第33卷 第21期 计 算 机 工 程 2007年11月
Vol.33 No.21 Computer Engineering November 2007
·软件技术与数据库· 文章编号：1000—3428(2007)21—0078—03 文献标识码：A 中图分类号：TP311

# 基于JPF的类数据流测试生成技术

唐春艳，钟 诚
(广西大学计算机与电子信息学院，南宁 530004)

**摘 要：**通过设置陷阱性质，用时序逻辑公式表示数据流测试的覆盖准则，将测试生成问题简化成模型检测中寻找反例的问题，自动生成满足数据流覆盖准则的类方法测试调用序列，提出了一种适用于类方法调用序列自动生成的搜索算法，并在程序模型检测器JPF上实现。算法分析和实验结果表明，该算法能生成高效的方法调用序列并明显减少测试生成代价。

**关键词：**程序模型检测；数据流测试；类测试序列生成；Java

# Data-flow Test Generation for Classes with Java PathFinder

TANG Chun-yan, ZHONG Cheng
(School of Computer and Electronics and Information, Guangxi University, Nanning 530004)

【**Abstract**】The coverage criteria on data-flow test is achieved by applying trap properties and the temporal logic relationship, the problem of test sequence generation is transformed into the one of finding the counter example in model check, and an automatic generation approach of the test sequences calling methods in class is presented. A searching algorithm to generate automatically the test sequences is designed and implemented with the program model checker called Java PathFinder. The algorithm analysis and experimental results show that the presented algorithm can generate efficiently the test sequences and reduce remarkably the test generation cost.

【**Key words**】program model check; dataflow test; generation of test sequences for classes; Java

对类进行测试的主要技术有数据流分析方法，使用源码中的数据流关系来指导测试用例的选取。从Harrold等人面向过程的数据流分析技术[1]到Harrold等提出的类数据流分析框架[2]，都没有实现数据流的自动测试。Buy和Orso等提出一种将数据流分析、符号执行和自动推理相结合的类自动测试框架[3]。模型检测是一种形式化的自动验证技术，它用于检验系统模型是否满足时序逻辑公式表示的性质。大量研究实验表明，将模型检测应用于软件测试生成方法可以明显减少测试代价，提高软件质量[4~7]。但将模型检测与类测试相结合的研究相对较少，特别是类方法调用的交互测试。本文提出将数据流分析和模型检测相结合，通过设置陷阱性质，用时序逻辑公式表示数据流测试的覆盖准则，将测试生成问题简化成模型检测中寻找反例的问题，来研究实现类测试的自动生成方法。

## 1 基于JPF的类数据流测试方法

本文要实现的类数据流测试方法是，以定义-使用对为覆盖准则，使用JPF(Java PathFinder)中的random(*n*)、ramdomBool()和ignoreIf(*cond*)方法，设置陷阱性质引导反例的生成，从而自动生成类方法之间的调用序列。

### 1.1 数据流分析和测试

数据流分析通过对变量构造定义-使用对来实现。数据流测试[8]的基本思想是：一个变量的定义，通过辗转的使用和定义，可以影响到另一个变量的值，或影响到路径的选择等，因此，可以选择一定的测试数据，使程序按照一定变量的定义-使用路径执行，并检查执行结果是否与预期的相符，从而发现代码的错误。

### 1.2 JPF

JPF[9]是一个应用于Java程序的显式模型检测器，能通过在待分析的代码中调用Verify类来处理非确定选择。本文主要用到Verify类的3个方法：(1) ramdomBool()：不确定地返回一个布尔值；(2)random(*n*)：不确定地返回一个[0, *n*]之间的整数值；(3) ignoreIf(*cond*)：当条件*cond*为真时，可使模型检测器从当前状态回退，即不再搜索当前状态可能产生的所有子状态。

### 1.3 测试方法框架及方法调用准则

基于JPF的类数据流测试方法框架如图1所示。

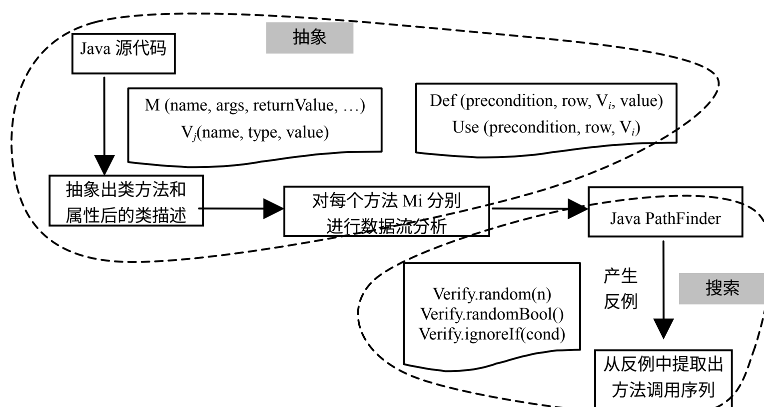


图1 基于JPF的类数据流测试方法框架

**基金项目：**广西自然科学基金资助项目(桂科自0339008)；广西科技信息网络中心资助课题
**作者简介：**唐春艳(1982－)，女，硕士研究生，主研方向：高可信软件技术；钟 诚，博士、教授
**收稿日期：**2006-11-10 **E-mail：**tcygxu@163.com

基于JPF的类数据流测试方法框架分为抽象和搜索两部分，抽象部分针对待测试的Java源码进行类及其数据流信息的提取；搜索部分则根据陷阱性质的思想[4]，在JPF具体搜索环境的设置下，实现定义-使用对并遍历方法调用，以生成满足测试覆盖准则的调用序列。

为防止类方法的无限调用，本文提出如下方法调用的选择准则：

(1)当前属性值须符合待调用方法的前条件要求；

(2)在相同的前条件下，同一方法的两次调用必须产生不同的结果(如两次调用执行的是方法内的不同分支)，否则认为是冗余调用，并用 Verify.ignoreIf(*cond*)进行回退；

(3)不考虑单个方法内的定义-使用对，只考虑公有方法之间的调用序列；

(4)依据定义-使用对的测试覆盖准则，最终生成的方法调用序列是从构造方法开始，经指定变量 $V$ 的定义 $D$ 处，到 $V$ 的使用 $U$ 处终止的一条执行路径，且在定义 $D$ 和使用 $U$ 的方法调用之间不允许调用可能对 $V$ 进行重新定义的方法。

## 1.4 实例分析

本文选用自动售货机实例[3]——类*CoinBox*对本文提出的测试方法进行说明和分析。Java类*CoinBox*描述如下：

```
1   public class CoinBox {
2       int totalQtrs;
3       int curQtrs;
4       int allowVend;
5       CoinBox() {
6           totalQtrs = 0;
7           allowVend = 0;
8           curQtrs = 0;
9       }
10      protected int IsAllowVend() {
11          return allowVend;
12      }
13      public void returnQtrs() {
14      curQtrs = 0;
15      }
16      public void addQtr() {
17          curQtrs = curQtrs + 1;
18          if (curQtrs > 1)
19              allowVend = 1;
20      }
21      void vend() {
22          if (IsAllowVend() == 1) {
23              totalQtrs = totalQtrs + curQtrs;
24              curQtrs = 0;
25              allowVend = 0;
26          }
27      }
28  }
```

其中，*returnQtrs*()包含一个没有将 *allowVend* 重置为 0 的错误，这样包含两次或两次以上对 *addQtrs*()的调用再跟一个 *returnQtrs*()的调用序列，将导致对象进入一个不该出现的状态，尽管投入的硬币都已退还顾客，但对 *vend*()的调用依然是成功的。

## 1.5 对 *CoinBox* 类的测试生成方法分析

提取 *CoinBox* 类的所有方法和属性，方法 $M$ 为{*CoinBox*, *IsAllowVend*, *returnQtrs*, *addQtr*, *vend*}，属性 $V$ 为{*totalQtrs*, *curQtrs*, *allowVend*}。

分析确定 *CoinBox* 类中各个属性的定义和使用情况。

表 1 列出了经第(1)、第(2)步分析后，*CoinBox*类中各方法中属性的定义和使用情况，其中$V_i(0\le i\le 2)$表示属性*totalQtrs*, *curQtrs*和*allowVend*，$v_i(0\le i\le 2)$表示属性的取值。

**表 1　*CoinBox* 类中属性的定义和使用情况**

| 方法名(id) | 属性名(id) | | |
|---|---|---|---|
| | *totalQtrs*($V_0$) | *curQtrs*($V_1$) | *allowVend*($V_2$) |
| *CoinBox*($M_0$) | $d_1$=Def(1,6,$V_0$,0) | $d_2$=Def(1,7,$V_1$,0) | $d_3$=Def(1,8,$V_2$,0) |
| *IsAllowVend*($M_1$) | | | $u_1$=Use(1,11,$V_2$) |
| *returnQtsr*($M_2$) | | $d_4$=Def(1,14,$V_1$,0) | |
| *addQtsr*($M_3$) | | $u_2$=Use(1,17,$V_1$)<br>$d_5$=Def(1,17,$V_1$,$v_1$+1)<br>$u_3$=Use(1,18,$V_1$) | $d_6$=Def($V_1$>1,19,$V_2$,1) |
| *Vendl*($M_4$) | $u_4$=Use($V_2$=1,23, $V_0$)<br>$d_7$=Def($V_2$=1,23, $V_0$,$v_0$+$v_1$) | $u_5$=Use($V_2$=1,23,$V_1$)<br>$d_8$=Def($V_2$=1,24,$V_1$,0) | $d_9$=Def($V_2$=1,25,$V_2$,0) |

由表 1 可知，*CoinBox*类中所有可能的定义-使用对$<d_1, u_4>,<d_2,u_2>,<d_4,u_3>,<d_2,u_5>,<d_4,u_5>,\cdots$，其中有些是无效的，如$<d_4,u_3>$和$<d_2,u_5>$。可以通过下述步骤进行　区分。

设置 JPF 下的陷阱性质。

例如，要得到满足$<d_5,v_5>$的测试，设置陷阱性质*fd*和*fu*：

```
5       CoinBox() {
        …
8       curQtrs = 0;        fd = 0;
9       }
13      public void returnQtrs() {
14      curQtrs = 0;        fd = 0;
15      }
16      public void addQtr() {
17      curQtrs = curQtrs + 1;     fd = 1;
        …
20      }
21      void vend() {
        …
23       totalQtrs = totalQtrs + curQtrs;
        if( fd = = 1){
        fu = 1;
        assert(1 = = 0);
        …
28   }
```

在源码 17 行令*fd*=1，此外，所有对*curQtrs*定义处都令*fd*=0，即在 17 行定义了*curQtrs*后，再调用任何其他对*curQtrs*的定义，都将导致*fd*无效，同时在 23 行对*curQtrs*的引用处，加入*fd*和*fu*的一个时序判断语句，实现对$<d_5,v_5>$的判定。

利用 JPF 中 *Verify* 类设置具体搜索环境。

```
static int N = 3;
 public static void main(String[] args)
  {   …
        for (int i = 0; i < N; i++)
        {       Verify.beginAtomic();
                int f = Verify.random(2);   //0,1,2
                switch (f) {
                case 0:
                        cb.returnQtrs();
                        invokeSeq += "->returnQtrs";
                //Verify.ignoreIf(!bs.isNewState(cb));
```

```
                break;
            case 1:
                cb.addQtr();
    invokeSeq += "->addQtr";
            // Verify.ignoreIf(!bs.isNewState(cb));
                break;
            case 2:
                cb.vend();
                invokeSeq += "->vend";
```

其中，$N$ 表示搜索树高；*bs* 存储已访问过的状态信息，用于判断上一次调用产生的结果状态是否冗余。使用 JPF 默认的深度优先搜索算法，产生如图 2 所示的搜索树。

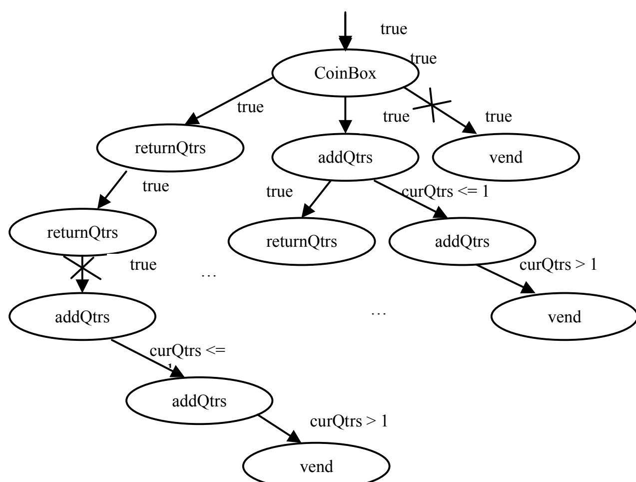


**图 2　类 *CoinBox* 的搜索生成树**

这样，当搜索进行到assert语句时，将产生一个异常并生成反例，该反例包含满足$<d_5,v_5>$覆盖的方法调用序列 *CoinBox*()->*addQtr*()->*addQtr*()->*vend*()。

当试图寻找满足$<d_2,u_5>$的方法调用序列时搜索结果是失败的，即没有生成反例，因此，判定$<d_2,u_5>$无效。

对于$<d_4,u_5>$，当$N$=3 时，判定为无效；当$N$=5 时，有 *CoinBox*()->*returnQtrs*()->*addQtr*() ->*addQtr*()->*returnQtrs*() ->*vend*()，即发现了类*CoinBox*中的错误调用序列。而当$N$=5 时，$<d_5,v_5>$对应的调用序列变为 *CoinBox*()->*returnQtrs*() ->*returnQtrs*()-> *addQtr*()->*addQtr*()->*vend*()，产生了冗余调用。

采用 2.3 节中的选择准则，对于调用序列 *CoinBox*()->*returnQtrs*()->*returnQtrs*()，由于两次调用 *returnQtrs*()的前条件相同而被截断。同样，对于 *CoinBox*()->*vend*()，由于不符合 *vend*()的前条件而被终止(如图 2)。

## 2　实验结果

本文的实验是在 CPU 主频为 2.8GHz、内存为 512MB 的 PC 微机和 Windows XP 操作系统环境下进行的。

表 2 给出了在不同搜索深度下，对类数据流测试的覆盖率进行对比的结果。当 $N$=5 时，覆盖率比 $N$=3 时有较大的提高，并发现了类中错误的定义-使用对。因此，在进行类测试时，一定要选取合适的搜索深度。本文以类方法的分支情况为准，在 *CoinBox* 类中，*returnQtrs*()只有 1 条执行路径，*addQtr*()和 *vend*()都有 2 个分支，则选取 $N$ 为 1+2+2=5。采用 1.3 节中的调用准则，可以在很大程度上提高搜索效率。

表 3 给出应用JPF的深度优先搜索算法和本文提出的搜索算法对$<d_4,u_5>$的测试进行比较的结果。

**表 2　不同搜索深度的类数据流测试的覆盖率**

| N | all-defs | all-uses | all du-pairs | available du-pairs | inavailable du-pairs | intra-method du-pairs | Coverage /(%) | error find |
|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 5 | 17 | 7 | 6 | 4 | 53.8 | N |
| 5 | 9 | 5 | 17 | 8 | 5 | 4 | 61.5 | Y |

**表 3　DFS和本文算法对$<d_4,u_5>$的测试生成结果**

| Searching Techniques | Abs Time/ms | new states | visited states | backtracks | processed states | Total Memory/KB |
|---|---|---|---|---|---|---|
| DFS/JPF | 453 | 23 | 38 | 55 | 17 | 3 408 |
| 本文算法 | 328 | 8 | 3 | 5 | 2 | 4 200 |

由于考虑到可能产生的冗余状态，并及时进行回退，因此在内存消耗基本相同的情况下，本文算法与 DFS 相比，搜索过程中产生的新状态数、回退次数和处理的状态数都比较少，因此，本文算法的执行时间也相对较短。

## 3　结束语

本文将模型检测和类的数据流分析技术相结合，采用抽象和陷阱性质的方法，基于 JPF 模型检测器，提出一种自动生成满足数据流覆盖准则的类方法调用序列的搜索算法。算法分析和实验结果表明本文提出的算法是高效的，可以明显减少生成调用序列的长度以及测试生成的代价。下一步将研究如何把数据流测试扩展到交互类及交互组件之间的测试。

## 参考文献


1 Harrold M J, Soffa M L. Interprocedual Data Flow Testing[C] //Proceedings of the 3th Testing, Analysis, and Verification Symposium, Key West, Florida, United States. New Year: ACM Press, 1989: 158-167.

2 Harrold M J, Rothermel G. Performing Data Flow Testing on Classes[C]//Proceedings of 2nd ACM-SIGSOFT Symposium on the Foundations of Software Engineering, New Orleans, LA, USA. New Year: ACM Press, 1994: 154-163.

3 Buy U, Orso A, Pezze M. Automated Testing of Classes[C] //Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States. New Year: ACM Press, 2000: 39-48.

4 Gargantini A, Heitmeyer C. Using Model Checking to Generate Tests from Requirements Specifications[C]//Proceedings of the 7th European Software Engineering Conference. Toulouse: Springer-Verlag, 1999: 146-162.

5 Visser W, Pasareanu C, Khurshid S. Test Input Generation with Java PathFinder[C]//Proceedings of ISSTA'04, Boston, MA. New Year: ACM Press, 2004-07: 97-107.

6 Hong H S, Sung Deok Cha. Data Flow Testing as Model Checking[C] //Proceedings of the 25th International Conference on Software Engineering'03. Portland, Oregon: IEEE Computer Society, 2003-05: 232-242.

7 Hong H S, Lee I, Olsky O S, et al. Automatic Test Generation from Statecharts Using Model Checking[C]//Proceedings of the 1st Workshop on Formal Approaches to Testing of Software. 2001: 15-30.

8 Rapps S, Weyuker E J. Selecting Software Test Data Using Data Flow Information[J]. IEEE Trans. on Softw. Eng., 1985, 11(4): 367-375.

9 Mehlitz P C, Visser W, Penix J. The JPF Runtime Verification System[Z].(2006-11). http://javapathfinder.source forge.net.