运行时异常对软件静态测试的影响研究

金大海 宫云战 杨朝红 肖 庆
(北京邮电大学网络与交换技术国家重点实验室 北京 100876)

摘 要 当程序的执行过程中出现运行时异常,控制流动态地变更可能会产生非预期的执行逻辑,由此引入的缺陷将给软件静态测试工作带来巨大的挑战.针对这一问题,提出一种结合运行时异常的静态测试方法,将缺陷检测及控制流扩展交替执行,通过扩展分析路径达到提高测试充分度的目的.以异常模式状态机描述运行时异常行为,在包含运行时异常的控制流上,应用迭代方程得到运行时异常控制流序列,将在传统控制流上的一次缺陷检测过程扩展为在异常控制流序列上的多次检测.实验结果表明,结合运行时异常的静态测试方法虽然增加了时间开销,且引入一定的误报,但却可以发现传统测试方法所遗漏的缺陷,这点在航空、航天等高可信领域尤为重要.

关键词 软件测试;静态分析;运行时异常;异常模式;异常控制流
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01090

Research on the Effect of Runtime Exception in Software Static Testing

JIN Da-Hai GONG Yun-Zhan YANG Zhao-Hong XIAO Qing
(*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing* 100876)

Abstract When a runtime exception occurred during the execution of program, an unexpected executing logic maybe appeared due to the modified of control flow dynamically. Thus, a great challenge, detecting the defect caused by the unexpected executing logic, will be brought to software static testing. Therefore an approach for iterative defect detection in the presence of runtime exception is proposed, in which defect detection and control flow extension are interleaved and performed repeatedly. The finite state machine for runtime exception is defined as EM, which include all the state of runtime exception related, and the conversation behavior between different states. The control flow with runtime exception is defined as EC, which is the basis for defect detection. By the given iterative equation and algorithm, the sequence of EC can be got, and on which the traditional onetime testing is extended to multiple times. Experimental results show that, although sequence of EC increases the time cost and false positive, more defects can be detected than ever, and these defects are missed by other method or tools indeed.

Keywords software testing; static analysis; runtime exception; exception pattern; exception control flow

1 引 言

与传统的软件测试方法不同,面向代码缺陷的静态测试技术首先从软件的代码层面上总结缺陷,将其抽象成相应的缺陷模式,然后对程序中相关表达式的取值进行近似计算,最后将计算结果应用于缺陷检测.自 2000 年以来,面向代码缺陷的软件静

收稿日期:2010-11-16;最终修改稿收到日期:2011-05-14.本课题得到国家"八六三"高技术研究发展计划项目基金(2009AA012404)和国家自然科学基金(91018002)资助.**金大海**,男,1974 年生,博士,讲师,研究方向为软件测试. E-mail: jindh@bupt.edu.cn. **宫云战**,男,1962 年生,博士,教授,研究领域为容错计算、软件测试. **杨朝红**,男,1976 年生,博士,副教授,研究方向为软件测试. **肖 庆**,男,1979 年生,博士研究生,研究方向为软件测试.

态测试技术[1-5]正逐渐受到软件业界的青睐，已成为美国等一些国家的一种主流的软件测试技术. 目前具有代表性的代码缺陷检测工具主要有 Coverity①，CodeSonar②，FindBugs③，K8④ 和 Fortify⑤ 等等，对比检测工具的测试结果表明，现有工具均存在不同程度的误报及漏报情况.

为了能够处理一些非预期的情况，Java 语言提高对两类异常的支持：检查型异常和运行时异常. 编译器强制要求对检查型异常进行处理，而对运行时异常却难以限制. 当运行时异常发生后，系统会按照方法调用关系的逆序逐级上抛，直到相应异常被捕获，否则终止程序或线程. 运行时异常的影响对于被终止的程序或线程是明显的，但若在上抛过程中被捕获到，程序或线程并不会终止，但可能会影响其正常执行逻辑，由此造成难以检测及定位的错误.

由于异常处理机制的引入，使程序的控制流变得更为复杂，给静态分析及测试工作带来更大的挑战. 目前许多专家学者虽然就此进行研究，但对于“当一个运行时异常发生后，它是否会导致其它新的运行时异常？这些运行时异常对静态测试结果产生哪些影响？”等问题却是难以回答. 目前所有的代码缺陷检测技术及工具均没有考虑到该问题，由此引起的路径分析不充分是导致缺陷漏报的主要原因之一. 因此，本文就运行时异常对 Java 程序静态测试的影响进行进一步研究.

2　问题描述

面向代码缺陷的静态测试技术从理论上可描述为对问题 $D=\{P,M,A\}$ 的求解，这里 P 代表被测程序；M 是与 P 对应的代码缺陷模式集合；A 是缺陷检测算法，用于计算 P 中相关表达式的取值信息. 算法描述为 $A=\{S,\rho(L,X),\sigma(F)\}$，其中 S 为程序 P 中的路径；ρ 为近似地计算相关变量 X 的取值信息，L 为 P 的执行位置，X 为 P 执行到 L 处的相关变量；σ 为方法的摘要信息，近似地表示对 P 中被调用方法 F 的抽象. 由于运行时异常的引入，将受其影响的控制流路径 S 扩展为 S_1，而在 S_1 上又可能发生新的运行时异常，经过如此迭代过程后，可得到一个路径集合 $[S_1,\cdots,S_n]$，其中 S_i 是由 S_{i-1} 上发现的新运行时异常扩展而成，且 $S_{i-1}\subset S_i$，由此对 A 的描述转化为 $A=\{S\cup[S_1,\cdots,S_n],\rho(L,X),\sigma(F)\}$.

在静态测试过程中，如果忽略了对运行时异常的处理，即在 A 中遗漏了分析路径 $[S_1,\cdots,S_n]$，则会有 $\{\rho(L,X)\mid S\}\subseteq\{\rho(L,X)\mid S\cup[S_1,\cdots,S_n]\}$，由此会导致缺陷漏报情况的出现，即

$$\left.\begin{array}{l}\forall x,x\in\{\rho(L,X)\mid S\}\wedge\neg D(P,M,A)\\ \exists x,x\in\{\rho(L,X)\mid S\cup[S_1,\cdots,S_n]\}\wedge\\ \qquad D(P,M,A)\end{array}\right\}\Rightarrow FN(X),$$

其中 $D(P,M,A)$ 表示由于变量 X 的取值 x 引起 M，$\neg D(P,M,A)$ 表示变量 X 的取值 x 不会引起 M，$FN(X)$ 表示对变量 X 的漏报.

以下 Java 代码片段包括三级方法调用，$f3$ 调用 $f2$，$f2$ 调用 $f1$. 如果不考虑运行时异常的影响，现有测试方法可以很容易地检测到第 7 行 p 的空指针异常（尽管它可以被捕获到）. 但除此之外还存在其它的问题吗？结合运行时异常对路径的影响后，答案显然是否定的，实际结果应为

(1) 如果第 5 行的条件为真，则程序执行到第 7 行时会抛出一个空指针异常，且它在第 9 行被捕获.

(2) 第 3 行的 q 初始值为空，当出现结果 1 的情况时，会略过第 8 行对 q 的资源分配操作而执行第 9 行的异常捕获，这样会在第 10 行触发 q 的空指针异常，且由于未被捕获而抛出 $f1$.

(3) 当出现结果 2 中的情况时，会在第 10 行抛出空指针异常，则原本在第 11 行释放资源 r 的操作被略过，造成对 r 的资源泄漏.

(4) 当出现结果 2 中的情况时，第 17 行的 $f1$ 会抛出空指针异常，由于此异常不能被 18 行捕获，从而造成 16 行对 q 分配的资源泄漏.

(5) 由于在 27 行捕获了所有类型的异常，使运行时异常不能传播到程序入口而终止线程过进程，因此以上问题会隐藏起来，难以检测及定位.

```
class test {
1.  FileInputStream p,r;
2.  void f1() throws IOException {
3.      FileInputStream q=null;
4.      try{
5.          if(p==null)
6.              System.out.println("null!");
7.          p.mark(0);
8.          q=new FileInputStream("output");
9.      }catch(NullPointerException e){}
10.     q.close();
11.     r.close();
```

① http://www.coverity.com/
② http://www.gramatech.com/
③ http://findbugs.sourceforge.net/
④ http://www.klocwork.com/
⑤ http://www.fortify.com/

```
12. }
13. void f2() throws IOException{
14.     FileInputStream q=null;
15.     try{
16.         q=new FileInputStream("input");
17.         f1();
18.     }catch(IOException e){
19.         q.close();
20.         return;
21.     }
22.     q.close();
23. }
24. void f3(){
25.     try{
26.         f2();
27.     }catch(Exception e){}
28. }
    ...
}
```

3 运行时异常描述及检测

3.1 异常描述

问题 $D=\{P,M,A\}$ 中的缺陷模式 M 是求解 D 的基础，如何将运行时异常特征包含于 M 之中是首先需要解决的问题，本文采用异常模式状态机对其进行统一描述如下.

定义 1. 异常模式状态机定义为一个三元组 $EM=\langle N,T,C\rangle$. N 为状态集合，包括一个异常模式中所有可能达到的状态，$N=\{N_{\text{start}},N_{\text{exception}},N_{\text{end}}\}\cup N_{\text{other}}$. $T=\{\langle n_i,n_j\rangle \mid n_i,n_j\in N\}$ 是状态转换集合，表示状态机可以从状态 n_i 转换到状态 n_j. C 是状态转换条件，$T:N\times C\rightarrow N$.

由于运行时异常的发生模式多种多样，所以不同类型的异常由不同的异常模式状态机来描述，但所有异常模式状态机间的共有状态有 $\{N_{\text{start}},N_{\text{exception}},N_{\text{end}}\}$. N_{start} 是状态机的唯一入口，N_{end} 是状态机的唯一出口，$N_{\text{exception}}$ 代表发生运行时异常的状态. N_{other} 表示除以上 3 种状态之外的其它状态，随异常类型的不同而不同.

在给定某种异常模式状态机的所有状态后，状态间的转换集合 T 及其对应的转换条件 C 需要手工设定. 图 1 是一个空指针异常模式状态机，除了共有的状态 N_{start}、N_{end} 和 $N_{\text{exception}}$ 外，还具有代表非空的状态 N_{not} 和代表可能为空的状态 N_{may}. 经人工分析，与空指针异常相关的状态转换条件有 C_1（确定对象可能空），C_2（确定对象非空），C_3（引用对象内容），C_4（超出对象作用域），C_5（自动转换），C_6（其它），与其相关的状态转换集合为 $T_1:N_{\text{start}}\times C_6\rightarrow N_{\text{start}}$，$T_2:N_{\text{start}}\times C_1\rightarrow N_{\text{may}}$，$T_3:N_{\text{start}}\times C_2\rightarrow N_{\text{not}}$，$T_4:N_{\text{may}}\times C_6\rightarrow N_{\text{may}}$，$T_5:N_{\text{may}}\times C_3\rightarrow N_{\text{exception}}$，$T_6:N_{\text{may}}\times C_4\rightarrow N_{\text{end}}$，$T_7:N_{\text{may}}\times C_2\rightarrow N_{\text{not}}$，$T_8:N_{\text{not}}\times C_6\rightarrow N_{\text{not}}$，$T_9:N_{\text{not}}\times C_4\rightarrow N_{\text{end}}$，$T_{10}:N_{\text{not}}\times C_1\rightarrow N_{\text{may}}$，$T_{11}:N_{\text{exception}}\times C_5\rightarrow N_{\text{end}}$.

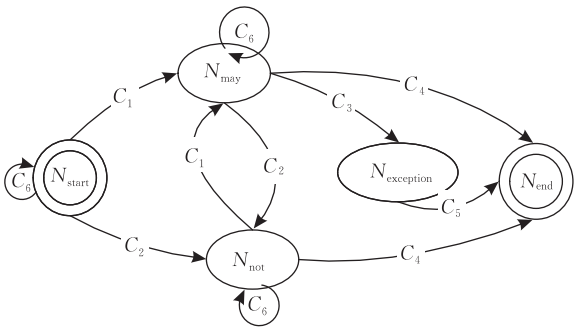


图 1 空指针异常模式状态机

对于其它类型的异常模式描述，是一个与上述空指针异常模式类似的手工生成过程：首先确定异常相关状态，然后分析所有状态间的转换条件，最后生成不同状态间的转换集合.

在异常模式状态机上进行状态转换时，需要结合具体变量 X 的取值信息 $\rho(L,X)$ 及状态转换条件 C 来完成，下面给出状态机实例的概念.

定义 2. 针对变量 X 创建的异常模式状态机 EM 称为对应 X 的异常模式状态机实例 $EM(X)$.

$EM(X)$ 上的实际状态转换为 $n_i=\{n_j\times C \mid \rho(L,X)\}$，结合具体变量 X 在位置 L 处的取值信息，通过状态转换条件 C 可将 $EM(X)$ 的当前状态 n_j 转换为状态 n_i.

对独立变量 X 的异常模式状态机实例描述为 $EM(X)$，对程序 P 所对应的异常模式状态机实例可做如下描述.

定义 3. 假设程序 P 包括 n 种既定的运行时异常模式 $\{EM_1,EM_2,\cdots,EM_n\}$，受 P 中的相关变量 X 影响，每种异常模式 EM_i 最多对应 m 种异常模式状态机实例 $\{EM_i(X_1),EM_i(X_2),\cdots,EM_i(X_m)\}$，则程序 P 对应的所有异常模式状态机实例构成 P 的异常模式状态机实例矩阵，记作

$$(EM_{ij})_{n\times m}=\begin{bmatrix} EM_1(X_1) & EM_1(X_2) & \cdots & EM_1(X_m)\\ EM_2(X_1) & EM_2(X_2) & \cdots & EM_2(X_m)\\ \cdots & \cdots & & \cdots\\ EM_n(X_1) & EM_n(X_2) & \cdots & EM_n(X_m)\end{bmatrix}.$$

$EM_i(X_j)=\varnothing$ 表示变量 X_j 与异常模式 EM_i 无关，不需要为 X_j 创建该类状态机实例. 对于 $(EM_{ij})_{n\times m}$

中所有非空状态机实例，表示需要为程序 P 中的变量 X_j 创建异常模式状态机实例. 在所有非空的状态机实例中，相同的状态为 $\{N_{start}, N_{exception}, N_{end}\}$，至于其它的中间状态 N_{other} 和状态转换条件 C 随 EM_i 的不同而各异.

3.2　异常检测

有了以上对运行时异常进行描述的方法后，可以为程序中与特定异常模式 EM 相关的变量 X 创建异常模式状态机实例 $EM(X)$，应用缺陷检测算法 $A=\{S,\rho(L,X),\sigma(F)\}$，结合与 X 相关的取值信息及状态转换条件实现状态转换. 当 $EM(X)$ 中出现异常状态 $N_{exception}$ 时，表示在位置 L 处可能发生运行时异常.

$\rho(L,X)$ 的计算原理是以传统的数据流迭代方程为基础，在遍历控制流每一个节点的过程中，计算相关变量的取值信息. 这个过程的详细信息可参考文献[6-8]，这里仅以 Java 代码片段中 4～8 行为例，简单描述对运行时异常的检测过程. 为 p 创建空指针异常模式状态机实例后，异常检测过程如图 2 所示. 其中控制流节点序号对应代码片段中的行号，unknow 表示对象取值未知，all 表示对象取值为空或非空，null 表示对象取值为空. 应用 p 的取值信息及状态转换条件，异常模式状态机实例 $EM(p)$ 的状态从控制流节点 4 上的 N_{start} 状态逐步转换到控制流节点 8 上的 N_{end} 状态，且在节点 7 上发现异常状态 $N_{exception}$.

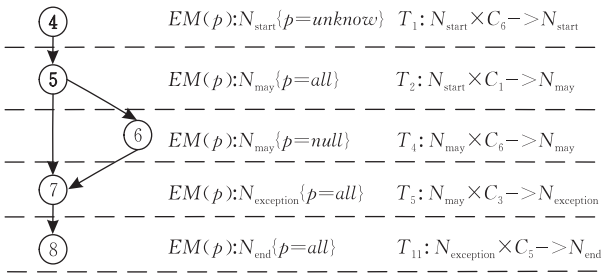


图 2　异常模式状态机实例在 $f1$ 片段上的状态转换

以上过程表明控制流节点 7 所对应的语句可抛出空指针异常，即为可抛出运行时异常的语句；若该运行时异常不能被当前方法所捕获，则当前方法为可抛出运行时异常的方法. 由于第 2 节程序中第 7 行抛出的运行时异常会在第 9 行被捕获，因此方法 $f1$ 在此处不能抛出该异常. 但在第 7 行异常触发的条件下，第 10 行也会抛出空指针异常，而该异常不会被 $f1$ 捕获，由此 $f1$ 为可抛出运行时异常的方法.

为了能够准确应用运行时异常信息，定义异常向量如下.

定义 4.　异常向量用于描述在静态测试过程中检测到的运行时异常，包括运行时异常发生的位置 P 及对应的异常类型 T，表示为 $\boldsymbol{TP}$. 向量的模 $|\boldsymbol{TP}|$ 表示异常相关变量的当前取值信息.

应用异常向量，通过异常模式状态机实例及缺陷检测算法检测发现运行时异常的过程可描述为 $EM(X)\,A(X)\rightarrow \boldsymbol{TP}$，其中向量的位置 P 由算法 $A(X)$ 中的 L 确定，向量的类型 T 与 $EM(X)$ 相关，向量的模 $|\boldsymbol{TP}|$ 与 $\rho(L,X)$ 相关.

当运行时异常被触发后，若方法 F 内的控制结构可以捕获到，则该异常不会传播出方法；否则该异常会被抛出方法 F，所有此类异常构成方法可抛出的异常向量，记作 $\boldsymbol{F}=\bigcup \boldsymbol{TP}$.

4　运行时异常控制流

4.1　与传统控制流的区别

传统方法在构建程序控制流时，除了包含显式抛出异常的语句和方法外，也包含隐式抛出异常的方法，如 Java 语言中的检查型异常，但对于可抛出运行时异常的语句和方法却很少涉及.

运行时异常控制流是描述包括运行时异常处理的程序有向图，它在程序原控制流的基础上，增加了描述运行时异常处理结构的点集和边集，包含的信息有 $\{N,E,N_{er},N_{ef},E_{re}\}$. 其中 N 是节点集，每个程序语句都对应图中的一个节点. 除了传统控制流图中的节点外，N_{er} 是包括可抛出运行时异常语句的节点，N_{ef} 是包括可抛出运行时异常方法的节点. 边集 $E=\{\langle n_i,n_j\rangle \mid n_i,n_j\in N\}$，表示节点 n_i 对应的语句执行后，可能立即执行节点 n_j 对应的语句. E_{re} 为 N_{er} 和 N_{ef} 节点的一条输出边，对应被抛出运行时异常的处理：被捕获的运行时异常通过 E_{re} 连接到相应的捕获节点，未被捕获的运行时异常通过 E_{re} 连接到控制流的异常退出节点. $N=\{N_{er},N_{ef}\}\bigcup N_{other}$，$N_{other}$ 表示传统控制流中的节点，$E=\{E_{re}\}\bigcup E_{other}$，$E_{other}$ 表示传统控制流中的边.

基于以上描述，运行时异常控制流与传统控制流区别总结如下：

(1) 为了便于定位运行时异常，每个语句作为一个控制流节点，而不是一个语句块；

(2) 每一个能够抛出运行时异常的方法，都有一个异常退出节点作为所有方法内未被捕获异常的

后继节点.为了保证控制流中出口的唯一性,每一个控制流增加一个正常退出节点,它和异常退出节点的后继节点为控制流的退出节点;

(3)对应每一个可抛出运行时异常的节点 N_{er} 或 N_{ef},通过边 E_{re} 连接到与其类型对应的异常捕获节点,或没有与其对应的异常退出节点.

4.2 异常控制流序列

当运行时异常被触发时,它会动态地扩展当前控制流,而在被扩展的控制流上又可能出现新的运行时异常,这个运行时异常对控制流的扩展和在扩展的控制流上发生新运行时异常的迭代过程会产生一个控制流序列,其定义如下.

定义 5. 异常控制流序列表明运行时异常行为与扩展的控制流之间的交互关系,包括运行时异常对控制流的扩展和在扩展的控制流上检测新的运行时异常以及如此重复迭代的过程,表示为一个四元组 $EC=\langle RS,ES,G,T\rangle$. RS 是一个控制流有序集合$\{RS_0,RS_1,\cdots,RS_n\}$,表示受运行时异常的影响,控制流从 RS_0 开始逐步扩展到 RS_n;$ES=\{ES_0,ES_1,\cdots,ES_n\}$ 是一个与 RS 对应的集合,由若干异常向量集合构成,$\boldsymbol{TP}\in ES_i$ 表示在 RS_i 上可发现的异常向量;G 是从 RS_i 上生成 ES_i 的规则,T 是通过 ES_i 将 RS_i 转化为 RS_{i+1} 的规则.

在规则 G 下,$ES_0=\bigcup\limits_{\forall N\in REC_0}\{F|F\in N\}\cup\{D(P,M,A)|RS_0\}$ 包括两部分内容,其一是控制流 RS_0 中所有包括方法调用的节点所抛出的异常向量;其二是在控制流 RS_0 上对问题 $D(P,M,A)$ 求的有效解. D 中的 M 为定义 3 中的状态机实例矩阵 $(EM_{ij})_{n\times m}$,在计算过程 $(EM_{ij})_{n\times m}\times A$ 中,

$$\begin{bmatrix} EM_1(X_1) & EM_1(X_2) & \cdots & EM_1(X_m)\\ EM_2(X_1) & EM_2(X_2) & \cdots & EM_2(X_m)\\ \cdots & \cdots & & \cdots\\ EM_n(X_1) & EM_n(X_2) & \cdots & EM_n(X_m)\end{bmatrix}\times\begin{bmatrix}A(X_1)\\A(X_2)\\\cdots\\A(X_m)\end{bmatrix}=$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{m}EM_i(X_j)A(X_k),$$

当变量 X_j 与异常模式 EM_i 无关时 $EM_i(X_j)A(X_k)=\varnothing$,另外,当 $j\neq k$ 时 $EM_i(X_j)A(X_k)=\varnothing$. $EM_i(X_j)A(X_j)\neq\varnothing$ 表示对非空实例 $EM_i(X_j)$ 应用算法 $A(X_j)$,结合 EM_i 的状态转换条件 C 实现状态转换.当在 RS_0 的节点 N 得到 $EM_i(X_j)$ 的异常状态 $N_{exception}$ 时,即有 $EM_i(X_j)A(X_j)\rightarrow\boldsymbol{TP}$,其中位置 P 对应到 RS_0 中的节点 N,类型 T 对应到 EM_i 表示的异常类型,$|\boldsymbol{TP}|$ 对应到 X_j 的取值 $\rho(N,X_j)$.

在规则 G 下,$E_k=\{D(P,M,A)|RS_k\}$ 是在异常控制流 RS_k 上对问题 $D(P,M,A)$ 求的新有效解. M 为状态机实例矩阵 $(EM_{ij})_{n\times m}$,对于 M 中的任一实例 $EM_i(X_j)$ 所对应的变量 X_j,算法 $A(X_j)$ 在路径 $S\cup\bigcup\limits_{l=1}^{k}S_l$ 上的不同执行位置 L 对 X_j 的取值信息进行计算,最后得到控制流 RS_k 上所有可能抛出的新异常集合 $ES_k=\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{m}EM_i(X_j)A(X_j)-\sum\limits_{i=0}^{k-1}ES_i$.

在规则 T 下,$RS_{i+1}=RS_i\times ES_i$,应用 ES_i 中记录的异常 $\bigcup\limits_{j=1}^{\mathrm{sum}(ES_i)}T_jP_j$,在控制流 RS_i 上应用位置 P_j 标记抛出运行时异常的节点 N_{er},应用类型 T_j 添加相应异常处理边 E_{re},最后将其扩展为 RS_{i+1},$\mathrm{sum}(ES_i)$ 为 ES_i 中异常向量的数量.

通过对异常生成规则 G 和控制流转换规则 T 的反复迭代,可以实现异常控制流集 RS 的生成.迭代方程表示如下,假设 RS_0 已知,迭代过程从 $t=1$ 开始,直到 $ES_t=\varnothing$ 结束.

$$RS_t=\begin{cases}RS_0, & t=0\\ RS_{t-1}\times ES_{t-1}, & t\geq 1\end{cases},$$

$$ES_t=\begin{cases}\bigcup\limits_{\forall N\in RS_0}\{F|F\in N\}\cup\{(EM_{ij})_{n\times m}\times A|RS_0\}, & t=0\\ \{(EM_{ij})_{n\times m}\times A|RS_t\}-\sum\limits_{i=0}^{t-1}E_i, & t\geq 1\end{cases}.$$

基于以上描述,异常控制流序列与传统异常控制流间的区别如下:

(1)传统异常控制流节点中与异常相关的节点只有抛出显式异常和隐式异常的节点,而运行时异常控制流中除了这些节点外,还包括抛出运行时异常的 N_{er} 和 N_{ef} 节点以及处理运行时异常的边 E_{re};

(2)在传统异常控制流 RS_0 上进行静态测试时,只能报告当前发现的运行时异常信息 E_0 以及与此相关的非异常问题;而应用异常控制流序列的方法,从传统异常控制流 RS_0 开始,通过迭代方程将其扩展为$\{RS_0,RS_1,\cdots,RS_n\}$,这样可发现更多的运行时异常$\{ES_1,\cdots,ES_n\}$以及其它非异常问题.

5 异常控制流序列生成算法

5.1 算法描述

下面以前面介绍的迭代方程为基础,给出异常控制流序列的生成算法.算法的输入为方法 F 对应的不包含节点 N_{er} 和 N_{ef} 的传统异常控制流 RS_0、异

常模式状态机实例矩阵$(EM_{ij})_{n\times m}$以及用于计算相关内容取值信息的方法[6-7]，输出为异常控制流集RS和对应的异常向量集ES.

```
1. begin
2.   I=0; ES[I]=∅;
3.   RS[I]=CFG(F);
4.   for each N in RS[I] do
5.     if F′∈N then ES[I]=ES[I]∪F′;
6.   end for
7.   for each EM_i(X_j) in (EM_ij)_{n×m} do
8.     ES[I]=ES[I]+{EM_i(X_j)A(X_j)|RS[I]};
9.   end for
10.  while ES[I]≠∅ do
11.    ES[I+1]=∅
12.    for each TP in ES[I] do
13.      RS[I+1]=RS[I]+TP;
14.    end for
15.    for each EM_i(X_j) in (EM_ij)_{n×m} do
16.      ES[I+1]=ES[I+1]+
         {EM_i(X_j)A(X_j)|RS[I+1]}−Σ_{i=0}^{I} ES[i];
17.    end for
18.    I++;
19.  end while
20.  for each E_re in RS[I].Nexcp_exit do
21.    add(F,E_re);
22.  end for
23. end
```

第4～6行首先统计F中所有被调用方法F'的异常向量$\boldsymbol{F}'$，并将其累加到$ES[0]$中；7～9行遍历方法F的状态机实例矩阵$(EM_{ij})_{n\times m}$，应用方法A在控制流$RS[0]$上检测异常向量，也将结果累加入到$ES[0]$中；若经过前面的处理后$ES[0]$不为空，则从第10行开始应用迭代方程求解$RS[I+1]$，指导$ES[I+1]=\varnothing$终止.

第10～19行为主迭代过程，直到在当前控制流上没有新的异常向量产生才终止循环. 其中第12～14行首先计算$ES[I]$中的异常向量$\boldsymbol{TP}$对$RS[I]$的变更，即应用位置P在$RS[I]$中标记异常抛出节点N_{er}或N_{ef}，应用异常类型T在$RS[I]$中加入异常处理边E_{re}，由此得到$RS[I+1]$；接下来的15～17行再次遍历$(EM_{ij})_{n\times m}$，在$RS[I+1]$上应用算法A对所有状态机实例进行状态转换，以发现新的异常向量$\boldsymbol{TP}$，并加入到$ES[I+1]$之中. 通过10～19行的while循环，可以逐步迭代出RS中的所有成员，直到没有最新异常出现，即$ES[I]=\varnothing$.

最后，为了得到当前方法F所能抛出的所有异常向量，第20～22行将当前控制流中所有连接到异常退出节点N_{excp_exit}的边加入到其对应的异常向量$\boldsymbol{F}$中.

以上算法循环条件是以控制流的扩展为基础的，对它的有限性证明如下：(1)假设方法F中有m条语句，则最坏情况下的循环次数为m次，因此总迭代次数是有限的；(2)将RS映射成函数$RS=\{y|y=f(x)\}$后，对于$f(x)$定义域$x\in[0,m]$内的任意两点x_1及x_2，且x_1小于x_2，则恒有$f(x_1)\subset f(x_2)$，因此函数$f(x)$在区间$x\in[0,m]$上是单调增加的. 根据上面(1)、(2)可得，单调有界的函数$f(x)$必定是有极限的，因此求解异常控制流集RS的循环次数是有限的.

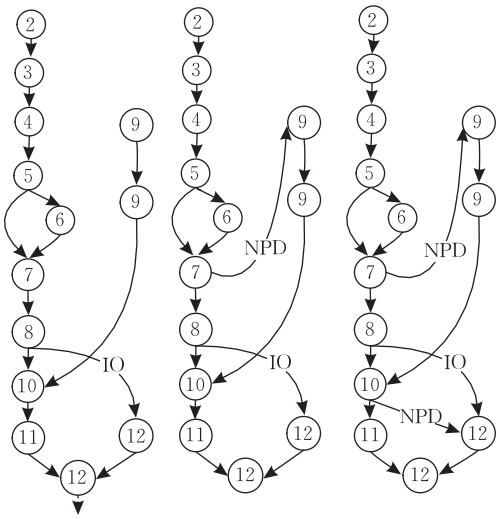
5.2　应用举例

下面以代码片段中方法$f1$为例，对以上算法在软件静态测试中的应用步骤进行举例说明. 以方法$f1$作为当前分析单元，其输入信息包括$f1$的初始控制流图$RS[0]$，如图3(a)所示. 其中每个节点上的序号对应源代码的行号，为了避免对资源泄漏的过多误报，控制流中屏蔽了close方法的检查型异常. 假设对$f1$检测的缺陷模式有如下两种：空指针模式EM_{npd}和资源泄漏模式EM_{mlf}，由于资源泄漏不会引起运行时异常，因此不需要为其创建异常向量. 首先为$f1$创建的状态机实例分别为$EM_{npd}(p)$、$EM_{npd}(q)$、$EM_{npd}(r)$、$EM_{mlf}(q)$、$EM_{mlf}(r)$，限于篇幅，状态机实例的创建方法不在本文讨论. 这些实例共同构成$f1$的状态机实例矩阵$(EM_{ij})_{2\times 3}=\begin{bmatrix}EM_{npd}(p) & EM_{npd}(q) & EM_{npd}(r)\\ EM_{mlf}(p) & EM_{mlf}(q) & EM_{mlf}(r)\end{bmatrix}$. 由于$p$在$f1$中与资源泄漏模式无关，故$EM_{mlf}(p)=\varnothing$，$r$与空指异常无关，故$EM_{npd}(r)=\varnothing$.

对控制流$RS[0]$和$ES[0]$进行初始化操作后，得到初始控制流$RS[0]$为方法$f1$的基本控制流(图3(a))以及初始异常向量集合$ES[0]=\varnothing$. 除库函数外，$f1$没有调用其它子函数，算法4～6行执行完成后，当前异常向量集$ES[0]=\varnothing$.

算法7～9行在控制流$RS[0]$上执行$(EM_{ij})_{2\times 3}$中的非空实例$EM_{npd}(p)$、$EM_{npd}(q)$、$EM_{mlf}(q)$、$EM_{mlf}(r)$，在$RS[0]$的节点7上得到$EM_{npd}(p)$：$N_{exception}$，将异常向量$\boldsymbol{T}_{npd}\boldsymbol{P}_7$加入到集合中，得到$ES[0]=\{\boldsymbol{T}_{npd}\boldsymbol{P}_7\}$.

算法12～14行判断$ES[0]$非空后，应用$ES[0]$中的异常向量扩展$RS[0]$，首先标记节点7为异常



(a) $RS[0]$ (b) $RS[1]$ (c) $RS[2]$

图 3 方法 $f1$ 的异常控制流序列

抛出节点,然后在节点 7 上增加一条抛出空指针异常的边,且该边连接到异常捕获节点 9.经此操作得到 $RS[1]$,如图 3(b)所示.

算法 15～17 行在控制流 $RS[1]$ 上执行 $(EM_{ij})_{2\times3}$ 中的非空实例 $EM_{\text{npd}}(p)$、$EM_{\text{npd}}(q)$、$EM_{\text{mlf}}(q)$、$EM_{\text{mlf}}(r)$,在 $RS[1]$的节点 10 上得到新异常状态 $EM_{\text{npd}}(q):N_{\text{exception}}$,将新异常向量 $\boldsymbol{T}_{\text{npd}}\boldsymbol{P}_{10}$ 加入到集合中,得到 $ES[1]=\{\boldsymbol{T}_{\text{npd}}\boldsymbol{P}_{10}\}$.

算法 12～14 行判断 $ES[1]$非空后,应用 $ES[1]$ 中的异常向量扩展 $RS[1]$,首先标记节点 10 为异常抛出节点,然后在节点 10 上增加一条抛出空指针异常的边,由于该异常没有被捕获而连接到异常退出节点 12.经此操作得到 $RS[2]$,如图 3(c)所示.

算法 15～17 行在控制流 $RS[2]$ 上执行 $(EM_{ij})_{2\times3}$ 中的非空实例 $EM_{\text{npd}}(p)$、$EM_{\text{npd}}(q)$、$EM_{\text{mlf}}(q)$、$EM_{\text{mlf}}(r)$,在 $RS[2]$的退出节点上可得到新异常状态 $EM_{\text{mlf}}(r):N_{\text{exception}}$,由于资源泄漏模式不会抛出运行时异常,因此不需要为其创建异常向量,得到 $ES[2]=\varnothing$而结束循环.

在算法的最后,当前控制流为 $RS[2]$,连接到它的异常退出节点的边有两条:从节点 8 抛出的检查型异常、从节点 10 抛出的运行时异常 E_{npd},因此需要将这两种异常加入到方法 $f1$ 对应的异常向量中,供分析其调用者时使用.

6 实验环境及结果

6.1 实验环境

为了对比应用异常控制流序列前后对静态测试的影响,实验过程在北京邮电大学自主研发的软件缺陷检测系统 DTS(Defect Testing System)[①]上展开,DTS 的测试架构如图 4 所示,主要包括如下内容:

(1)配置文件用于确定需要检测的缺陷类型,可以决定异常模式状态机的数量.

(2)通过分析 Java 源文件中的相关对象,可确定每种异常模式状态机需要创建实例的数量.

(3)抽象语法树以源文件为基础创建,它是对程序分析的基础.

(4)控制流图中存放每个分析单元的控制流信息,它是缺陷检测的基础.

(5)符号表中存放作用域、符号声明及使用、表达式类型推导等信息,它从抽象语法树结构中获取.

(6)函数间分析过程一方面从抽象语法树和符号表中收集信息,通过函数间的调用关系确定分析次序;另一方面对于分析完成的方法,保存其可抛出的异常信息,用于其调用者的分析过程.

(7)区间运算对程序中相关变量的取值信息进行近似计算,它依赖于抽象语法树、控制流图和符号表等数据结构.

(8)当前分析单元的状态机实例创建完成后,需要在控制流的每个节点上运行该实例进行状态转换.

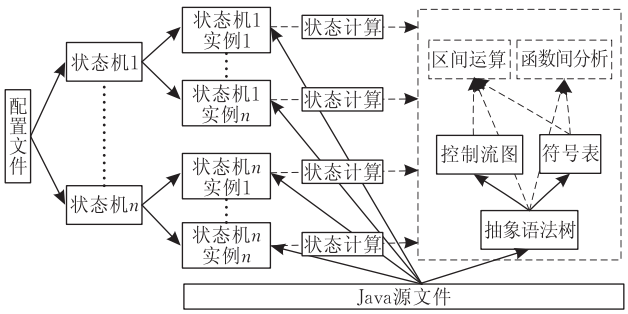


图 4 DTS 测试架构

原 DTS 测试框架在获得分析单元的必要信息后,根据既定的缺陷模式及单元内容确定需要创建的状态机实例数量,然后逐一遍历控制流的节点,在每个控制流节点上应用函数间分析和区间运算技术计算所有状态机实例的状态,当遇到缺陷状态时,报告缺陷并退出相关状态机实例.应用异常控制流序列后,相当于在 DTS 的控制流遍历过程中修改如下内容:增加对方法抛出异常的处理,在分析前扩展初始控制流;在初始控制流上检测并保存运行时异常,

① http://www.dtstesting.com/

将新运行时异常应用于对初始控制流的扩展；在扩展后的初始控制流上继续进行运行时异常的检测及控制流扩展的工作，直到没有新异常出现为止. 以上过程将控制流序列的生成算法集成到 DTS 的测试过程中，将在初始控制流上的一次遍历过程转变为在控制流序列上的多次遍历过程.

6.2　实验结果

实验选取 9 个开源 Java 工程作为测试对象，选取空指针异常模式(NPD)及资源泄漏模式(MLF)作为测试目标，对比应用异常控制流序列前后的过程数据及结果数据. 设定 $DTS(A)$不应用异常控制流序列，$DTS(B)$应用该方法.

对比测试结果如表 1 所示. 状态机类型有 NPD 和 MLF 两类，建立状态机实例的数量共 48766 个，平均需要为每个工程建立 5418 个状态机实例. $DTS(A)$创建的基本控制流共 16809 个，而应用了控制流序列后，$DTS(B)$中将控制流扩展为 18631 个，总增长率为$[C(B)-C(A)]/C(A)\times 100\%=10.8\%$，平均增长率为$\left(\sum_{i=1}^{9}(C(B)_i-C(A)_i)/C(A)_i\right)/9\times 100\%=10.14\%$. 由此带来的开销是 $DTS(B)$的总体分析时间较 $DTS(A)$增长了$[T(B)-T(A)]/T(A)\times 100\%=8.01\%$，平均增长率为$\left(\sum_{i=1}^{9}(T(B)_i-T(A)_i)/T(A)_i\right)/9\times 100\%=9.24\%$.

表 1　对 9 个 Java 开源程序的测试结果对比

工程	文件	代码行	状态机实例	控制流数量		分析时间/s		检测结果(NPD+MLF)	
				$C(A)$	$C(B)$	$T(A)$	$T(B)$	$DTS(A)$ (IP/FP)	$DTS(B)$ (IP/FP)
areca-7.1.1	426	58849	4497	1919	2141	312	341	186/21	201/32
aTunes-1.8.2	306	52603	3210	1252	1393	183	200	136/23	157/42
cobra-0.98.1	449	69537	2894	1240	1271	325	348	35/6	44/14
freecol-0.7.3	343	114838	7563	2304	2754	1191	1358	345/53	374/71
freemind-0.8.1	509	102112	4657	1832	2077	818	911	245/18	259/27
jstock-1.0.4	165	38139	3490	978	1017	364	381	51/3	64/14
megamek-0.32.2	535	177512	16823	4719	5151	5679	6032	363/25	389/47
robocode-1.6	233	53408	3068	1472	1645	234	267	185/33	201/48
SweetHome3D-1.8	154	59943	2564	1093	1182	238	255	116/10	131/25

时间开销的增加是应用控制流序列技术的一个代价，其另外一个代价是误报率的增加. 表中的 IP 表示总检测记录数，FP 表示经人工判断后的误报数. $DTS(A)$的平均误报率为$\left(\sum_{i=1}^{9}(FP(A)_i)/IP(A)_i\right)/9\times 100\%=14.14\%$，而引入控制流序列后，$DTS(B)$的平均误报率增长为$\left(\sum_{i=1}^{9}(FP(B)_i)/IP(B)_i\right)/9\times 100\%=20.09\%$，较 $DTS(A)$增长了 5.95%. 引入误报的主要原因在于若 $ES[I]$中出现了误报的异常，则会在 $RS[I+1]$中引入不可达路径，由此可能导致一系列的误报出现.

虽然这种方法增加了时间开销及误报率，但是它带来的效果是增加了缺陷的报告，将 $DTS(A)$报告的 1470 个缺陷增长为 1500 个，平均增报率为$\left(\sum_{i=1}^{9}(D(B)_i-D(A)_i)/D(A)_i\right)/9\times 100\%=2.18\%$，其中 $D(X)_i=IP(X)_i-FP(X)_i$. 经人工分析后，由于未捕获运行时异常，其中有 8 个缺陷发生后程序会异常终止，而剩下的 22 个缺陷发生后则不会出现这种情况，即它们可以较深地隐藏在程序的执行过程中. 虽然此类缺陷的数量不是很大，但是通过与其它同类测试工具对比，新发现的缺陷在其它工具的报告中是未曾出现的. 这也是这种方法的价值所在——以可承受的代价尽可能多的发现软件中存留的问题，这点在航空、航天等高可信领域尤为重要.

7　相关工作比较

运行时异常对软件静态测试提出了新的问题，本文提出的应用异常控制流序列的方法结合了运行时异常行为与控制流扩展间的迭代关系，具有如下优点：

(1) 扩展传统异常控制流，将抛出运行时异常的边加入其中，在此基础上支持对新异常及缺陷的检测；

(2) 在静态测试过程中，通过迭代方程将在初始控制流上的一次遍历过程转变为在控制流序列上的多次遍历过程.

目前有许多文献都涉及了异常处理机制对控制流、数据流或静态测试的影响,有些文章也包括了有关运行时异常的内容,主要有如下几种:

Fu 等人[9]在对网络服务应用进行健壮性测试时,综合考虑了 Java 程序中的显式异常及 JDK 库所抛出的隐式检查型异常;Mao 等人[10]在研究面向对象程序的健壮性和可靠性时,在 C++异常处理模型的基础上,提出了一种应用静态异常分析方法提高程序的健壮性.以上研究虽然考虑到不同语言中的显式异常和隐式异常,但会由于缺少对运行时异常分析而遗漏某些特殊路径,从而影响程序的健壮性.

Sinha[11-12]等人在进行程序分析的过程中,描述了异常发生及传播机制,从过程内和过程间两个方面介绍了异常处理对控制流的影响,并给出异常处理机制对构建测试及覆盖技术的影响,但在他们所提出的异常处理结构和标准中仍然缺少与运行时异常相关的内容,因此难免影响测试覆盖的充分度.

为了能够充分有效地支持程序分析,Choi 等人[13]提出一种控制流表示方法,叫做要素控制流图 FCFG 方法.该方法将程序中潜在抛出运行时异常的语句(PEI)都加入到控制流中.虽然 FCFG 方法可以处理运行时异常,但由于潜在异常抛出点都是预先确定的,而并非通过计算求得,因此采用此技术的静态测试方法必然会出现较高的误报率.

Weimer 等人[14]提出一种数据流分析方法来检测由于异常处理不当导致的资源泄漏问题,该方法包含了传统显式、隐式及运行时异常对控制流的影响以及由此导致的资源泄漏问题.他们的方法虽然考虑了运行时异常,但仅限于过程内的影响,未涉及运行时异常在过程间的传播,且仅限于研究运行时异常的直接影响,其影响对控制流的扩展以及在扩展后的控制流上是否能出现新的异常则没有研究,因此采用此技术的静态测试方法可能会出现一定的漏报情况.

Chatterjee 等人[15]为面向对象程序提供了一个异常处理的模型,可以确定某个运行时异常在过程间传播后可以达到什么程序位置,同时开发了一个静态分析工具 Jex 处理本地异常和全局异常.他的方法同样也由于没有考虑运行时异常与控制流扩展之间的交替影响而出现漏报情况.

8 结束语

本文提出一种迭代的代码缺陷检测方法,将运行时异常对控制流的扩展及在扩展的控制流上检测新运行时异常这两个迭代的过程相结合,用于研究运行时异常对软件静态测试的影响.实验结果表明,由于控制流的扩展,会导致分析时间的成倍增长;另外在生成控制流序列的过程中还会引入部分不可达路径,这使得误报率也有所增长.但以上代价所带来的效果是深入地检测出传统方法所遗漏的缺陷,这些遗漏的缺陷在航空、航天、电信、金融等高可信领域具有重要的价值.因此,对于如何提高分析效率以及如何降低误报率将是以下一步研究的主要内容.

参 考 文 献

[1] Ball T, Rajamani S K. The slam project: Debugging system software via static analysis//Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, 2002: 1-3

[2] William R B, Jonathan D P, David J S. A static analyzer for finding dynamic programming errors. Software-Practice & Experience, 2000, 30(7): 775-802

[3] Ryder B G, Tip F. Change impact analysis for object-oriented programs//Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Utah, 2001: 46-53

[4] Hallem S, Chelf B, Xie Y, Engler D. A system and language for building system-specific, static analyses//Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation. Berlin, 2002: 69-82

[5] Dawson E, Benjamin C, Andy C et al. Checking system rules using system-specific, programmer-written compiler extensions//Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation. San Diego, 2000: 1-16

[6] Yang Zhao-Hong, Gong Yun-Zhan, Xiao Qing et al. The application of interval computation in software testing based on defect patterns. Journal of Computer-Aided Design & Computer Graphics, 2008, 20(12): 1630-1635(in Chinese)
(杨朝红,宫云战,肖庆等. 基于缺陷模式的软件测试中的区间运算应用. 计算机辅助设计与图形学学报, 2008, 20(12): 1630-1635)

[7] Wang Ya-Wen, Gong Yun-Zhan, Xiao Qing et al. Variable range analysis based on interval computation. Journal of Beijing University of Posts and Telecommunications, 2009, 32(3): 36-41(in Chinese)
(王雅文,宫云战,肖庆等. 扩展区间运算的变量值范围分析技术. 北京邮电大学学报, 2009, 32(3): 36-41)

[8] Xiao Qing, Gong Yun-Zhan, Yang Zhao-Hong et al. Path sensitive static defect detecting method. Journal of Software, 2010, 21(2): 209-217(in Chinese)

(肖庆，宫云战，杨朝红等. 一种路径敏感的静态缺陷检测方法. 软件学报，2010，21(2)：209-217)

[9] Fu C，Milanova A，Ryder B G et al. Robustness testing of Java server applications. IEEE Transactions on Software Engineering，2005，31(4)：292-311

[10] Mao C Y，Lu Y S.，Improving the robustness and reliability of object-oriented programs through exception analysis and testing//Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems. Shanghai，2005：432-439

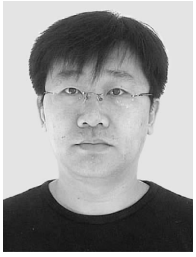
[11] Sinha S，Harrold M J. Analysis and testing of programs with exception-handling constructs. IEEE Transactions on Software Engineering，2000，26(9)：849-871

[12] Sinha S，Harrold M J. Criteria for testing exception-handling constructs in Java programs//Proceedings of the International Conference on Software Maintenance. Los Alamitos，1999：265-274

[13] Choi J D，Grove D，Hind M et al. Efficient and precise modeling of exceptions for the analysis of Java programs//Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York，1999：21-31

[14] Weimer W，Necula G C. Exceptional situations and program reliability. ACM Transactions on Programming Languages and Systems，2008，30(2)：8-51

[15] Chatterjee R，Ryder B G，Landi W A. Complexity of concrete type-Inference in the presence of exceptions. IEEE Transactions on Software Engineering，2001，27(6)：481-512



JIN Da-Hai，born in 1974，Ph. D.，lecturer. His research interests include software testing and software static analysis.

GONG Yun-Zhan，born in 1962，Ph. D.，professor. His research interests include software testing and fault tolerant computing.

YANG Zhao-Hong，born in 1976，Ph. D.，associate professor. His research interests include software testing and software static analysis.

XIAO Qing，born in 1979，Ph. D. candidate. His research interests include software testing and software static analysis.

Background

When a runtime exception occurred during the execution of program，an unexpected executing logic maybe appeared due to the modified of control flow dynamically. Thus，a great challenge，detecting the defect caused by the unexpected executing logic，will be brought to software static testing. This paper is to improve the adequacy of defect detection in software static testing. There are some international researches that combine the runtime exception analysis with software static testing. Such as，factor control flow graph was proposed by Choi，which added the statements that will throw potential runtime exception into control flow，while the scale of control flow and false positive will be increased. Weimer focused on the directly effect of runtime exceptions to the resources leak，and Chatterjee focused on the transmission of runtime exceptions among procedures. However，it has not been researched whether other runtime exceptions will be happened based on an occurred one，and how about the false negative of traditional static testing.

This paper propose an approach for iterative defect detection in the presence of runtime exception，in which defect detection and control flow extension are interleaved and performed repeatedly. Experimental results show that，although sequence of control flow with runtime extension increases the time cost and false positive，more defects can be detected than ever，and these defects are missed by other method or tools indeed.

Defect Testing System (DTS) is sponsored by the National High Technology Research and Development Program (863 Program) of China (2009AA012404)，the National Natural Science Foundation of China (91018002). DTS analyze source code of C，C++ and Java，detect possible error and potential risks in runtime by static method. It covers the path that traditional method of dynamic testing is difficult to cover，and collect relevant information to detect the defect. It is a necessary complement to traditional testing methods，particularly important in the high confidence field. In order to improve the accuracy of static analysis，such technique as sensitive path analysis，effective path analysis，sensitive context analysis，effective context analysis are used，and the sequence of control flow with runtime extension is a part of path analysis.