基于程序切片的测试用例生成系统研究与实现

王志文 黄小龙 王海军 刘 烃 俞乐晨
(西安交通大学电子与信息工程学院 西安 710049)

摘 要 介绍了一种基于程序行为切片的测试用例生成系统的实现方案,系统在不扫描全部程序路径的情况下,生成可以覆盖全部程序行为的测试用例集。系统分为静态分析、动态符号执行以及测试用例生成3个模块。在静态分析模块中根据输入的程序代码分析程序的控制流和信息流,提取程序的控制依赖和数据依赖,并计算程序的潜在依赖;动态符号执行模块求解约束条件、生成测试用例和分析代码执行过程;测试用例生成模块根据执行路径和依赖关系计算被路径覆盖的程序行为切片和未被覆盖的程序行为切片,然后根据未被覆盖的程序行为切片,引导符号执行生成能覆盖新的程序行为切片的测试用例。实验证明,本系统生成的测试用例集可以保证覆盖所有的程序行为,同时能显著减少生成的测试用例数量。

关键词 软件测试,测试用例集生成,动态符号执行,程序切片
中图法分类号 TP319 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.09.012

Program Slicing-guied Test Case Generation System

WANG Zhi-wen HUANG Xiao-long WANG Hai-jun LIU Ting YU Le-chen
(School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

Abstract A program slicing-guided test case generation system was introduced in this paper, which could generate the set of test case covering all program behavior without scanning all paths of the program. It consists of three modules: the static analysis, dynamic symbolic execution and test case generation. In the static analysis module, the control flow and information flow of input program are analyzed to extract the control dependency and data dependency, and the potential dependency is also computed. The dynamic symbol execution module is applied to solve the constraints, generate test case and monitor the execution path. In the test case generation module, the covered and uncovered program slices of the execution test scase are computed to guide the new test case generation. In the experiments, the test case set generated by our system, can cover all program behaviors and significantly reduce the number of test cases.

Keywords Software testing, Test case set generation, Dynamic symbolic execution, Program slicing

1 引言

软件测试是用来保证软件质量的基本手段,也是软件开发过程中最耗费人力物力的过程[1]。在正常情况下,程序的输入空间往往很大甚至无穷尽,测试人员不可能运行完所有的测试用例。因此选择和生成有效并具有代表性的测试用例集,就成了软件测试中最基础和最具有挑战性的问题之一。根据测试标准,传统的测试用例挑选方法主要有基于代码覆盖、分支覆盖和路径覆盖等。

代码覆盖的目标是覆盖被测代码中的可执行语句。而分支覆盖的目标是覆盖被测程序中的判定分支。虽然基于代码覆盖和分支覆盖的测试用例生成方法在工业上广泛应用,但是这两种方法在发现程序错误方面与随机用例生成方法相比并没有统计意义上的优势[2,3]。路径覆盖被认为是最强的覆盖标准之一。

基于路径覆盖的测试用例生成方法在这几年得到了大力发展,各种工具也不断出现,比如JPF-SE[4]、Concolic[5,6]KLEE[7]等等,但是这些工具都面临程序路径状态空间爆炸的问题。这些工具都是基于符号执行方法实现的。

符号执行是20世纪70年代提出的一种程序验证方法,是一种基于符号化的模型检验方法,广泛用于符号调试、测试用例生成等领域。其核心思想是使用符号值代替具体的变量输入,并使用符号表达式来表示程序中各变量的值。最终,程序的输出值被转化为一个以符号值作为输入的函数。符号执行将程序抽象为符号执行树,其中顺序语句对应着树的计算节点,分支语句对应着分支节点,而对于循环语句,将其按循环次数展开为语义上等价的分支语句。一般,一条循环语句对应一组分支节点。可以认为,在符号执行过程中,程序只有顺序和分支两种结构。符号执行的过程本质上是路径条件的构造过程。路径条件指的是对于执行该路径的测试用例,程

到稿日期:2013-12-25 返修日期:2014-01-16 本文受国家自然科学基金(91118005,91218301,61221063,61203174,61202392),国家科技支撑计划(2011BAK08B02),教育部博士点基金(20110201120010),中央高校基本科研业务费专项资金资助。
王志文(1973—),男,副教授,主要研究方向为网络安全与管理、可信网络,E-mail:wzw@mail.xjtu.edu.cn;黄小龙(1987—),男,硕士生,主要研究方向为可信软件;王海军(1983—),男,博士生;刘 烃(1981—),男,博士,讲师;俞乐晨(1990—),男,硕士生。

序输入值所需要满足的数学约束条件。因而一个路径条件唯一地对应一条执行路径。一个路径条件由一组子条件组成，每一个被执行分支的条件作为一个子条件。初始时路径条件为 true，在探索程序的过程中，每遇到一个分支语句，就更新路径条件，将被执行分支的条件加入到路径条件中。由于每一个分支语句都对应着 true 和 false 两个分支，而符号执行基于静态分析，变量没有具体的数值，因此无法确定执行哪一条分支。所以对两条分支都进行探索(搜索顺序可按需定义、深度优先、广度优先等)，即分别以两个分支的条件作为子条件来更新路径条件。这样就得到了两个新的对应于两条不同执行路径的路径条件。之后，继续对这两条路径分别进行探索。符号执行实现了对程序的全路径探索。当程序探索结束时就得到了被测程序所有执行路径的路径条件。最后，检查所有得到的路径条件，如果路径条件可以被满足，则说明该路径是一条可执行路径。将路径条件输入约束求解器即可解出对应的测试用例。

符号执行存在两个阻碍，使其难以大规模使用。

1)符号执行是一种基于搜索的遍历算法，需要对程序的所有分支进行遍历，虽然可以通过一些附加的剪枝条件进行优化，但其算法的复杂度非常高，为 $O(2^n)$，其中 n 为被测程序中条件语句(包括分支、循环、逻辑运算)的数量。

2)符号执行不能很好地解决测试用例集更新的问题，每次代码进行修改后，只能重新遍历一次符号执行树来生成一个新的测试用例集。由前面的分析可知，重新生成一个测试用例集的时间开销比较大，而且一个软件可能会频繁变更，如果每次变更之后都通过符号执行生成一个新的测试用例集，测试的效率就会受到影响。

针对符号执行方法的不足，系统根据静态依赖信息指导动态符号执行，以生成一个覆盖所有程序动态行为切片[8,9]的测试用例集，该测试用例集与全路径覆盖的测试用例集具有相同的程序行为，并且规模小得多，在一定程度上缓解了程序路径状态空间爆炸问题。

2 系统介绍

2.1 系统原理及框架

本系统实现了基于程序行为切片[10]覆盖的测试用例生成方法。程序行为切片的定义：给定一条路径 ϵ 以及路径上的一个节点 n_i，该节点 n_i 在路径 ϵ 上的程序行为切片是该路径上所有对节点 n_i 存在影响的节点。因此程序行为切片是对程序路径的一种划分，并且需要计算路径上每个节点的程序行为切片。从直观上来看，程序行为切片对路径的划分和程序语句对路径的划分一样，可以有多条包含有相同程序行为切片的路径。在这种情况下只需要生成其中的一条路径，就能够大大减少路径的数量。此外，程序行为切片覆盖能有与路径覆盖相同的程序行为，因此路径覆盖能够检测出的错误，程序行为切片覆盖也能检测出来。

本系统的分析对象是 java 源代码，主要分为 3 个模块：程序静态分析模块、动态符号执行模块和测试用例自动生成模块，系统的框架如图 1 所示。

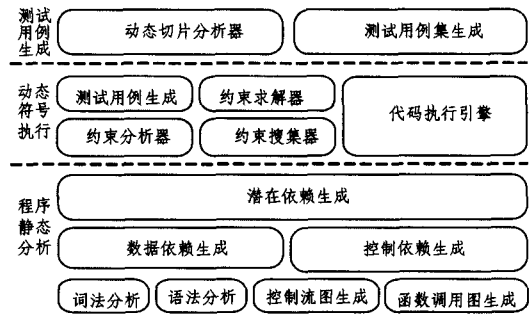

图 1 系统的框架图

2.2 程序静态分析模块

本模块主要包括词法语法分析、控制流分析、函数调用分析、控制依赖分析、数据依赖分析、潜在依赖分析这 6 个部分，如图 2 所示。

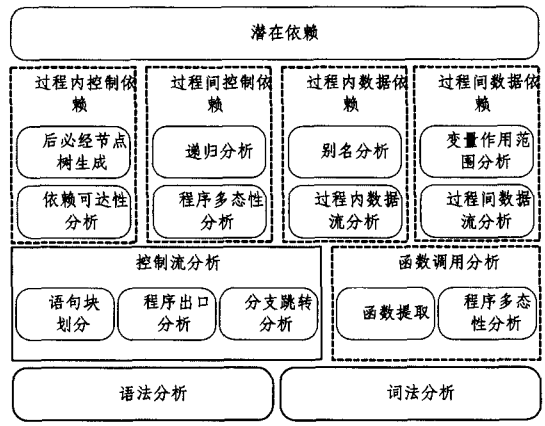

图 2 静态分析模块框架图

1)词语法分析是静态依赖分析模块的基础，对 java 源代码进行词法分析和语法分析后，就可以把程序源代码转变成抽象语法树。本系统的词法语法分析是在开源工具 polyglot[11]基础上二次开发所得。

2)控制流分析主要包括语句块划分、程序出口分析和分支跳转分析这 3 个部分。语句块划分部分把函数内语句块按照单入口、多出口的条件划分成基本语句块单元，语句块单元是控制流分析的基本单位；程序出口分析部分可以找出语句块单元中存在的函数出口，由于 java 语言的异常处理机制，函数的出口不仅仅只有 return 语句，对于声明了 throws 的函数，它可能抛出异常的每一条语句，其都是函数出口；分支跳转分析部分计算每一个语句块的分支跳转，按照跳转关系连接语句块，生成控制流图。

3)函数调用分析是在控制流图的基础上进行的，它包括函数提取和程序多态性分析两个部分，函数提取部分提取出程序中所有的函数，分析函数之间的调用关系。程序多态性分析部分主要分析 java 代码中类的继承关系和接口的实现情况，如果某一个函数调用了某一个接口中的另一个函数，则要找到函数真正实现处，尽可能找出有效的函数调用关系。

4)控制依赖[12]分析包括计算过程内和过程间的控制依赖。过程内的控制依赖计算的是函数内部的控制依赖关系，在控制流图的基础上生成后必经节点树。概括地讲，后必经节点指的是程序中某个节点到出口节点所必须经过的节点，后必经节点树指的是所有父节点都是子节点的后必经节点的

树。在后必经树上进行依赖可达性分析，分析控制信息的传递，进而生成过程内的控制依赖。过程间的控制依赖是函数之间语句的控制关系，通过对函数调用图的分析计算生成。过程间的控制依赖难点是要考虑函数递归和程序多态性。

5)数据依赖[12]分析包括计算过程内的和过程间的数据依赖，计算过程内的数据依赖时首先分析程序中变量的定义和使用关系，对于类对象或者数组，需要进行别名分析。别名分析主要针对的是引用类型，由于多个引用可以指向同一个值对象，可能导致数据依赖的不准确，别名分析把多个相同对象的引用当作一个，进而消除别名影响。利用局部数据流分析计算出过程内的数据依赖，数据流分析主要分析某个变量赋值后的影响范围，得到每条语句的可达变量定义。过程间的数据依赖主要针对类变量或者静态变量，通过全局数据流分析得到过程间的数据依赖。过程内的和过程间的数据依赖单独计算可以有效降低分析的复杂度。

6)潜在依赖[9]的计算方法是：计算一个节点 n 的所有直接数据依赖节点(n_1，n_2，n_3，…，n_i)，并且节点 n 和节点 n_i 是通过变量 v 进行数据依赖；如果数据依赖节点 n_i 的直接或间接控制依赖节点为 n_qF(n_q 节点的 F 分支)，则检查 n_qT(n_q 节点的 T 分支)的直接或者间接控制语句是否包含对变量 v 的定义，如果不包含则 n_i 潜在依赖于 n_qT；如果数据依赖节点 n_i 的直接或间接控制依赖节点为 n_qT，则检查 n_qF 的直接或者间接控制语句是否包含对变量 v 的定义，如果不包含则 n_i 潜在依赖于 n_qF。

2.3 动态符号执行模块

本模块的功能包括执行代码，获取测试用例执行路径等程序执行过程中的动态信息，并根据一个动态切片求解出执行时能覆盖该动态切片的测试用例等，框架如图 3 所示。该模块是在开源工具 JPF-SE 的基础上二次开发所得，主要包括代码执行引擎、约束搜集器、约束分析器和约束求解器 4 个部分。

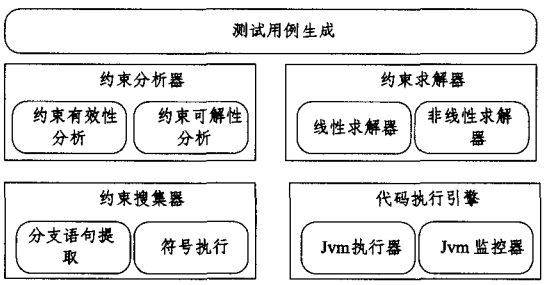

图 3 动态符号执行模块框架图

1)代码执行引擎是一个具体执行器，通过内嵌 jvm 执行 java 字节码，对 java 字节码的执行过程进行监控，获取 java 字节码执行时的动态信息，主要包括执行路径等信息。在静态分析模块的依赖计算中，有些依赖是静态无法准确计算的，会包含很多伪依赖，比如涉及到程序多态性的控制依赖和数组的数据依赖。在该引擎中，搜集程序的实际执行信息，根据实际执行信息去除伪依赖。

2)约束搜集器在代码执行引擎执行某一个测试用例时，搜集该测试用例执行过的所有条件语句的约束条件，并在给定一个未被覆盖的动态切片后，提取出该动态切片中的条件语句，计算出覆盖该动态切片的某一条路径的约束条件。

3)约束分析器主要包括约束有效性分析和约束可解性分析。约束有效性分析判断给定的约束条件是否有效，约束可解性分析判断给定的约束条件是否可解。如果碰到无效的或者无法求解的约束，就让约束搜集器根据给定的动态切片重新生成一组约束，直到生成的约束有效且可以求解。

4)约束求解器主要负责求解约束生成一个测试用例，约束求解器模块包括线性求解器和非线性求解器，对于简单的约束，线性求解器可以胜任，但对于字符串运算等复杂约束，只有非线性求解器才能求解。

2.4 测试用例自动生成模块

本模块调用上面两个模块来生成覆盖所有动态切片的测试用例集。本模块的工作流程如下：

1)根据输入约束条件，随机生成一个测试用例，利用动态符号执行器的代码执行引擎执行该测试用例，并在执行过程中获取程序执行的动态信息，包括执行路径等。

2)利用动态切片分析器计算出覆盖的和未覆盖的动态切片集；动态切片分析器主要根据控制依赖、数据依赖、组合依赖这 3 种依赖关系计算出程序静态行为切片，在静态行为切片的基础上，结合程序执行路径，计算出这条执行路径上覆盖的和未覆盖的动态切片集，如图 4 所示。

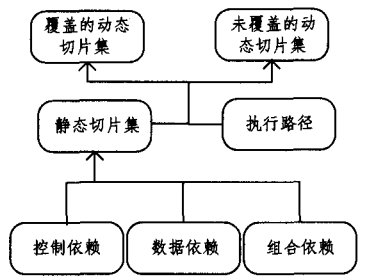

图 4 动态切片分析器

3)选取一个未覆盖的动态切片，利用动态符号执行器生成一个测试用例来覆盖该动态切片。

4)循环执行直至所有的动态切片都被覆盖，输出生成的覆盖所有动态切片的测试用例集。

3 实例分析

本部分结合具体的例子来说明系统的运行过程，图 5 是一个示例程序。

```
void Test(int x,int y,int z,int m)
1. int a=0;
2. int b=0;
3. int c=2;
4. int a=3;
5. int first_out,second_out;
6. If (x+y<0)
7.      a=1;
8. if (z>0)
9.      b=2;
10. if(x-y>1)
11.      c=a * b;
12. if (m>2)
13.      d=4;
14. first_out=c;
15. second_out=d;
```

图 5 示例程序

首先计算出程序中的控制依赖、数据依赖以及潜在依赖，结果如表1所列。表中的控制依赖(6T,7)表示第7条语句依赖于第6条语句的T分支。数据依赖(1,11,a)表示第11条语句通过变量a依赖于第一条语句。潜在依赖(6F,11,a)表示第11条语句通过变量a依赖于第6条语句的F分支。

表1 依赖关系提取

依赖类型	依赖关系
控制依赖	(6T,7)(8T,9)(10T,11)(12T,13)
数据依赖	(1,11,a)(2,11,b)(3,14,c)(4,15,d)(7,11,a)(9,11,b)(11,14,c)(13,15,d)
潜在依赖	(6F,11,a)(8F,11,b)(10F,14,c)(12F,15,d)

根据静态分析结果引导动态符号执行生成覆盖程序所有切片的测试用例集，如表2所列。对于执行路径，以路径上经过的分支语句来表示。假设某一执行路径经过了2F、3T这两个分支语句，则其表示为[2F,3T]。在程序开始执行时，随机生成一个测试用例(0,−2,1,3)，在动态符号执行引擎中获取该测试用例执行的程序路径，然后根据静态依赖信息和执行路径，计算该测试用例在执行过程中覆盖的程序行为切片和未被覆盖的程序行为切片，从未被覆盖的程序行为切片中选取一个切片，搜集要覆盖该切片的路径约束，再用约束求解器求解出一个测试用例覆盖该切片，在这个新的测试用例中计算覆盖的和未覆盖的程序行为切片，如此迭代计算，最后得出覆盖所有程序行为切片的测试用例集。对于上面的例子，只需要6个测试用例就可以覆盖所有的程序行为切片，而全路径覆盖需要生成16条路径，本系统生成的测试用例集相比于全路径覆盖小很多，一定程度上解决了路径状态空间爆炸问题。

表2 测试用例集生成过程

No	生成的程序切片	测试用例	执行路径	覆盖的程序行为切片	未被覆盖的程序行为切片
1	无	(0,−2,1,3)	[1,2,3,4,5,6T,7,8T,9,10T,11,12T,13,14,15]	[1];[1,2];[1,2,6T];[1,2,6T,7];[1,2,6T,7,8T];[1,2,6T,7,8T,9];[1,2,6T,7,8T,9,10T];[1,2,6T,7,8T,9,10T,11];[1,2,6T,7,8T,9,10T,11,14];[3];[4];[4,12T];[4,12T,13];[4,12T,13,15]	[6F];[6T,8F];[10F];[12F]
2	[6F]	(2,0,1,3)	[1,2,3,4,5,6F,8T,9,10T,11,12T,13,14,15]	[1,2,6F];[1,2,6F,8T];[1,2,6F,8T,9];[1,2,6F,8T,9,10T];[1,2,6F,8T,9,10T,11];[1,2,6F,8T,9,10T,11,14]	[6T,8F];[6F];[12F];[6F,8F]
3	[6T,8F]	(0,−2,−1,3)	[1,2,3,4,5,6T,7,8F,10T,11,12T,13,14,15]	[1,2,6T,7,8F];[1,2,6T,7,8F,10T];[1,2,6T,7,8F,10T,11];[1,2,6T,7,8F,10T,11,14]	[10F];[12F];[6F,8F]
4	[6F,8F]	(2,0,−1,3)	[1,2,3,4,5,6F,8F,10T,11,12T,13,14,15]	[1,2,6F,8F];[1,2,6F,8F,10T];[1,2,6F,8F,10T,11];[1,2,6F,8F,10T,11,14]	[10F];[12F]
5	[10F]	(0,−1,1,3)	[1,2,3,4,5,6T,7,8T,9,10F,12T,13,14,15]	[3,10F];[3,10F,14]	[12F]
6	[12F]	(0,−2,1,0)	[1,2,3,4,5,6T,7,8T,9,10T,11,12F,14,15]	[4,12F];[4,12F,15]	无

结束语 为了解决动态符号执行方法在生成测试用例集时面临的路径状态空间爆炸问题，本文设计了一个基于静态依赖信息引导动态符号执行自动生成测试用例集的系统。相比于动态符号执行，本系统生成的测试用例集大大减小。程序中的无关分支越多，系统生成的测试用例集越小，从而一定程度上解决了路径状态空间爆炸问题。

参考文献

[1] Majumdar R, Sen K. Hybrid concolic testing[C]// 29th International Conference on Software Engineering, 2007, ICSE 2007. IEEE, 2007: 416-426

[2] Bohme M. Software regression as change of input partitioning [C]// 2012 34th International Conference onSoftware Engineering(ICSE). IEEE, 2012: 1523-1526

[3] Böhme M, Oliveira B C S, Roychoudhury A. Partition-based regression verification[C]// Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013: 302-311

[4] Păsăreanu C S, Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode[C]// Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010: 179-180

[5] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[J]. ACM Sigplan Notices, 2005, 40(6): 213-223

[6] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C[M]. ACM, 2005

[7] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]// OSDI. 2008, 8: 209-224

[8] Gyimóthy T, Beszédes á, Forgács I. An efficient relevant slicing method for debugging[C]// Software Engineering—ESEC/FSE'99. Springer Berlin Heidelberg, 1999: 303-321

[9] Qi D, Nguyen H D T, Roychoudhury A. Path exploration based on symbolic output[C] // Proceeding of the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2011: 278-288

[10] Weiser M. Program slicing[C]// Proceedings of the 5th international conference on Software engineering. IEEE Press, 1981: 439-449

[11] Nystrom N, Clarkson M R, Myers A C. Polyglot: An extensible compiler framework for Java[C] // Compiler Construction. Springer Berlin Heidelberg, 2003: 138-152

[12] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1987, 9(3): 319-349