调用执行时，就在其父结点上增加子结点，而当该函数退出时，为了从树上找到当前结点的父结点，树中每个结点还需要维护指向父结点的指针。因此，建立函数调用树时采用带父结点指针的孩子兄弟二叉树表示。当调用树建立后，将树信息以合理方式存储入库，以便于以后的分析和查找。

这里考虑到 XML 语言的灵活性和可定义性，采用 XML 作为中间转接层。在此设计的 XML 文件，只具有一种类型的结点元素 function，该元素具有 3 个属性，funcName，funcID，callCount；该元素根据调用情况嵌套使用；并且这里的 XML 文件元素只有起始卷标和结束卷标的嵌套组合或空字符。图 3 (a)所示为某测试项目在插装中建立起来的函数调用树，其相应的 XML 文件如图 3(b)所示。



(a) 某测试项目的函数调用树

```
<?xml version="1.0" ?>
<!--   A function call sequence description.      -->
<function funcName="main" funcID="1" callCount="1">
  <function funcName="CMyTest::f1" funcID="2" callCount="101">
    <function funcName="f2" funcID="3" callCount="101">
      <function funcName="f3" funcID="4" callCount="1010" />
    </function>
  </function>
  <function funcName="f2" funcID="3" callCount="1">
      <function funcName="f3" funcID="4" callCount="10" />
  </function>
  <function funcName="f3" funcID="4" callCount="10" />
</function>
```

(b) (a)中函数调用树生成的 XML 文件

图 3 函数调用序列的 XML 表示法

4.2 算法 1

算法 1：函数调用树建立算法（参数为待插函数的 ID）

(1)若根结点 pTreeRoot 不为 NULL，转(3)；否则申请新的树结点，并置新结点的调用次数为 1，函数 ID 为新函数的 ID，新结点的左右孩子及父指针均为 NULL；

(2) 置 pTreeRoot 指向新结点，置 pCurNode 指向新结点，转(6)；

(3)判断待插函数结点是否已经是当前函数（pCurNode）的子函数．若是子函数，则该子函数的调用次数加 1，并令 pCurNode 指向子函数结点，转(6)；否则，转(4)；

(4)申请新的树结点，并置新结点的调用次数为 1，函数 ID 为新函数的 ID，新结点的左右孩子为 NULL，父指针指向 pCurNode；

(5)将新结点作为 pCurNode 的子结点插入，令 pCurNode 指向新结点；

(6)结束。

4.3 算法 2

算法 2：将调用树写入 XML 文件算法 writeXML()

建立新的 XML 文件，首先写入 XML 声明及注释，进入遍历二叉树的递归算法，因为建立树的时候采用左孩子右兄弟的二叉树来存储树，所以遍历采用先序遍历，可以得到的遍历序列为（根、左孩子、右兄弟），对应为函数调用的（父函数、第一个子函数的调用序列、其它子函数的调用序列）。

void CInstrumentor:: preOrderTraverse(FILE* xmlFp，Node *node)；

这里，xmlFp 是 XML 文件指针，node 为当前要遍历的结点；无返回值。

(1)若结点 node 为 NULL，转(5)；否则，向 xmlFp 输出当前结点的函数信息，即建立一个当前函数的开始卷标<function funcName="…" funcID="…" callCount="…">；

(2)进入递归，遍历当前结点的左孩子；

(3)向 xmlFp 输出当前结点的结束信息，即建立一个当前函数的结束卷标</function>；

(4)进入递归，遍历当前结点的右兄弟；

(5)结束。

图 3 XML 文件解析的函数调用序列图如图 4 所示。

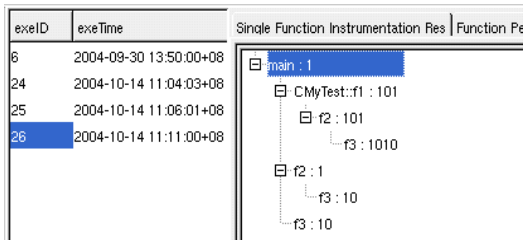


图 4 图 3 XML 文件解析的函数调用序列图

4.4 函数控制流分支跟踪插装

对一个函数的控制流来说主要有两部分，分支与循环的执行，分别对这两类进行单独的插装设计。

(1)分支插装，在分支进入点进行插装，每个分支通过不同的 ID 标识，当执行到当前分支时，instrumentor 收集当前分支的执行条件和经过路径（即当前语句 ID）；

(2)循环插装，在循环前、循环进入点进行插装，每个循环具有不同的标识，执行进入循环后，统计循环运行的次数，及记录循环进入、退出的边界条件。

分支与循环的识别是在使用 lex 和 yacc 进行静态分析时，将分支 ID 和循环 ID 信息全部存储入库。插装时根据分支和循环类型及位置插入相应的函数。为了唯一标识本次执行路径及循环执行情况，分支插装信息入库函数的参数是本次测试执行的 ID、分支 ID、执行本条路径的条件值；循环信息入库函数的参数是本次测试执行的 ID、循环 ID 和含有本循环的进入、退出条件、运行次数等的循环信息对象。

5 实验结果

5.1 插装对程序执行性能的影响曲线

插装对被测程序的性能影响如图 5 所示。

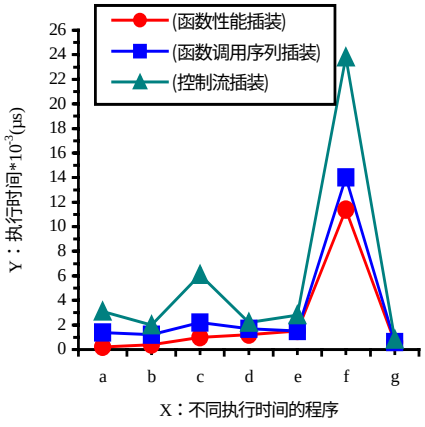


图 5 插装对被测程序的性能影响

从图 5 中可以看出，函数性能插装对被测程序影响最小，几乎与原程序运行时间重叠；而函数调用序列插装影响要相对大一些，控制流分支插装对程序的影响最大，同时其波动也很大，这是因为不同程序的分支及循环的使用情况不同.

值得指出的是，函数性能插装和函数调用序列插装只受函数调用多少的影响，不受函数长度的影响，因为它只在函数的首尾进行插装，被测程序代码越多，插装代码的影响越小；而函数控制流分支插装则与被测函数长度关系较大，随着分支数的增加，插装代码所占比例也随之增高。

5.2 函数性能插装分析结果显示

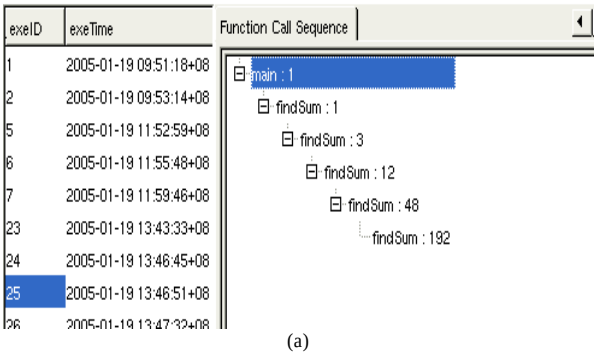
函数和子程序性能分析示意见表 1。

表 1 函数和子程序性能分析示意

Function Name	Numbers of calls	Min (μs)	Max (μs)	avg (μs)	cumulative (μs)	%Total time
findSum (int, int)	5	0.000 312	0.000 328	0.000 319	0.001 597	31.17%
findSum (int, int)	256	0.000 644	0.000 758	0.000 723	0.185 088	34.23%

5.3 函数调用序列结果显示

不同输入顺序的调用序列图比较见图 6，在图 6(a)中，输入：{1, 2, 3, 4}；N=19(exeID=25)；输出：3 4 4 4 4；执行时间：0.040835μs，在图 6(b)中，输入：{4, 3, 2, 1}；N=19(exeID=32)；输出：4 4 4 4 3；执行时间：0.023678μs。



(a)

```
<?xml version="1.0" ?>
<!--  A function call sequence description.  -->
<function funcName="main" funcID="1" callCount="1">
  <function funcName="findSum" funcID="2" callCount="1">
    <function funcName=" findSum " funcID=" 2" callCount="1">
      <function funcName=" findSum " funcID="2" callCount="1" >
        <function funcName=" findSum " funcID="2" callCount="1" >
          <function funcName=" findSum " funcID="2" callCount="1" />
        </function>
      </function>
    </function>
  </function>
</function>
```

(b)

图 6 不同输入顺序的调用序列图比较

findSum 为一个递归函数，比较上述的调用序列图，发现改变输入顺序，对于同样的输入和输出数字，递归次数和执行时间大大减少，因此使用插装不仅可以获知递归程序的递归深度和次数，并且有助于进行程序优化。

5.4 函数控制流分支跟踪插装

图 7 是使用约当消元法求解多元一次方程组的流程图，虚线显示了无解时的执行路径。根据图 7 中所示结果，语句

（下转第 91 页）