一个面向路径的软件测试辅助工具

邱晓康, 李宣东
(南京大学计算机软件新技术国家重点实验室, 江苏南京 210093; 南京大学计算机科学与技术系, 江苏南京 210093)

摘 要: 软件测试作为保证软件质量、提高软件可靠性的重要手段. 路径覆盖准则就是考察软件测试充分性的一种重要准则. 由于严格的路径覆盖测试不可实现, 选择一些对软件整体质量影响较大的重点路径进行测试, 以提高软件测试工作的效率和效益. 主要针对面向对象的软件系统提出了一种通用的基于统计的自动化辅助工具, 通过对程序代码的静态分析和插装, 以及由大量随机测试用例驱动所得到的统计分析结果, 为软件的功能测试和可靠性测试中的重点路径选择提供参考依据. 对该工具的主要思想、相应算法以及一些具体的实现问题进行了阐述.

关键词: 面向路径; 面向对象测试; 插装; 统计
中图分类号: TP311. 5 文献标识码: A 文章编号: 0372-2112 (2004) 12A-2301-04

A Path-Oriented Tool Supporting for Testing

QIU Xiao-kang, LI Xuan-dong
(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China*;
Dept. of Computer Science & Technology, Nanjing University, Nanjing, Jiangsu 210093, China)

Abstract: Software testing is an important method of assuring quality and enhancing reliability and path coverage is an essential criterion for testing adequacy. Since the rigid path-cover testing is infeasible, we select and test some key paths, which outweigh others on affecting the overall quality of the system, to enhance test efficiency and profit. A general statistics-based automated tool aiming at object-oriented software systems was proposed. By static analysis and interpolation on code, this tool offers a diagnosis of statistical data derived from continual executions, which are driven by random test cases, for information of selecting key paths in correctness testing and reliability testing. Primary idea, corresponding algorithms of this tool, and some problems in implementation are exposed.

Key words: path-oriented; object-oriented testing; interpolation; statistics

1 引言

测试充分性是软件测试的一个重要问题. 路径覆盖测试就是一种针对白盒测试的常用充分性准则, 它不仅要求软件测试过程中观察输入、输出的正确性, 而且要求观察程序运行的整个路径, 要求程序的运行覆盖所有完整路径. 在面向对象系统中, 路径的概念发生了变化, 可以看作是由代表方法的结点和代表消息的边组成的方法-消息序列, 又称为方法/消息路径(MM Path)[1].

路径覆盖是一种非常严格的覆盖准则. 但事实上, 即使是规模很小的程序, 包含的逻辑路径数量也是相当大的, 从而使得完全的路径覆盖在实际的软件测试中不可行. 在这种情况下, 测试员往往只能选择一个有一定效果且开销较小的覆盖准则, 如分支覆盖准则; 或者是根据一定的标准, 选择所有完整逻辑路径中的一个有限子集来进行测试, 如基本路径覆盖测试就是选择了一个能覆盖系统中所有状态的路径子集. 在本文中考虑另外一种客观而又简单的标准: 即客户实际操作中的路径执行频率. 一条路径在实际使用中的执行频率在很大程度上决定了其在软件测试过程中的重要程度. 在面向对象的软件系统中, 就体现为对象方法的调用频率. 一些在实际操作中调用频率较高的方法的质量, 对于软件整体质量的影响是非常大的, 因此我们有理由对它们进行重点或完全的测试, 特别是在相对完整的路径覆盖测试成本太高同时也不现实, 或者预算有限的情况下.

对用户实际使用的情况进行模拟, 常用的方法是 Beta 测试. 但 Beta 测试常常与实际情况差距较大, 而且会耗费测试人员大量时间, 成本过高, 实际往往不可行. 另一种模拟方法是使用随机测试技术的自动化工具, 即测试用例是根据随机数产生的, 也称猴子测试员[2]. 由于可以自动生成测试用例, 因此自动化程度高、成本低廉. 同时, 经大量随机用例测试通过的程序会提高用户对程序的信心, 从而得到令人满意和可信的结果.

收稿日期: 2004-09-02; 修回日期: 2004-11-18
基金项目: 国家自然科学基金项目(No. 60233020); 863 高技术研究开发计划项目(No. 2002AA116090); 江苏省自然科学基金项目(No. BK2004080, No. BK2002079); 英特尔未来科学家计划(No. ISRC2004)

在测试过程中程序产生中相关数据的获取和记录可以通过程序插装[3]来完成. 这是动态测试中常用的一种技术, 在保持被测程序原有逻辑完整性的基础上插入一些探针, 当被测程序运行时, 通过探针的执行采集程序的运行特征数据. 基于这些特征数据分析, 可以获得程序的控制流及数据流信息, 进而得到逻辑覆盖等动态信息.

2 功能

根据以上思路, 本文提出一种面向对象软件测试的自动化辅助工具. 该工具通过大量随机测试用例驱动插装后的被测程序运行, 记录下运行时的相关数据, 并对统计数据进行分析找到调用频率较高的类和方法, 作为重点测试时选择的参考依据. 本工具还可以对软件开发中的资源配置进行指导. 如在对软件测试之前进行, 或在测试基本完成后、有多余资源(包括金钱、时间、人力等)的情况下, 使用本工具得到的结果, 为进一步的补充测试指出方向; 或者在对系统进行测试的同时, 使用本工具来确定对被测系统运行效率影响较大的类和方法, 然后可针对该部分相关的测试内容作进一步的优化, 以最小代价、最大限度地提高系统的测试效率.

除功能测试外, 该工具还可以考虑应用于软件可靠性测试方面. 软件可靠性[4]是指软件在特定环境和给定时间内无失效运行的概率. 通过将系统划分为若干具有相互独立可靠性的模块并分别进行软件可靠性的测试, 然后使用本工具进行基于用户操作的随机测试统计, 得到在整体的软件可靠性测试中各个部分被运行的概率, 由此可推算出整个被测软件大致的可靠性水平, 从而降低可靠性测试的复杂度.

3 算法设计

本工具的设计可分为程序插装、随机用例生成、驱动运行和统计分析等几个模块. 整体的工作流程和框架见图 1. 下面对各个模块的功能进行描述.

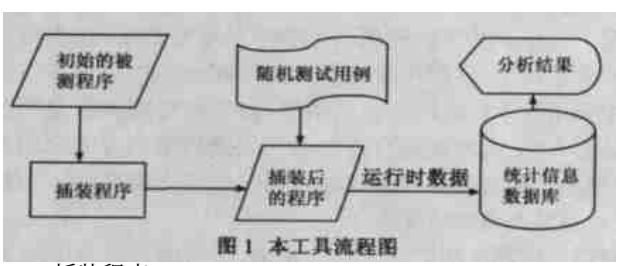

图 1 本工具流程图

3.1 插装程序

插装程序是本工具中的核心部分, 主要完成静态分析、创建数据库表和数据采集三个方面的工作.

静态分析是对被测程序代码进行静态扫描分析, 根据所用程序设计语言的语法规则, 找出其中的各个类、方法、构造函数、析构函数, 以及全局或静态变量等要素. 对于缺少析构函数或构造函数的类, 必须在代码中显式地为其增加, 否则插装程序中后续的数据采集等工作将无法正确完成. 我们采用的方法是将读入的源程序代码流匹配成由空格符、制表符、换行符等控制字符分隔成的标记流(大括号、小括号等特殊字符视作一个单独的标记), 通过对代码的一次顺序扫描, 完成所需的工作. 具体的算法流程如下所示.

算法 1 静态分析

```
(1)  public static void analyze() {// 内部变量定义及初始化;// 打开源程序文件;// 建立与数据库的连接;
(2)      while(token! = EOF) {// 读取下一个标记;
(3)          if(token= = "class") {// 读取下一个标记作为一个新类名;// 将当前类置为该类, 并将该类的记录写入数据库;
(4)              // 找到界定该类体的左右括号;}
(5)          else if(token= = "{") {
(6)              if (entry into a method) {// 读取前若干个标记作为方法名;// 将该方法的记录写入数据库;
(7)                  // 找到该方法体结束处的右括号;}
(8)              else if (entry into a constructor) {// 找到该构造函数的形参个数和类型;// 将该构造函数的记录写入数据库;
(9)                  // 找到该构造函数体结束处的右括号;}
(10)             else if (entry into a finalizer) {// 将该析构函数的记录写入数据库;
(11)                 //找到该析构函数体结束处的右括号;}}
(12)         else if (token = = "}") {if (escape from a class) {
(13)             if (no any constructor exists)// 为该类添加一个空的默认构造函数;
(14)             else if (finalizer not exists)// 为该类添加一个空的析构函数;}}}
(15)     //终止与数据库的连接;// 关闭源程序文件;}
```

我们还需要在统计信息数据库中建立相应的表, 用以保存运行时相关的统计信息, 如插装模块的运行次数, 类的生成实例、消亡实例和存活实例的个数, 方法的被调用次数、执行时间、最近执行时刻等. 对于每个被测模块, 我们在数据库中创建两个表——Classes_In_〈Project_Name〉和 Methods_In_〈Project_Name〉, 分别用来存放被测模块中类和方法的记录.

数据采集在被测模块的各个方法中插装语句, 这些语句完成的功能是采集数据库中所需的信息, 并更新数据库中相应的数据. 基于上文所示的静态分析程序, 可作如下插装: 在算法 1 的第 8、13 行后, 插装执行 SQL 语句的代码, 将表 Classes_In_〈Project_Name〉中当前类对象个数加 1; 在第 10、14 行后, 将当前类对象个数减 1; 在第 7 行后, 将表 Methods_In_〈Project_Name〉中当前方法调用次数加 1.

为保证插装后程序的鲁棒性, 在被测程序中插入的语句不仅包括修改数据库中的记录, 还应包括对各种异常情况的处理. 在被测模块中还要插装一些必要的初始化语句, 如引入插装语句要调用的组件和程序库, 建立与数据库的连接等. 以 Java 语言为例, 需要完成的初始化工作包括: 在源代码中插入 import 语句, 引入插装程序所在的程序库; 以及在适当位置加入处理对统计信息数据库进行读写的 ExecuteDB 类的对象定义.

3.2 随机用例生成

已出版的文献讨论随机测试问题已有至少 20 年的历史[5]. 主要是学术界在统计学方面对这个问题感兴趣. 前人的研究已经得到了一些结果和实用工具. 常用的方法是把测试用例表示为从初态开始经过若干个中间状态到达终态的状态

和边的序列. 从初态开始, 在每一个状态都生成一个 0~ 1 间的随机数, 根据这个数选择这个状态的一条出边, 转移到下一个状态. 这样产生的测试用例是随机的, 符合用户的使用习惯. 但这种方法是与问题领域无关的, 产生的随机用例往往不能很好地符合实际. 一种有效的改进是根据实践得到操作配置[6], 并据此得到转移至各个状态的不同概率.

在通常的软件随机测试中, 一个需要解决的问题是: 多少随机测试用例才是充分的? 一种典型的方法是用使用链和测试链的比较来解决这个问题. 但在本文提出的工具中, 随机用例生成只是为获取统计数据而使用的一项辅助技术, 并不作为主要的测试手段, 因此测试充分性问题的重要性已经大大降低. 本工具对测试用例的数量没有限制, 一般情况下只要能得到较为明显的、可分析出有效结论的统计数据即可终止.

3.3 驱动程序

驱动程序相当于一个主程序, 用以完善待测模块的运行环境, 以及对经过插装的被测模块进行编译和连接. 然后, 它接收不同的测试数据, 并把这些数据传送给被测模块, 驱动其运行. 驱动程序完成的另一部分工作是用户交互图形界面, 主要包括对待测项目的创建和管理, 项目中文件的添加、删除和状态管理, 插装后代码文件的管理, 评估准则的选择, 以及统计结果的查看分析等.

3.4 统计结果分析

本文所提出工具的目的是为了找到需要重点测试的类和方法, 即确定被测系统所有类和方法的一个真子集. 这需要对所得的统计结果进行分析. 一般的, 可以通过一个评估函数 $F(t)$ 和一个阈值 T 来判断一个类或方法是否需要重点测试. 评估函数和阈值可以通过大量实验数据的经验得到, 也可根据被测系统的特点由用户给定. 本文中假设了一个评估函数, 但该函数只是实验性的, 其有效性有待检验, 需要通过实验进行修改.

设 $C(i)$ 表示类 i 的对象实例个数, 则类的评估函数可表示为

$$F(t) = \sum_{\substack{C(i) \leqslant C(t) \\ \text{for all } i}} C(i) \Big\backslash \sum_{\text{for all } i} C(t) \qquad (1)$$

阈值 T 则确定为 0.5.

设 $E(i)$ 表示方法 i 的被调用次数, 则方法的评估函数可表示为

$$F(t) = \sum_{\substack{E(i) \leqslant E(t) \\ \text{for all } i}} E(i) \Big\backslash \sum_{\text{for all } i} E(t) \qquad (2)$$

阈值 T 仍然为 0.5.

4 系统实现

4.1 设计与限制

根据该工具的设计思想, 我们实现了一个针对 Java 语言的原型系统. 系统的开发环境为 Sun ONE Studio 4 CE, 数据库使用 Sun PointBase 4.3. 该系统没有包含随机用例生成的功能, 用户可以使用其他一些专门的生成工具来完成, 或者在被测程序中插入特定的生成代码. 这样的简化并不影响对工具设计思想和功能的体现.

4.2 实现技术

下面介绍一些在系统实现过程中遇到的问题及解决的方法, 和其中使用的一些改进和优化技术.

4.2.1 内隐类和嵌套结构 为产生高效的运行代码, Java 语言支持类定义的嵌套, 即内隐类(inner class). 并且内隐类的结构可以递归嵌套. 为在实现系统中支持这一极实用的结构, 我们在静态分析程序中引入了对类的嵌套结构进行记录的堆栈数据结构 level, 用以存储分析程序当前读入位置所处的类和方法环境嵌套序列. 当分析程序读入某一类或方法体时, 将该类或方法的名称作为当前环境推入栈中; 当结束该类或方法体的分析时, 则使其出栈. 因此, 一个类在栈中当且仅当读取源文件的当前位置处于该类的定义之中. 这样就可保证在任何时刻, 当有内隐类或其中的方法被定义时, 根据堆栈 level 中存储的信息, 都能找到其所定义的位置.

4.2.2 数据库访问的缓冲机制 为了尽量减少测试代码对被测系统运行速度的影响, 我们曾试图对数据库的每一个更新操作都采用建立一个独立线程来执行的办法, 但经过实验发现不可行. 由于测试时运行速度和产生线程的速度非常快, 一方面由于线程数量无限增长, 另一方面由于对硬盘上数据库中数据的频繁读写产生的瓶颈, 造成速度极其缓慢. 我们采用了一种建立缓冲区的解决办法. 在插装程序的初始化工作中, 增加建立缓冲区域的内容. 即根据测试时所涉及的数据库表在内存中建立缓冲区作为一个临时的虚拟数据库, 在插装语句中把对数据库的修改转化为对缓冲区内相应变量的读写. 在对数据库访问频率高、但涉及范围并不大的情况下, 可大大减少对被测系统效率的影响. 对数据库的物理读写则通过建立一个 Daemon 线程, 不断将缓冲区中被更新过的数据写入数据库. 这样可以保证数据库中数据能在一个可接受的时延内得到物理更新.

4.2.3 并发控制 一般情况下, 关系数据库系统中都提供了内置的并发控制机制, 在系统设计中无需考虑. 但由于本系统使用了上文所描述的缓冲机制后, 数据的并发访问实际上是在内存中进行, 数据库的并发控制机制无法发挥作用. 考虑到该问题, 在实现系统的设计中依靠了 Java 语言内置的同步机制(Synchronized)[7] 来解决. 系统将对缓冲区进行访问的关键语句置于声明 synchronized 的保护之下, 使得当同时有两个以上对同一数据记录进行访问的请求时, 必须被同步顺序执行, 从而保证了缓冲区中数据的正确性.

4.3 实例分析

4.3.1 实例选择 选择合适的实例软件进行分析是非常重要的. 系统规模不宜过大; 但又必须具有一定的复杂度, 以保证所得到的结果具有典型意义. 因此我们在实例的具体选择时遵循了以下两条标准:

- 初始程序运行一次的时间应足够短, 一般不超过 5 秒.
- 程序结构中包含若干类、一定数量的方法和足够多的执行路径.

本文选择了一个简单的二叉搜索遍历程序作为分析实例. 该系统共包含 3 个类和 12 个方法, 每次执行的路径随用户输入的不同会有很大差异. 为方便对随机测试的模拟, 对其

原来的主程序进行了修改, 将原来由用户在控制台输入指令和数据改为由一个与当前时刻相关的伪随机数序列产生. 这样使其行为可较好地模拟一次随机用例驱动的测试运行, 因此可以直接运行插装后的程序代码来获取统计数据.

4.3.2 运行分析 通过对插装后的二叉搜索程序代码的编译, 以及手动完成的 100 次运行, 在系统数据库中得到了一个可供分析的统计数据. 两个数据库表中的数据分别如图 2 和图 3 所示.

CLASS_ID	CLASS_NAME	PCLASS_ID	CREATED_OBJ_NO	KILLED_OBJ_NO	LIVE_OBJ_NO
1	BinarySearchTree	NULL	0	0	0
2	TreeNode	NULL	197	0	197
3	BinSTree	NULL	302	0	302

图 2 表 Classes_In_bst 中的统计数据

METHOD_ID	METHOD_NAME	EXE_TIME_NO	RECENT_EXE	CLASS_ID
1	public static void main(String[] args)	100	2004-04-22 12:51:53.706	1
2	Integer getValue()	1210	2004-04-22 12:51:53.808	2
3	void setValue(Integer given)	0	NULL	2
4	void leftSon(TreeNode given)	0	NULL	2
5	void leftSon(TreeNode given)	0	NULL	2
6	void rightSon(TreeNode given)	0	NULL	2
7	BinSTree leftSubtree()	117	2004-04-22 12:51:12.697	3
8	BinSTree rightSubtree()	145	2004-04-22 12:51:12.697	3
9	void insertNode(TreeNode given)	97	2004-04-22 12:51:52.808	3
10	void preorder()	150	2004-04-22 12:51:14.94	3
11	void inorder()	100	2004-04-22 12:51:10.373	3
12	void postorder()	175	2004-04-22 12:51:14.94	3
13	TreeNode search(Integer target)	150	2004-04-22 12:51:12.697	3

图 3 表 Methods_In_bst 中的统计数据

根据公式(1)和(2), 对以上得到的数据进行计算分析, 可以得到以下结果:

- F(BinSTree) = 1, 是唯一超过阈值 0.5 的重点类
- F(getValue()) = 1, 是唯一超过阈值 0.5 的重点方法

这个结果并没有让人感到非常满意: getValue() 函数的调用频率虽然远远超过了其他方法, 但该函数仅包含一个语句, 而我们希望能找到其他一些更加典型的重点方法. 造成结果不理想的一个主要原因是评估函数和阈值的选取不当. 如果阈值降低 0.02, 方法 inorder() 就达到了重点方法的标准. 由此可见, 评估函数和阈值的选取在很大程度地影响了本工具的测试结果和有效性.

5 总结

本文工作的主要思想在于通过在面向对象软件测试中度量路径(类和方法)的相关重要性, 为设计高效的测试方案提供依据. 众所周知, 虽然路径覆盖是确保软件测试充分性的关键, 但进行完全的路径覆盖测试是不现实的. 即使是相对充分的路径覆盖测试也要消耗大量的人力和财力资源, 极高的成本常常使开发机构望而却步. 当前的研究结果表明, 一些典型的软件企业仅测试了其开发的 30% 的源代码, 一个重要原因就是成本太高. 因此, 在无法进行完全测试的前提下, 提高软件测试效率的方法及工具的研究具有重要的现实意义.

参考文献:

[1] Jorgensen P. Object-oriented integration testing[J]. Communication of ACM, 1994, 37(9): 30-38.

[2] Patton R. 软件测试[M]. 周予滨, 等, 译. 北京: 机械工业出版社, 2002.

[3] Huang J C. Program instrumentation and software testing[J]. Computer, 1978, 11(4).

[4] ANSI/IEEE. Standard Glossary of Software Engineering Terminology [S]. STD-729-1991. ANSI/IEEE, 1991.

[5] Jorgensen P C. Software Testing: A Craftsman's Approach[M]. Boca Raton: CRC Press, 1995.

[6] Musa J. Operational profiles in software reliability engineering[J]. IEEE Software, Mar 1993. 14-32.

[7] Eckel B. Java 编程思想(第 2 版)[M]. 候捷, 译. 北京: 机械工业出版社, 2002.

作者简介:

邱晓康 男, 1981 年生, 浙江嘉兴人, 硕士研究生, 主要研究领域为软件工程, 软件测试, 模型检验. E-mail: qiuxk@seq.nju.edu.cn.

李宣东 男, 1963 年生, 湖南邵东人, 博士, 教授, 博士生导师, 主要研究领域为软件工程, 形式化方法, 模型检验.