计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
47(10): 1840-1848, 2010

JUTA: 一个 Java 自动化单元测试工具

严 俊[1] 郭 涛[2] 阮 辉[1] 玄跻峰[3]
[1](中国科学院软件研究所 北京 100190)
[2](中国信息安全测评中心 北京 100085)
[3](大连理工大学数学科学学院 辽宁大连 116024)
(junyan@acm.org)

JUTA: An Automated Unit Testing Framework for Java

Yan Jun[1], Guo Tao[2], Ruan Hui[1], and Xuan Jifeng[3]
[1](*Institute of Software, Chinese Academy of Sciences, Beijing* 100190)
[2](*China Information Technology Security Evaluation Center, Beijing* 100085)
[3](*School of Mathematical Sciences, Dalian University of Technology, Dalian, Liaoning* 116024)

Abstract Testing is very important and time consuming in the development of high-quality software systems. This paper proposes an automatic testing tool JUTA for unit testing of Java programs. The approach is based on sharp analysis of the programs. JUTA firstly employs the Java optimization framework Soot to parse a single Java method into byte code and translates it into a control flow graph (CFG). It then performs depth-first or breadth-first search on the CFG to extract paths from it. Some techniques such as path length restriction are used to prevent path number explosion. Finally JUTA analyzes the paths based on the combination of symbolic execution and constraint solving. The goal of path analysis lies in two folds. It can generate a set of test cases satisfying the test criterion such as statement coverage. The test set typically has small number of test cases that are all executable. In addition to test generation for dynamic testing, it can also be used in static testing. JUTA can reveal certain kinds of errors from the source code automatically if the user provides proper assertions to describe the errors. The experimental results show that this tool is efficient for both dynamic and static testing.

Key words Java unit testing; dynamic testing; static testing; program analysis; symbolic execution

摘 要 描述了一个 Java 自动化的单元测试工具 JUTA. JUTA 首先调用工具 Soot 解析单个 Java 方法的源码,并将源码解析成一个控制流图. 在此基础上,采用符号执行的方法分析控制流图上的路径. 工具能够自动地产生满足覆盖率标准的程序的测试用例. 这种方法产生的所有测试用例都是可执行的,并且一般来说具有较小的测试用例数. 如果用户能够合理地给出描述程序错误的断言,框架 JUTA 能够自动地检查源码中部分特定类型的错误. 实验结果表明工具对 Java 单元代码的动态测试和静态测试均能在可接受的时间内给出有效的结果.

关键词 Java 单元测试;动态测试;静态测试;程序分析;符号执行

中图法分类号 TP311

收稿日期:2008-11-13;**修回日期**:2010-02-04

基金项目:国家自然科学基金项目(60903049);国家"八六三"高技术研究发展计划基金项目(2009AA01Z148,2009AA010313);中国科学院知识创新工程领域前沿项目(ISCAS2009-DR01)

0　引　　言

软件质量历来就是学术界和工业界共同关心的问题,因为软件质量问题造成巨大经济损失事件不胜枚举.当前,随着信息技术的迅速发展,特别是互联网技术的广泛应用,Java 技术以其通用性、平台移植性、安全性等特征,吸引了众多软件开发者,在各个重要的行业部门得到了广泛的应用.因此,如何提高软件质量,尤其是提高应用广泛的 Java 程序的可靠性,长期以来一直是一个非常重要的研究课题.

软件测试是提高软件质量的重要手段.优秀的软件开发机构把 40%的工作量花在软件测试上,软件测试费用则占到了软件开发总费用的 30%～50%,而对于一些要求高可靠性、高安全性的软件,测试费用已经抬高到了整个软件项目开发所需费用的 3～5 倍.

单元测试是软件测试的早期阶段,这种测试方法主要测试程序的单元函数或者类中的方法,其主要原理是对源代码中的控制结构和处理过程等进行分析,检查程序内部处理是否正确,包括语句结构、分支和循环结构等.由于单元测试需要分析源代码,所以测试的代价往往比较高.目前,在工业界,单元测试大多仍然是靠人工完成的,如手工设计测试用例、人工审阅代码等,不但十分繁琐,而且测试质量与测试人员的主观经验紧密相关.随着计算机软硬件规模的不断发展,以人工为主的测试方法已经无法满足测试的需求.目前,采用计算机辅助的自动化软件测试技术成为软件工程中一个活跃的研究领域.

本文介绍一个 Java 自动化的单元测试工具 JUTA(an unit testing and analyzing tool for Java).与 Junit 等传统测试框架不同,JUTA 提供了一系列的精度和自动化程度较高的技术来生成测试用例以及检查代码中的错误,以提高测试的自动化程度和效率.

1　预备知识

本节将简要介绍 JUTA 采用的控制流和数据流分析技术.

1.1　控制流图

形式化地说,控制流图(control flow graph,CFG)是一个有向图 $G=\langle N,E,s,f\rangle$,其中 N 代表节点的集合,E 代表边的集合,s 和 f 分别表示控制流图的起点和终点.图中每个点 $n\in N$ 代表程序中的一系列顺序运算(即这些语句在程序执行过程中要么全部都执行,要么全部都不执行),或者说是一个基本块(basic block);而每条边,$e=(n_i,n_j)\in E$,代表从节点 n_i 到 n_j 的一个转移.用流图作为程序结构的模型时,程序的运行过程可以用流图中的路径来刻画.控制流图中的一条执行路径 p 代表一个执行序列 $p=\langle n_1,n_2,\cdots,n_m\rangle$,其中 m 是路径的长度,$(n_i,n_{i+1})\in E(1\leqslant i<m)$.一般来说,程序的执行必须从起点 s 开始.我们称 $n_1=s$ 的这条路径为部分路径(partial path).如果 $n_1=s$ 并且 $n_m=f$,我们称 p 是程序的一条完整路径(complete path).在下文中,如无特殊说明,我们所说的程序路径都是完整路径.

例 1. 图 1 为一段简单的代码及其控制流图.函数 *foo* 根据输入 i 和 j 的值计算参数 *good* 的值并返回.从控制流图上易知程序有 3 条从起点 S 到终点 F 的完整路径:$p_1=\langle S,S_1,S_2,S_3,S_4,F\rangle$,$p_2=\langle S,S_1,S_2,S_3,F\rangle$,$p_3=\langle S,S_1,F\rangle$.

```
    public int foo(int i,int j){
/*S*/   int good=TRUE;
/*S1*/  if((i>2) && (j>3)){
/*S2*/    j--;
/*S3*/    if(i+2*j<5)
/*S4*/      good=FALSE;
        }
/*F*/   return good;
      }
```

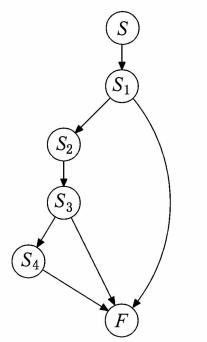


Fig. 1　A piece of example code.

图 1　一段示例代码

对于基于控制流图的结构测试来说,主要的测试方法就是找到一系列的可行测试路径,这些路径的输入数据就是测试用例.例如在例 1 中,$\{i=3,j=4\}$就是对应于路径 p_2 的一个测试用例.测试人员通过执行这些测试用例来检查程序的错误.显然,由于程序代码的复杂性,通过枚举所有的测试路径来测试程序是不现实的.为了减少测试的代价,人们定义了一系列的测试覆盖准则(coverage criterion),作为测试的标准来限制测试集的大小[1].最简单的控制流覆盖准是语句覆盖(statement coverage,即在软件测试过程中,只有当程序中的所有语句都得到了运行,才能称该测试是充分的)和分支覆盖(branch coverage,即要求在软件测试过程中,所有

的控制转移都得到了覆盖). 由于在控制流图模型中,语句与节点相对应,语句覆盖意味着测试路径覆盖所有的节点. 同样地,分支覆盖就是所有的测试路径覆盖了控制流图中所有的边.

1.2 基于路径的程序测试

程序中的一次完整的执行过程对应控制流图中的一条完整路径. 但是,并不是所有的完整路径都代表程序中的一次运行过程. 这是因为可能不存在输入数据使得程序按照该路径执行. 这样的完整路径称为不可行路径(infeasible path),否则称为可行的(feasible)[2]. 例如,在例 1 中,路径 p_1 执行的语句如图 2 所示(符号"@"后的表达式表示程序执行该逻辑判断为真的分支). 易知,程序执行到第 2 个判断语句时,表达式($i+2\times j<5$)不可能满足,程序控制流图中的分支 S_3-S_4 不可能被执行到. 那么,路径 p_1 是一条不可行的路径.

```
  good=TRUE;
@((i>2) && (j>3));
  j=j-1;
@(i+2×j<5);
  good=FALSE;
```

Fig. 2 Path p_1.

图 2 路径 p_1

由于控制流测试准则都是基于控制流图中的测试路径来定义,很自然地,我们可以采用如下方法来生成测试用例:首先从控制流图中找到一组路径满足覆盖准则,然后从每条路径中找出各变量的一组初始值使得程序能够按照路径执行,这些初值就构成了测试用例集合. 一般来说,一个自动测试用例生成(automated test case generation)系统主要可以分成 3 个部分[3]:

1. 程序分析器(program analyzer),其主要工作是把程序转化成容易处理的中间模型(一般就是我们前面提到过的控制流图);

2. 路径选择器(path selector),如何选择路径满足我们的测试需求(比如覆盖准则);

3. 测试数据生成器(test data generator),如何从测试路径中得到我们需要的测试数据.

在实际开发中,程序分析器也被称为前端(frontend),后两部分合起来被称为后端(backend).

这样的自动测试生成的系统主要用于程序的动态测试,即采用测试用例测试被测程序. 与动态测试对应的静态测试方法,其框架与测试生成大体相同. 主要的不同指出在于其输出并不是测试用例,而是直接报告程序是否包含某类错误,或者给出能检出某类错误的具体的测试数据.

由于程序测试的需求众多,实际的测试系统尤其是商用系统,可能还包括一些其他的辅助模块,比如测试脚本的自动生成器等.

1.3 路径分析方法

不可行路径给自动化测试带来了一个难题,我们不可能简单地通过从程序控制流图的信息中抽取路径来选择测试用例,在测试用例选择时必须分析数据流的信息. 为了精确地分析控制流图中一条完整的路径,我们可以采用符号执行(symbolic execution)[4]的方法. 所谓符号执行,相对于普通的程序执行来说,就是程序并不真正用数值来替换程序的变量,而是用一组关于初始值的表达是来表示程序中出现的项. 用符号执行加约束求解进行程序分析的基本思想是:采用 Hoare 逻辑可以将程序路径表示成$\{P\}Q\{R\}$,其中 P 是程序的前置条件(pre-condition. 执行程序前需要满足的条件,或者称为前断言),R 是程序的后置条件(post-condition,程序执行后需要满足的条件,或者称为后断言). 假定在程序的符号执行过程中由$\{P\}Q$可以推导出约束条件 $c_1\wedge c_2\wedge\cdots\wedge c_n$,则应该有 $c_1\wedge c_2\wedge\cdots\wedge c_n\rightarrow R$. 在一般的测试中,如果后断言永真,那么可以对约束条件 $c_1\wedge c_2\wedge\cdots\wedge c_n$ 求解,可以得到变量的初值,从而得到测试用例. 另一方面,我们可以对约束 $c_1\wedge c_2\wedge\cdots\wedge c_n\wedge\neg R$ 求解,如果有一组解满足这一约束,说明存在一组输入使运行程序的结果和规范不符. 如果程序的规范正确,则程序中必定包含错误.

对于不包含函数调用代码来说,程序的语句可以分成 3 类:1)动作(action),也就是赋值语句;2)判断语句(predicate),也就是一个逻辑表达式;3)其他不影响变量的表达式,如控制台打印语句 System. out. println. 我们用一个具体的例子说明符号执行的原理. 考虑图 2 的路径 p_1,如果 i 和 j 的初值为 i_0 和 j_0,我们可以自顶向下依次执行这段程序,用程序的赋值语句替换判断语句,最后得到一个只含有判断语句的约束集合$\{i_0+2\times(j_0-1)<5;(i_0>2)\ \&\&\ (j_0>3)\}$. 这个集合就是程序初值的约束. 由于这个约束集是不可满足的,所以路径 p_1 是不可行的. 关于符号执行的具体细节请参考文献[2,5-6].

符号执行的好处是能够精确地分析程序的行为,但它的缺点也是显而易见的. 由于符号执行方法过于依赖源代码,并且需要消耗较大的计算资源,因

而无法精确分析带有很多模块调用(特别是当这些子模块代码不可见的时候)的程序.同时,程序需要能够用约束精确模拟复杂程序结构,比如指针和数组的执行,这是一件很困难的事情.最后,为了保证分析的精确性,符号执行需要有一个能力非常强的完备解法器(complete solver,就是说,只要约束有解,解法器必然能够给出一组解).针对不同的程序,这个解法器可能需要处理位操作等复杂的约束.这一点也限制了符号执行方法的使用.工具 JUTA 中采用较为简单的方式在工具的处理能力和精确度之前作了一个平衡.我们将程序中的表达式限制为布尔逻辑和线性数值约束两种类型,并且将程序的外部调用抽象成简单的赋值语句和后置断言.具体内容见本文第 2 节.

2　工具 JUTA 的设计与实现

本节将详细介绍 JUTA 工具的流程以及工具的模块组成,同时对每一个模块给出其功能以及简要的描述,主要目的是为工具提供一个整体的框架描述.

2.1　工具流程

为了便于理解源程序需要对程序的源代码进行一定的处理.首先要做的就是编译分析,包括语法语义错误的检查、建立中间表示(intermediate representation,IR)并生成控制流图等.同时为了以后的分析,我们还需要识别出源代码中的循环,并对循环进行处理,建立一棵循环控制树.这部分通常成为前端处理.有了前端处理的程序结构和数据信息,我们就可以通过算法遍历控制流图,得到一系列的路径,并通过符号执行和约束求解来分析这些路径,对其可行性进行判断,对于可行的路径为其生成测试数据,最后收集这些测试数据就得到了满足某些性质的目标测试数据集.因而,从整体上工具分为两个部分:前端处理和后端分析,其整体结构如图 3 所示:

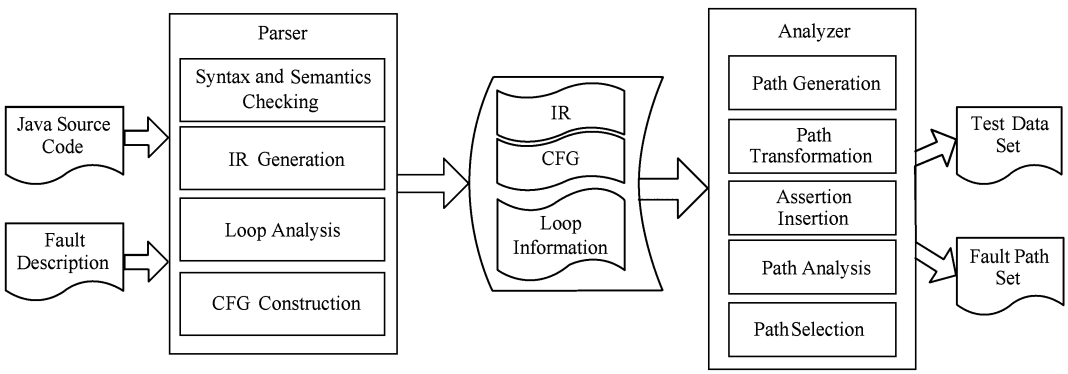


Fig. 3　Framework of JUTA.

图 3　JUTA 的框架

目前,工具 JUTA 实现了两方面的功能:静态测试与动态测试.本文 1.2 节曾指出,这两种测试方法可以公用相同的前端,主要区别在于工具的后端.接下来我们将按照功能分别介绍工具的各模块.

2.2　工具输入

工具输入包含两部分.首先是被测的 Java 源码.源码中包含被测 Java 代码的单个方法.由于工具 JUTA 的主要功能是完成单元测试,所以,工具将丢弃方法中的外部调用.用户在测试时,需要手工将重要的外部调用替换成赋值语句或者约束表达式.例如,对于语句

$$y = foo(x),$$

假定外部调用 foo 将 y 的值改变为 foo_x,并且这个值的范围是$[lb,ub]$.那么可以将这行代码替换为:

$y = foo_x$;

$assert(lb <= foo_x \&\& foo_x <= ub)$;

另一部分的输入是一些描述性质的断言以及断言的插入位置.具体请参见本文 2.4.2 节.

2.3　前端处理

前端分析整体上相当于一个编译器的前端,功能上相当于一个不产生目标代码的编译器.在这一部分我们借用的是 Java 分析优化工具框架 Soot[7] 的编译分析功能.

Soot 是一个功能很强大的分析框架,融合了编译前端以及分析器的各种功能.工具 JUTA 的前端在 Soot 框架下扩展其功能,产生后端分析所需要的中间结果.在 JUTA 中,我们主要使用了 Soot 的语法语义检查、中间表示、循环控制分析以及控制流图

生成等模块. 接下来简要介绍这几个模块的功能.

2.3.1 语法语义检查模块

该模块主要完成源代码语法语义的检查,确保给定的输入程序是没有语法语义错误的,同时建立一棵等价于源程序的抽象语法树.

2.3.2 中间代码生成模块

这部分通过遍历抽象语法树,对各种语句进行处理,产生中间表示,便于以后的分析和处理. Soot 提供了 4 种中间表示形式:Baf,Jimple,Shimple 和 Grimp. 工具 JUTA 使用了 Soot 的 Jimple 中间表示. Jimple 是一种简单的带类型的 3 地址表示形式. 例如图 4 为 Java 源码与对应的 Jimple 代码.

图 4(a)所示的一段简单的 Java 代码,经过 Soot 翻译后得到图 4(b)所示:

```
public static int func(int x){
  int y=-1;
  if(x>0)
    y=1;
  else
    y=2;
  return y;
}
```

(a)

```
public static int func(int){
  int x,y,temp$0,temp$1;
  x:=@parameter0:int;
  y=-1;
  if x>0 goto label0;
  goto label1;
label0:nop;
       temp$0=1;
       y=temp$0;
       goto label2;
label1:nop;
       temp$1=2;
       y=temp$1;
label2:nop;
       return y;
}
```

(b)

Fig. 4 Java source code and the corresponding Jimple code. (a) Java source and (b) Jimple code.

图 4 Java 源码与对应的 Jimple 代码. (a) Java 源码; (b) Jimple 代码

2.3.3 控制流图建立模块

本模块通过遍历中间表示的列表,将中间表示序列划分为基本块,控制流图即是以基本块为节点的有向图,描述了源代码中的控制流. 工具 JUTA 使用 Soot 中的 Brief Block Graph,只为源代码中基本的控制流提供支持,目前无法处理异常(exception).

2.3.4 循环控制分析模块

这一部分主要是分析控制流中的循环,Soot 中将每一个循环抽象成一个对象,每一个循环都存储自己的循环头部以及循环体. 一般来说,结构化的程序中的循环分成 3 类,孤立循环(single loop)、嵌套循环(nesting loop)以及串联的循环(concatenated loop)[8]. 我们可以建立一个抽象的循环控制森林,森林中每一个节点是一个表示循环的对象,用来描述循环之间的关系. 循环 A 是循环 B 的子节点,当且仅当循环 B 内嵌在循环 A 之中. 当被测程序中不包含串联循环时,循环控制森林退化为循环控制树.

2.4 后端分析

本节将介绍工具的后端. 考虑到动静态测试的不同之处,我们将分别介绍动态和静态测试的功能模块,之后对两者进行一个比较.

2.4.1 动态测试后端

动态测试的目标是生成一组满足覆盖率的测试数据. 所以,动态测试的后端需要分析前端处理后得到的数据信息(包括中间表示、控制流图以及循环控制树),根据路径产生策略以及覆盖准则产生满足要求的路径,并且通过分析路径的可行性得到测试数据. 这一部分有以下 4 个模块:路径生成、路径转换、路径分析以及路径选择.

1) 路径生成模块

这一模块将分析由前端处理得到的控制流图,并抽取出一系列的路径. 这些路径可以用基本块的编号序列表示. 路径的产生过程是通过遍历控制流图来得到的. 由于程序中可能有循环,那么,从图中可能抽取出无限条路径. 为了产生有限条测试路径,我们可以限制循环次数或者限制路径的长度(路径表示中基本块的个数). 在工具 JUTA 中可以通过参数来选择使用的路径限制策略.

路径产生是一个遍历控制流图各顶点的过程,在图的遍历算法中有两种常见的策略:深度优先遍历(DFS)和宽度优先遍历(BFS). 当采用 BFS 算法时,是按照路径长度由短到长的次序生成路径,所以在队列中需要存储最多两层的中间节点,当路径长度限制比较大或者循环次数限制值比较大时,队列中需要存储很多的节点. 在使用 DFS 算法时,由于优先考虑循环的退出节点,在栈中存储的元素数目比较少,但是生成的最终路径不是按照由短到长的顺序. 而且很多路径都是由几个相同的基本块所组成. 由上述的比较可知,使用 DFS 占用较少的系统资源,但是在处理过程中,尤其是在 CFG 包含环时容易陷入一个分支无法跳出,无法快速达到覆盖率指标,比较适用于静态测试;使得 BFS 能够较快达到覆盖率,但是代价相对较高,适用于在测试覆盖准则指导下的测试生成. 工具 JUTA 的路径生成模块实现了这

两种策略,用户也可以用参数指定所使用的遍历策略.在默认情况下使用的是深度优先.

2) 路径转换模块

前面 1.3 节中介绍过,对于不包含函数调用代码来说,程序的语句可以分成 3 类.而我们主要关注其中两类语句,即赋值和判断.路径转换模块将采用基本块的编号序列表示的路径,根据基本块对应的 Jimple 中间表示,翻译成一些简单的赋值和判断语句,方便下一步的路径分析.

3) 路径分析模块

这一部分主要是用来对产生的路径进行分析.张健等人在文献[2,5-6]中提出了基于符号执行以及约束求解的路径分析方法,并给出了一个工具 EPAT 用于判断 C 程序路径的可行性.目前,EPAT 支持布尔逻辑表达式和线性数值约束表达式及其混合,同时还支持数组和指针下标的运算.

考虑到经过断言插入和路径转换后的路径只包含两类基本的表达式:判断和赋值,从而可以调用 EPAT 来分析 Java 程序的路径条件,判断出路径是否可行,并对可行的路径产生测试数据.

4) 路径选择模块

为了采用较少的测试数据测试程序,可以在所有可行路径中选择一些有代表性的路径来测试被测代码.可行的选择标准就是覆盖准则,例如语句覆盖.在这种测试方法中,我们力图用最少的测试数据去覆盖尽可能多的语句或者是基本块.这个问题实际上是一个测试集约简(test reduction)问题.Wang 等人指出,这种以覆盖路径中的点或者边为目标的测试约简问题可以采用线性规划方法求解[9].具体到这个问题,由于路径生成模块可能产生的路径数目太多,一般的线性规划工具无法处理这么大规模的问题,工具 JUTA 通过以下 3 个步骤来约简测试集.

① 不可行路径删除.首先要做的就是删掉不可行路径.这一步在我们的前几个模块已经完成,所以可以不用考虑.

② 删除冗余路径.采用贪心算法来减少可行路径集合的大小,将那些对覆盖率没有贡献的测试路径予以删除.

③ 寻找最优路径集合.可以使用线性规划工具进一步删除冗余的路径,得到一个最小的路径集合.

实验结果表明,这个路径约简方法能够得到接近最优的测试集合(参见 3.1 节).

2.4.2 静态测试后端

静态测试的目标是给出程序是否包含某类错误的一个判断.在原理上,工具 JUTA 的静态测试与动态测试都是采用路径分析方法,所以静态测试后端重用了动态测试后端的 3 个模块,即路径生成、路径转换、路径分析.在本文 1.3 节中指出,可以通过简单的断言来描述程序的错误性质,从而自动地、静态地检查程序的错误.为了便于用于描述程序错误性质,工具在静态测试的后端中的路径分析模块之前增加一个模块,用于在路径中插入断言表达式.当此模块功能打开时,工具 JUTA 的输出就是满足这些条件的测试路径集合.

在工具 JUTA 中,断言插入模块读入条件表达式,并将这些条件插入到路径的相应位置.目前,JUTA 支持在指定的基本块前后加入前置或者后置条件.断言的语法形式是

BBID expr Position Mode,

其中:

①*BBID* 表示基本块的编号;

②*expr* 表示描述错误的性质的条件表达式,目前支持线性数值约束与布尔逻辑约束的混合;

③*Position* 表示插入断言的位置有两个选项:pre 和 post,分别表示插入到基本块之前或者之后;

④*Mode* 表示断言插入的模式,有 4 种选项,F 表示在路径中第 1 次出现该基本块时插入;L 表示在路径中最后一次出现该基本块时插入,A 表示在所有的基本块的对应位置插入该断言;而当这一项为一个数值 i 时,表示在该基本块第 i 次出现的位置插入断言.

例 2. 假定为了测试如下循环程序是否有数组下标越界的错误.

```
/* block 14 */   for(int i=0;i<n;i++){
/* block 15 */       read(a[2*i]);
                        ⋮
                   }
```

我们在基本块 15 中发现被测程序对数组 a[] 下标进行了运算,则可以在基本块 15 之后加入如下条件表达式(假定数组大小 20),描述下标越界:

15 (2*i<0 || 2*i>=20) post L,

然后让工具枚举从控制流图起点到基本块 15 的部分路径并检查可行性.这样,如果 *JUTA* 输出一条可行的部分路径,即存在一条可以执行的输入使得描述错误的断言可满足,则表示程序在基本块 15 中可能含有数组下标越界的错误.

2.4.3 动静态测试后端的不同之处

动态测试需要分析完整路径并给出测试数据，而静态测试并不需要给出完整的路径，只需找到包含错误的部分路径即可. 因而在静态测试中，路径生成模块给出的路径是以可能出错的基本块为终点的部分路径. 另外，静态测试需要一个断言插入模块插入用户定义的断言以检查程序中的错误，而动态测试需要一个路径选择的模块来约简测试集. 表 1 比较了两个后端的不同之处：

Table 1 Comparison Between the Backends of Static and Dynamic Testing

表 1 静态测试与动态测试后端比较

Module	Static Testing	Dynamic Testing
Path Generation	Partial Path	Complete Path
Assertion Insertion	Insert User Written Assertions	—
Path Selection	—	Test set reduction

3 实例研究

本文我们将通过部分实例来展示工具的功能.

3.1 动态测试

动态测试的目标是产生测试用例，并达到测试覆盖率. 我们选择了 6 个方法实例，设置每个循环限制的循环次数上限是 3 次，覆盖准则是语句覆盖，得到一个可行路径集合 F. 然后通过路径选择模块约简得到一个最优测试集合 O. 表 2 列出了实验结果，所有实例的运行时间均在 1 min 以内. 这里，圈复杂度(cyclomatic complexity)[10]描述了程序的逻辑复杂度，其定义为

$$v(G) = e - n + 2 = d + 1 ,$$

其中，e 为控制流图中边的数目，n 为节点(基本块)的数目，而 d 为控制流图中判断节点(condition node，假定程序中所有的判断节点均为二叉节点)的个数. 一般来说，满足语句覆盖的最后测试集的大小不超过代码的圈复杂度[1].

Table 2 Unit Test Generation

表 2 单元测试用例生成

Function	$v(G)$	$\|F\|$	$\|O\|$
triDiv	6	11	3
reminder	2	5	1
bubbleSort	4	33	2
binarySearch	3	5	1
insertSort	4	9	1
gcd	4	4	1

3.2 静态测试

静态测试的目标是给出程序是否含有某类错误的判断. 我们在冒泡排序程序中引入了一个错误，如图 5 所示：

```
public static int[] bubbleSortBad(int [] arr,int size){
    int temp;
    for(int i=0;i<size-1;i ++)
      for(int j=0;j<=size-i-1;j ++)
      /* 错误，"<="应该为"<" */
        if(arr[j+1]<arr[j]){
          /* block 4 */
          temp=arr[j];
          arr[j]=arr[j+1];
          arr[j+1]=temp;
        }
      return arr;
    }
```

Fig. 5 Faulty code of buble sort.

图 5 冒泡排序的错误代码

为了检测程序是否有数组下标溢出的错误，考虑到出现的数组下标有 j 和 $j+1$，所以我们插入如下断言到基本块 4(图 5 中黑斜体所示部分)：

4 ($j<0 \| j>=size \| j+1<0 \| j+1>=size$)post L.

设置循环次数上限为 3 次，工具 JUTA 在 4 s 内发现了 4 条以基本块 4 为终点的部分路径，其测试数据如表 3 所示. 易知，这 4 条路径对应的数组长度均为 2，但是路径执行过程中访问了 $arr[2]$，发生了数组越界错误.

Table 3 Test Data for Array Index Out of Bounds

表 3 检出数组下标溢出的测试数据

Test Path	Test Data
1	$size=2, arr[0]=1, arr[1]=0, arr[2]=0$
2	$size=2, arr[0]=1, arr[1]=0, arr[2]=1$
3	$size=2, arr[0]=0, arr[1]=1, arr[2]=0$
4	$size=2, arr[0]=0, arr[1]=0, arr[2]=0$

4 相关工作

目前，知名度最高的 Java 单元测试工具是由 Gamma 和 Beck 编写的回归测试框架 JUnit[11]. JUnit 测试是程序员测试，即需要程序员知道被测试的软件如何完成功能和完成什么样的功能，在此基础上，由程序员设计测试用例完成测试. 类似的测试框架还有 Cactus[12]，一个基于 JUnit 框架的简单测试框架，用来单元测试服务端 Java 代码；Abbot[13]

是一个用来测试 Java GUIs 的框架.与我们的工具 JUTA 不同,这些框架提供了一些方便的功能,例如,自动生成测试脚本,但是不包含自动生成测试用例以及自动检查程序错误的功能.

学术界对于 Java 单元测试进行了多年的研究.在自动测试用例生成方面,Cute(concolic unit testing engine for C and Java)[14]采用符号执行(symbolic execution)与具体执行(concrete execution)相结合的方法(即所谓的 concolic 方法),用于自动测试 Java 代码.值得一提的是 Cute 支持指针和数组,但是它无法处理数组下标表达式的运算.类似地采用符号执行技术的 Java 自动测试工具还有 Symstra[15]等.

在静态分析方面,学术界也开发了部分针对 Java 的静态分析工具.ESC/Java[16]是一个 Compaq(现在为 HP 的一部分)开发的程序设计工具,尝试通过这个程序原文的静态分析来发现在 Java 中的普遍运行错误.ESC/Java2[17]是 ESC/Java 的扩展,程序员通过给方法添加前置断言、后置断言和循环不变量等 JML 注释,使用定理证明器 Simplify,对 Java 程序的规范进行形式化验证.如果不添加任何注释,ESC/Java2 也能查找空指针、数组越界等错误,并且可以通过添加注释移除误报.Findbugs[18]是一个轻量级静态分析工具,它检查类或者 JAR 文件,将字节码通过词法分析,与一组缺陷模式进行对比以发现可能的问题.Findbugs 同时也使用了数据流分析技术,它主要能检测空引用、同步错误、输入输出流、字符串相关、冗余变量等错误.正如文献[19]指出,这些静态分析工具一般精度不高,其好处是效率很高,能处理大程序;但其问题是分析结果的可信度差,可能会有很多的误报和漏报.所谓"误报"就是分析工具报告的错误实际上不存在;所谓"漏报"就是程序中某些错误没有被分析工具发现.工具 JUTA 中的静态分析方法是文献[19]提出的精确静态分析技术在 Java 代码分析上的实际应用,其目标是针对小规模的程序代码作较为精确的分析,在一定程度上减少误报和漏报率.

5 结论与未来的工作

本文描述了一个 Java 自动化单元测试框架 JUTA.JUTA 对于一个给定的 Java 方法能够自动地扫描代码,产生满足覆盖准则的、较小的测试数据集合.同时,用户能够方便地在被测代码中插入一些表达式,从而生成满足表达式的测试用例.这样,对于能够用简单条件表达式描述的程序错误,工具也能够自动的发现该类错误.实验结果表明,该工具能够在可接受的时间内给出符合测试准则的测试用例.如果用户能给出错误描述的断言,工具能够自动地找出程序中的错误.

目前,框架 JUTA 仅仅实现了 Java 语言一些自动化单元测试一些基本的功能.下一步的工作主要是对 JUTA 的一些功能进行扩展.例如,目前仅仅支持对单个方法进行测试,未来将会扩展到对 Java 的单个类进行测试,这样,工具需要对方法调用进行精确和自动化的建模.另外,工具经过适当的扩展可以支持更多的测试准则(动态测试)以及更加灵活的错误性质描述(静态测试).除此之外,还需要增加一些与测试相关的功能,例如测试脚本自动生成、错误定位等.

参 考 文 献

[1] Zhu Hong, Hall P A V, May J H R. Software unit test coverage and adequacy [J]. ACM Computer Survey, 1997, 29(4): 366-427

[2] Zhang Jian, Wang Xiaoxu. A constraint solver and its application to path feasibility analysis [J]. Int Journal of Software Engineering and Knowledge Engineering, 2001, 11(2): 139-156

[3] Edvardsson J. A survey on automatic test data generation [C] //Proc of the 2nd Conf on Computer Science and Engineering in Linköping. Linköping: ECSEL, 1999: 21-28

[4] King J C. Symbolic execution and program testing [J]. Communications of the ACM, 1976, 19(7): 385-394

[5] Zhang Jian. Specification analysis and test data generation by solving Boolean combinations of numeric constraints [C] // Proc of the 1st Asia-Pacific Conf on Quality Software (APAQS). Piscataway, NJ: IEEE, 2000: 267-274

[6] Zhang Jian. Symbolic execution of program paths involving pointer and structure variables [C] //Proc of the 4th Int Conf on Quality Software (QSIC). Piscataway, NJ: IEEE, 2004: 87-92

[7] Sable Research Group. Soot: A Java optimization framework [CP/OL]. [2010-01-01]. http://www.sable.mcgill.ca/soot/tutorial/index.html

[8] Beizer B. Software testing techniques [M]. New York: John Wiley & Sons, 1989

[9] Wang H S, Hsu S R, Lin J C. A generalized optimal path-selection model for structural program testing [J]. The Journal of Systems and Software, 1989, 10(1): 55-63

[10] McCabe T J. A complexity measure [J]. IEEE Trans on Software Engineering, 1976, SE-2(4): 308-320

[11] Beck K, Gamma E. JUnit [CP/OL]. [2010-01-01]. http://www.junit.org/

[12] Apache Software Foundation. Cactus [CP/OL]. [2010-01-01]. http://jakarta.apache.org/cactus/

[13] Wall T. Abbot [CP/OL]. [2010-01-01]. http://abbot.sourceforge.net/

[14] Sen K, Agha G. CUTE, jCUTE: Concolic unit testing and explicit path model-checking tools [C] //LNCS 4144: Proc of the 18th Int Conf on Computer Aided Verification (CAV). Berlin: Springer, 2006: 419-423

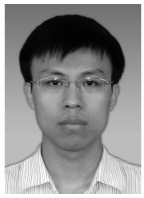
[15] Xie T, Marinov D, Schulte W, et al. Symstra: A framework for generating object-oriented unit tests using symbolic execution [C] //LNCS 3440: Proc of the 11th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Berlin: Springer, 2005: 365-381

[16] Flanagan C, Rustan K, Leino M, et al. Extended static checking for Java [C] //Proc of ACM SIGPLAN Conf on Programming Language Design and Implementation (PLDI). New York: ACM, 2002: 234-245

[17] Cok D R, Kiniry J. ESC/Java2: Uniting ESC/Java and JML [C] //LNCS 3362: Proc of Int Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS). Berlin: Springer, 2004: 108-128

[18] Pugh B, Hovemeyer D. Findbugs [CP/OL]. [2010-01-01]. http://findbugs.sourceforge.net/

[19] Zhang Jian. Sharp static analysis of programs [J]. Chinese Journal of Computers, 2008, 31(9): 1549-1553 (in Chinese)
(张健. 精确的程序静态分析[J]. 计算机学报, 2008, 31(9): 1549-1553)



Yan Jun, born in 1980. PhD and associate professor. Member of China Computer Federation. His main research interests include constraint processing, program analysis and software testing.

严 俊, 1980 年生, 博士, 副研究员, 中国计算机学会会员, 主要研究方向为约束求解、程序分析与软件测试.



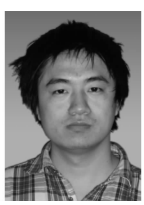
Guo Tao, born in 1974. PhD and associate professor. His main research interests include information security vulnerability analysis.

郭 涛, 1974 年生, 博士, 副研究员, 主要研究方向为信息安全漏洞分析.



Ruan Hui, born in 1984. Master of Science. His main research interests include program analysis and automatic software testing.

阮 辉, 1984 年生, 硕士, 主要研究方向为程序分析与软件测试自动化.



Xuan Jifeng, born in 1984. PhD candidate. His main research interests include intelligent computing and automatic software testing.

玄跻峰, 1984 年生, 博士研究生, 主要研究方向为智能计算与软件测试.

Research Background

Program correctness is one of the most important issues for computer scientists and engineers. For the past ten years, we have been working on the automatic test data generation for programs, and on finding bugs in programs through static checking.

Building on our previous works, we propose an automatic testing tool for Java unit testing in this paper. This tool generates a set of paths from the flow graph of a Java unit program. Then, based on the symbolic execution technique, it computes a set of test data that satisfies the statement or branch coverage criteria from the path set, or analyzes the program's behavior with user-written assertions. Compared with other testing tool on Java programs, our method is more accurate and thus more automatic to some extent. Our experimental results show that the method is quite accurate and efficient.

This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 60903049, the National High-Tech (863) program with project No. 2009AA01Z148 and 2009AA010313, and the CAS Innovation Program under grant No. ISCAS2009-DR01.