ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.5, May 2010, pp.930−941
doi: 10.3724/SP.J.1001.2010.03544

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于状态机模型的构件健壮性测试*

雷 斌[1,2], 王林章[1,2], 卜 磊[1,2], 李宣东[1,2+]

[1](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

[2](南京大学 计算机科学与技术系,江苏 南京 210093)

Robustness Testing for Components Based on State Machine Model

LEI Bin[1,2], WANG Lin-Zhang[1,2], BU Lei[1,2], LI Xuan-Dong[1,2+]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

+ Corresponding author: E-mail: lxd@ nju.edu.cn

Lei B, Wang LZ, Bu L, Li XD. Robustness testing for components based on state machine model. *Journal of Software*, 2010,21(5):930−941. http://www.jos.org.cn/1000-9825/3544.htm

Abstract: This paper defines robustness based on a formal semantics of component system, and proposes a testing framework to detect robustness problems. The approach is implemented in a tool named RoTesCo (robustness testing for components). It traverses the state machine of the component under testing to generate paths, which cover all the transitions. Firstly, the call sequences following the paths drive the component to different states. Secondly, it feeds method calls with invalid parameters or inopportune calls to the component. The test oracle is automated by distinguishing different types of exceptions. RoTesCo is evaluated on a benchmark of real-world components from widely-used open source projects and it has produced encouraging results.

Key words: software testing; component; robustness; state machine; invalid input

摘 要: 基于形式化的构件语义定义了健壮性,并提出一种基于状态机的构件健壮性测试方法.基于该方法实现了原型工具 RoTesCo.首先遍历状态机生成一组覆盖所有转换的路径,基于这些路径的测试用例驱动构件发生状态转换;然后用无效输入和不当调用在构件的不同状态来测试其健壮性.通过区分测试中捕获异常的类别,自动报告健壮性错误.以通用的开源项目构件组成评测平台,实验数据显示,RoTesCo 的测试效率比已有的算法表现得更优越.

关键词: 软件测试;构件;健壮性;状态机;无效输入

中图法分类号: TP311 **文献标识码**: A

基于构件(component)的软件开发用已有构件直接构建软件系统,以提高复用率,缩短软件开发周期.商业构件体系结构包括 EJB(enterprise JavaBeans)[1]和 COM(component object model)[2]等.构件开发给测试带来了新的挑战.构件封装特定功能,并通过接口(interface)提供服务.构件通过形如 $pre \Rightarrow post$ 的契约(contract)来定义服务.

* Supported by the National Natural Science Foundation of China under Grant Nos.90818022, 60603036, 60721002 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2007AA010302 (国家高技术研究发展计划(863)); the Jiangsu Provincial Natural Science Foundation of China under Grant No.BK2007139 (江苏省自然科学基金)

Received 2008-08-26; Accepted 2008-11-28

即,如果服务在前置条件 *pre* 为真时被调用,构件保证执行正常终止,并有终止时后置条件 *post* 为真[3].通常,构件不绑定特定的调用代码,因此在复用构件时,很难避免无效输入(invalid inputs),即违反前置条件的输入.脆弱的构件可能会因之失效;当构件能够有效处理无效输入时,我们称其为健壮(robust)的构件.

以计算器中的除法功能为例:*Method*: *divide* (*x*: *int*, *y*: *int*): *float Contract*: $y\neq0\Rightarrow return=x/y$.

该契约定义,当 $y\neq0$ 时,方法 *divide* 返回结果 x/y.功能性测试人员用满足契约的测试用例检查返回值是否符合预期,但他们往往不检查 $y=0$ 时该构件的行为.忽略除零问题可能导致运行时刻抛出"浮点异常",这种小错误在关键性任务系统却可能成为致命缺陷.健壮性测试者则用如{MAX_*INT*,−1,0,1}等特殊值检验构件是否能够有效处理.当然,设计者可以把 *Contract* 精化为 *Contract*′:$(y\neq0\Rightarrow return=x/y)\wedge(y=0\Rightarrow return=0)$.这时,功能性测试覆盖了部分健壮性测试用例.但无效输入可能是全输入域的一个相当复杂的子集.而契约设计者应致力于描述构件的功能,要求他们考虑所有的输入情况并不现实.因此,专门的健壮性测试可以弥补功能性测试的不足,提高构件的可靠性.

健壮性工具如 Ballista[4]和 Jcrasher[5]用预设的无效输入对应用程序接口(application programming interface,简称 API)进行测试.设计者分别用 UNIX Shell 程序和 Java 类作了实例分析,并检测出相应的健壮性缺陷.它们的测试目标是 API,并没有考虑软件所处的状态.Ballista 提供不同的参数组合,重复测试一个方法;JCrasher 初始化一个对象,调用某方法,在下一次调用之前恢复对象到初始值.然而,构件通常具有多个状态,某些软件缺陷只在特定状态下或者经过一定的状态转换后才会被激发.因此,要把这些工具直接应用到构件系统的健壮性测试则很难.我们针对构件的健壮性,尝试引入状态信息,以提高健壮性测试的效率.

另一方面,学术界提出了许多基于有穷状态机(finite state machine,简称 FSM)的测试方法,定义了多种覆盖准则和算法为单元测试和集成测试服务[6−9],但它们只针对功能性需求进行测试.

我们将二者结合,提出一个基于状态机的构件健壮性测试框架,先用有效输入序列驱动构件到达不同状态,然后用无效的输入值或不当调用检验其健壮性.本文有如下贡献:

- 基于构件语义模型定义了健壮性,该定义制导测试用例生成以及测试结果的分析.
- 在健壮性测试中引入状态机模型,获得了比不考虑状态的健壮性测试算法更高的效率.

本文第 1 节定义构件及其健壮性.第 2 节给出基于状态机的构件健壮性测试框架.第 3 节是原型工具和实例分析.第 4 节比较相关工作.最后作总结.

1 构件及其健壮性

1.1 构 件

构件是个抽象概念.在实现层次它可以是 EJB[1]中的一个 JavaBean 类,也可以是面向服务架构中用 WSDL 描述的一个服务[10].但下述观点是对构件的共识[11]:它可以被复用;并提供显式的功能规约.语义上,构件是一个包含提供接口(provided interfaces)和需求接口(required interfaces)的软件单元.

本文测试框架的形式化基础是构件演算[3,12].定义 1~定义 6 来自于构件演算.

定义 1(接口). 一个接口 *I* 是对一组域变量和一组方法的声明,*I*=(*FDec*,*MDec*),其中:

- *FDec* 是一组变量,每个变量形如 *x*:*T*,*x* 为变量名,*T* 为类型;
- *MDec* 是一组方法型构(signature),形如 *m*(*x in*, *y out*)声明了一种方法 *m*,及其输入参数列表 *x* 和输出参数列表 *y*,输入输出参数中每个参数为 *p*:*T*,*p* 为参数名,*T* 为类型.

如某环形缓冲(circular buffer)构件,其接口 *CB_IF* 包含如下定义:

FDec={*size*:*Integer*};*MDec*={*write*(*buf*:*char*[],*off*:*int*,*len*:*int*):*int*;*read*(*buf*:*char*[],*off*:*int*,*len*:*int*);*clear*()},

其中,*size* 是一个整型变量;方法 *write* 有 3 个参数,*off*:*int* 为整型,末尾的 *int* 表示返回值的类型.

定义 2(构件). 一个构件 *C* 是一个多元组,*C*=(*I*,*Init*,*MDec*,*MCode*,*PriMDec*,*PriMCode*,*InMDec*),其中:

- *I* 是一个接口,列出构件提供的方法列表;*Init* 是初始化命令;
- *MCode* 将 *MDec* 中的公共方法映射为卫式命令(guarded command),*PriMCode* 将 *PriMDec* 中的私有方法

映射为卫式命令;

- *InMDec* 是一组该构件依赖的服务.

构件系统中服务的行为我们用设计(design)来描述.可以将设计与对象系统中的公共方法相对应.

定义 3(设计). 一个设计为 $D=(\alpha,P)$,其中:α为一组状态变量;P 为断言 $ok\wedge p(x)\Rightarrow ok'\wedge R$,写作 $p\vdash R$,表示如果某服务在满足前置条件 $p(x)$时被调用,则调用会终止,而且终止时满足后置条件 $R.ok$ 和 ok'分别表示程序的正常启动和终止.

基于前置条件,我们给出健壮性测试第 1 种测试用例,即当 p 不真时,测试构件是否出错.

我们用反应式设计(reactive design)来定义反应式程序的语义,引入状态变量布尔值 $wait.wait'$为真,代表一个程序处在挂起状态;否则,可以被调用.$wait$ 和 $wait'$分别表示调用前和调用后程序是否挂起.

定义 4(反应式设计). 设计 D 是反应式设计,当且仅当它是函数 H 的不动点(fix point),即 $H(P)=P$ 时,有:

$$H(P)=_{df}(true\vdash wait'\wedge v=v')\lhd wait\rhd P,$$

其中,$P\lhd b\rhd Q=_{df}((b\wedge P)\vee(\neg b\wedge Q))$.

定义 5(卫式设计). 对卫式条件 g 和设计 $D=(\alpha,P)$,我们用卫式设计 $g\&D$ 来描述构件的反应式行为

$$(\alpha,P\lhd g\rhd(true\vdash wait'\wedge v'=v)),$$

即当卫式条件 g 不真时,程序保持在挂起状态;否则,提供 D 所定义的服务.

卫式设计保证了环境调用方法和执行方法的同步,只在 g 为真与 *wait* 非真同时满足时才发生.基于卫式设计,我们给出健壮性测试第 2 种测试用例,即当 g 不真时,测试构件是否出错.

定义 6(契约). 一个契约 *Ctr* 是一个四元组 $Ctr=(I,Init,Spec,Prot)$,其中:

- I 是一个接口,*Init* 是其初始条件;
- *Spec* 用卫式设计(guarded design)描述其中每一种方法;
- *Prot* 是接口使用者应遵循的交互协议.

本文中,我们使用状态机(state machine)系统地描述构件的卫式设计.对处于某状态 s 的构件来说,只有从 s 出发的转换所代表的服务可用.对某契约 *Ctr*,可以定义它的状态机 *SM*:

定义 7(状态机). 一个状态机 $SM=(S,S_0,\Sigma,Guard,Tran)$,其中:

- S 是一个状态的有穷集,$S_0\in S$ 是初始状态,*Guard* 是卫式条件的有穷集合;
- Σ是输入符号的有穷集,在构件的语境下代表构件接口的公共方法,即

$$\Sigma\subseteq Ctr.I.MDec;$$

- $Tran\subset S\times\Sigma\times Guard\times S$ 是状态转换的集合.

图 1 是前文中环形缓冲例子的接口的状态机,其中:*len* 是方法 *read* 和 *write* 的参数;*available* 和 *capacity* 为状态变量,分别表示构件有多少字符可读和有多少空间可写.

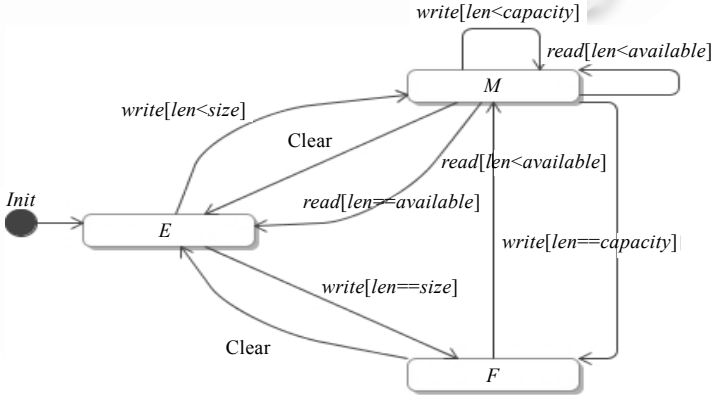


Fig.1 State machine of the circular buffer component

图 1 环形缓冲的状态机

1.2 健壮性

在理想的构件系统中,调用方代码应满足前置条件并处理契约定义的错误.但因为构件的使用方和提供方之间的信息不对称,构件复用存在很大的不确定性.如果设计时没有考虑无效输入,构件有可能会失效.IEEE 将健壮性定义为:在无效输入或压力环境下,系统或者构件能正常工作的程度[13].本文的工作专注于无效输入.首先从构件契约角度定义恰当的调用.

定义 8(调用恰当). 对状态机 SM 及其当前状态 s,$s\in SM.S$,一个方法 m 调用恰当,当且仅当

$$\exists s'\in SM.S,\ g\in SM.Guard,\ g\wedge((s,g,m,s')\in SM.Tran),$$

即至少有 1 个 s 的出边为 $g\&m$,且 g 为真.否则,m 为不当调用.

可能导致健壮性错误的方法调用有:

$\boldsymbol{Iv_1}$:无效输入,是指方法调用带有违反前置条件的参数;

$\boldsymbol{Iv_2}$:不当调用(inopportune),是指方法调用在状态机模型上没有对应的状态转换.

定义 9(理想的构件). 构件 C 为理想的,是指对任意方法 $m\in C.MDec$ 及其相对应的接口设计 $g\&(p\vdash R)$,m 实现的卫式命令 $g_m\rightarrow c_m$ 满足 $\begin{Bmatrix}(g\wedge pre)\Rightarrow post;\\(g\wedge\neg pre)\Rightarrow v'=v\wedge\neg wait'\\(\neg g)\Rightarrow wait'\end{Bmatrix}$,即 Iv_1 不改变状态,Iv_2 挂起构件.

理想构件是构件开发的目标,但对于测试而言,这个标准过高.另外,构件状态变量不一定能够全部被测试脚本观察到.我们适度放宽标准,定义比较容易测试的属性,即健壮构件.

定义 10(健壮的构件). 构件 C 为健壮的,是指对任意方法 $m\in C.MDec$ 及其相对应的接口设计 $g\&(p\vdash R)$,m 实现的卫式命令 $g_m\rightarrow c_m$ 满足 $\begin{Bmatrix}(g\wedge\neg pre)\Rightarrow ok'\\(\neg g)\Rightarrow wait'\vee ok'\end{Bmatrix}$,即 Iv_1 和 Iv_2 可能改变系统的其他状态变量,但 Iv_1 使得构件正常终止,Iv_2 或挂起构件或正常终止.两者都不会使构件返回错误信息.

ok'与 $wait'$都是形式化框架下的状态变量,其中,ok'表示服务正常停止,$wait'$表示服务被挂起.虽然停机和死锁问题都不可判定,但在具体测试平台(如 Java),观察到的现象却可以确定 ok'与 $wait'$非真.文献[14]定义了状态变量 $crash$,及健壮性构件接口的卫式设计为$(\alpha,((true\vdash post)\lhd pre\rhd\neg crash)\lhd g\rhd\neg crash)$,即 Iv_1 和 Iv_2 都无法使构件 $crash$ 变量为真,但是 $crash$ 本身的语义需要测试人员根据具体平台自定义.相比之下,本文清晰地界定了无效输入和不当调用发生时健壮的组件应满足的属性,并为定义构件失效、实现测试结果的自动分析提供了依据.

基于 Java 异常处理,可以定义构件健壮性失效如下:

定义 11(构件健壮性失效). 在 Java 平台上的构件系统发生健壮性失效,当且仅当如下异常被检获:

(1) 未处理的异常(unhandled exception),抛出后将向栈逐级传递,直到有一层截取(catch)这个异常(如图 2 所示),构件及调用方都被挂起.某调用方在使用构件 C,调用方法 m_1 时,抛出异常 e,而 $After$ 中没有 catch 语句匹配 e,便逐级往栈底传递,直到被其中一层的 catch 语句截取.最底层 Java 虚拟机将截取所有异常.如果不考虑 finally 关键字,代码 $Before;C.m_1();After$ 的执行语义与 $Before;C.m_1()$相同.如果 C 的生命域在异常处理层之上,C 将被从栈中清除.即使在异常处理之后仍然可见,也无法保证 C 的状态变量是否仍满足不变式(invariant),因此都有理由认为构件进入失效状态.在测试过程中应该发现这种情况,并通过调试予以消除,否则将为复用该构件的系统留下隐患.

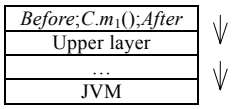


Fig.2 Exception propagation

图 2 异常传递

(2) 表示构件已经失效的程序员自定义的异常.通过定义和抛出显式的异常,程序员可以对构件失效状态进行防御式编程(defensive programming),将自定义的构件失效信息传递给调用方.

因此,只要发现构件发生健壮性失效,其服务没有停机,而且没有死锁,就可以推断该构件不满足健壮性定义.本文根据这个结论,提出构件健壮性测试框架,检测健壮性失效.

2 测试框架

图 3 显示了框架的工作流,其中,圆角节点为测试行为,方角节点为测试的输入和输出的工件(artifacts).右上方被测构件包含契约模型与实现;预设输入池需要测试人员手工设置,本文实现的原型工具提供了一个默认设置.本节给出测试框架的细节:首先,基于转换覆盖分析被测构件的状态机上的路径;根据路径生成有效输入序列以驱动构件到达不同的状态;无效输入调用或不当调用被添加到每个测试用例的最后,形成健壮性测试用例;对截获的异常进行分类,并报告健壮性问题.

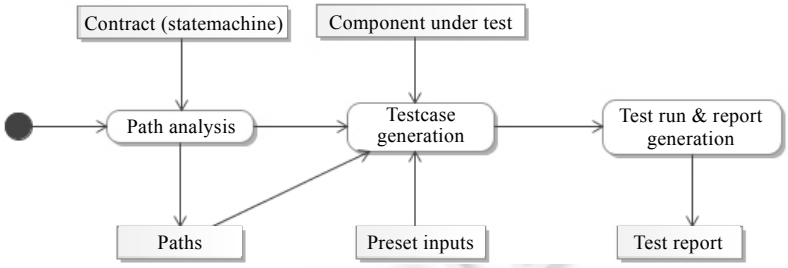


Fig.3 Activity diagram of the testing process

图 3 测试流程的活动图

2.1 路径分析(path analysis)

每种健壮性错误都有相应的触发条件,如 JVM 只在如下情况抛出异常 *ClassCastException*:形如(*B*)*a*;的映射语句将对象 *a* 显式转换为 *B* 的实例,而 *a* 实际上并不是 *B* 的对象.如图 4 所示,构件 *C* 有如下方法:*init* 赋一个新值到状态变量 *a*;*evolve* 把状态变量 *state* 赋为真值;*cast* 返回 *a* 的引用.图 4 右栏是一组测试构件 *C* 的调用序列.分析显示,仅当出现下列子串时才会抛异常:第 1 个 *evolve* 调用后面紧跟 *cast*,如子串 4 和子串 6.无状态的 API 测试并不能发现该错误,因为从初始状态调用任何单一的方法都不会触发这个缺陷.

```
class B extends A …
class C{
   A a=null;
   boolean state=false;
   void init(){
      if (!state) a=new A();
      else a=new B();
   }
   public void evolve(){state=true;}
   public B cast(){
      if (!state) return null;
      else return (B)a;
   }
}
```

Call Sequences:
√ 1. *cast*;
√ 2. *init*;
√ 3. *init*; *cast*;
× 4. *evolve*; *cast*;
√ 5. *evolve*; *init*; *cast*;
× 6. *init*; *evolve*; *cast*;

×: Raise exception
√: Pass the test

Fig.4 An example of robustness defect

图 4 健壮性缺陷示例

因此,我们用满足一定覆盖度的测试用例集(test suite)来遍历不同的状态和转换,以提高发现健壮性错误的可能性.常见的覆盖准则包含全节点(all-nodes)、全转换(all-transitions)[15]、MC/DC(modified condition/decision coverage)[16]等.基于复杂度和覆盖能力的折衷,我们选择全转换覆盖,它保证了所有节点以及转换都被至少遍历 1 次.与其他覆盖准则如 MC/DC 相比,全转换准则的分析算法复杂度较低.图 5 是基于广度优先的路径分析算法.算法的输出是一组路径,每条路径形如 $\rho = Init \longrightarrow S_0 \xrightarrow{m_1[g_1]} S_1 \xrightarrow{m_2[g_2]} \ldots \xrightarrow{m_n[g_n]} S_n$,$m_x$ 是接口中的一个方法,g_x 是状态转换之上的卫式条件.

```
Input: State machine SM=(S,S0,Σ,Guard,Tran).
Output: PathList, the set of generated path, each is a sequence of
        states and transitions.
Intermediate data: vq, queue to store the current states, in which
                   visited nodes are not stored again.
Begin
Vq.add(S0);
While (vq is not empty){ //visit all nodes
   Vertex sv=vq.head;
   For (Transition t: sv's outgoing edges) {
      updateTransition(t);
      vq.add(t.target);
   }
}

//add remaining transitions
for (SM's transitions: Transition t)
If (t.from is reachable and t is not covered by PathList){
   Select a random path p, p's last node is t.from;
   new Path p'=p.append(t.from).append(t);
   PathList.add(p);
}
end
Function updateTransition(Transition t){
   If (t.from==S0)
      new Path p from init;
   else {
       Select a random path p, p's last node is t.from;
       new Path p'=p.append(t.from).append(t);
   }
   PathList.add(p);
}
```

Fig.5 Traverse algorithm for paths of all-transitions coverage

图 5 全转换准则(all-transitions)路径生成算法

2.2 生成健壮性测试用例(robustness test case generation)

以上述路径为基础生成健壮性测试用例,试图触发构件的失效.每个测试用例由如下调用序列组成:

(1) 有效方法调用的一个有穷序列,驱动构件从初始状态 *Init* 开始一系列状态转换;

(2) 一个无效输入调用或不当调用,为第 1.2 节定义的 Iv_1 或 Iv_2.

本文实现了 Java 构件的测试用例生成,为转换路径上的每个调用生成实际参数.我们用 Octopus[17]的 Java 元模型构造生成这些对象的代码片段.每个方法调用对应一个 OJOperation 对象,而参数对应 OJParameter 对象.它们可以直接翻译为 JUnit 测试脚本.对有效方法调用的参数表,我们通过递归的方式来生成.给定参数表中的类型 T,调用函数 $generate(T)$:

(1) 如果 T 是简单类型(primitive),返回 T 的一个随机值;

(2) 如果 T 为复杂类型,有构造函数 $T(T_1,T_2,\ldots,T_n)$,则返回 $T(generate(T_1),generate(T_2),\ldots,generate(T_n))$.

在调用 T 的构造函数的语句之前,插入 $generate(T_x)$所返回的语句.当类型 T 具有多个构造函数时,随机选择一个.递归调用当遇到简单类型时停止.同时,为了提高代码生成的效率,我们限定递归深度小于深度值 *depth*,根据经验调整为 5.当达到这个深度时,返回空引用(null).

当某参数由方法调用的前置条件所限制时,随机生成器也相应调整随机区间.本文中,我们用 OCL(object constraint language)来描述这些约束(约束有可能非常复杂.我们实现了较简单的 OCL 约束的求解以解决实际问题,忽略过于复杂的约束),抽取 UML 模型中的 OCL 约束,获得满足约束的随机值.比如在文章开始部分的 divide 例子中,规约为 $y\neq 0$,则随机生成一个在[MIN_*INT*,0)和(0,MAX_*INT*]区间的值.

无效输入调用或不当调用在最后,测试构件的健壮程度.无效输入(Iv_1)与有效输入生成类似,区别在于用无效值替换一个参数.无效值包含违反前置条件的和根据经验常出现在失效测试用例中的参数:

(1) 简单类型,为每个类型手工建立一个数据池.比如,String 类型为{*null*,空串,超长串},整型为{MIN_*INT*,−1,0,1,MAX_*INT*}等.以整形为例,通常选取极端值{MIN_*INT*,0,MAX_*INT*}以及具有代表性的{1,−1}.

(2) 受到 OCL 约束的参数,在约束域外随机取值.比如,整型参数带有约束[0,MAX_*INT*],取随机的负整数.

(3) 复杂类型.取空引用(null)以及在第 1 次递归调用 $generate(T)$时,使用空引用或者无效值作参数.

不当调用(Iv_2),对状态机 *SM*、某状态 s 以及 s 的出边集合 *out*,有:

(1) 不在 s 的任意一条出边上的方法集合,即$\{true\&m|(m\in SM.\Sigma)\wedge(\forall g\in SM.Guard,g\&m\notin out)\}$.

(2) 在 s 某出边上的方法,但所有卫式条件都不真.即$\{(\neg\vee(\Phi(m)))\&m|\exists g\in SM.Guard,g\&m\in out\}$,其中,$\Phi(m)=\{g|g\&m\in out\}$,$\vee(\Phi(m))$是所有调用 m 的出边的卫式条件的析取(disjunction).

2.3 执行和测试结果分析(execution and test analysis)

测试生成的结果为 JUnit 脚本,可以直接编译执行.下图是为 CircularBuffer 构件生成的一个测试用例.其中,

方法名中的 1 和 2 分别代表路径和测试用例的编号.第 4 行是有效调用,第 6 行提供了一个字符数组,而第 7 行为 *write*(*char*[],*int*,*int*)方法的第 2 个参数传入了整型的预设值−1,以测试构件处理无效参数的能力.

```
1.  @Test
2.  public void test_CircularBuffer_1_2() throws BufferException {
3.    /* Init→Empty*/
4.    CircularBuffer CircularBuffer_2=new CircularBuffer(6);
5.    /* Primitive preset: 1*/
6.    char[] char_array_1={'J','t','H','g','k','f','b'};
7.    CircularBuffer_2.write(char_array_1,−1,3);
8.  }
```

Fig.6 An example of test script

图 6 测试脚本示例

对执行的结果,基于定义 11 自动区分不同的异常,判断某测试用例是否触发健壮性缺陷.Java 中,*Throwable* 是所有异常(exception)和错误(error)的父类.异常的类层次如图 7 所示,可分为以下几类:

(1) java.lang.Error:通常由虚拟机抛出,表示出现严重错误,不由程序员抛出或处理.

(2) 未检异常(unchecked exceptions):RunTimeException 的子类,程序可以直接抛出未检异常,无须在函数声明中列出.未检异常可能表示程序契约被破坏.如 *String.substring*(*int begin*, *int end*)中,当参数 *begin* 为负值时,抛出 ArrayIndexOutOfBoundsException.构件实现中的 Bug 也可能抛出未检异常.文献[5]对未检异常作了更详细的分类,但本文基于类层次的测试用例生成,可以避免文献[5]中第 2 组异常情况的出现.因此,我们只要观察到未检异常,都认为构件出现失效.

(3) 检验异常(checked exception):程序定义的非 RunTimeException 的子类,反映运行结果或转移控制流.

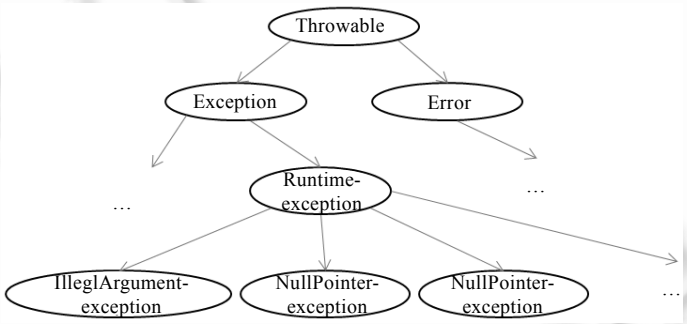


Fig.7 Class hierarchy of Java exceptions

图 7 Java 异常的类结构

构件健壮性测试的基准基于对异常的分析.当测试工具获取到下述异常现象时,报出健壮性错误,表示该构件已经失效.其调用栈可以用来进行调试.相应测试用例可重现软件失效.

(1) 未检异常:可能是构件内部的缺陷,也可能是未对传入参数作足够的检查.但两种情况都表明构件处理无效输入的能力有待改进.

(2) 失效标志异常:检验异常还可以用来在调用方和构件之间传递执行信息.比如,POP3 客户端构件可能会返回异常,表明网络连接错误.在对已有构件反向工程时,将它们与表示构件失效的异常相区别.我们在测试框架中提供扩展点,使测试人员可以设定特别的异常.

3 工具实现和实例分析

3.1 工具概述和实验设计

为了检验测试框架,我们实现了原型工具 RoTesCo.它包含下面几个功能模块:路径分析(path analyzer),遍历状态机并生成一组路径;测试生成(test case generator),基于路径生成可执行的健壮性测试用例;异常分析模块(exception analyzer),区分测试结果,生成测试报告.图 8 是测试用例生成模块的软件产品类图.构件的核心是接口定义,描述构件提供的服务.

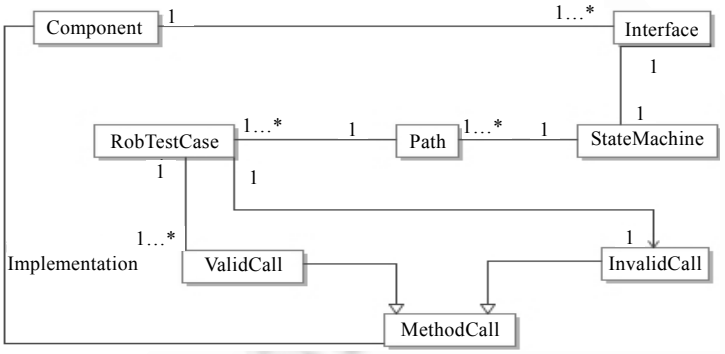


Fig.8 Class diagram of the robustness testing tool

图 8 健壮性测试工具生成的类图

RoTesCo 的实现基于一组软件库:模型分析模块接受基于 EMF[18]标准的 UML 文件,可以来源于 rCOS 建模工具[3]、MagicDraw 或其他支持 UML2 标准的工具;OCL 分析模块基于 Dresden[19]的类库进行扩充;利用 Java 反射机制[20]对构件接口进行分析;并使用 Octopus[17]的 Java 元模型生成 JUnit 测试用例.

如表 1,我们选取了一组开源软件的构件进行测试,见表 1.其中,CircularBuffer 来自 Kaffe Java 虚拟机[21];从 Apache 通用库中选取了 4 个容器构件;从 GNU ClassPath 项目中选择 3 个输入、输出流构件;最后是 GNU 网络库中 IMAP 和 POP3 协议客户端构件.除 IMAP 和 POP3 的文档中包含非形式化的状态机之外,其他构件都通过阅读文档及代码注释,进行手工逆向工程获取状态机模型.

Table 1 Benchmark

表 1 实验平台

Component	Source	States	Transitions	Methods	LOC	Paths
CircularBuffer	Kaffe	4	10	3~10	83	10
ArrayStack	Apache Commons	3	9	8~10	71	9
BinaryHeap	Apache Commons	4	19	7~19	188	19
FastArrayList	Apache Commons	5	33	10~85	792	33
FastHashMap	Apache Commons	5	21	7~47	422	19
BufferedInputStream	GNU Classpath	4	16	8~10	122	14
BufferedWriter	GNU Classpath	4	10	5~7	81	10
StringBufferInputStream	GNU Classpath	3	7	5~6	47	7
IMAPConnection	GNU inetlib	5	31	44~67	3 208	31
POP3Connection	GNU inetlib	4	13	14~18	528	14

3.2 环形缓存(circular buffer)实例

首先以环形缓冲构件(简称 CB)为例说明测试效果.该构件提供一个缓存,用户可以写入(write)、读出(read)字符或者清空(clear).除 *Init* 状态外,状态机还包含 3 个状态.该缓存的实现基于两个指针(整型数)*in*,*out* 和一个字符数组.使用者只需提供写入、读出的参数,具体实现对用户透明.构件定义了如下检验异常:

- BufferBrokenException,表示构件已经损坏;

- BufferOverflowException,在调用方试图写入超过缓存容量的字符时抛出;
- BufferUnderflowException,在试图读出超过实际存储的字符数时抛出.

RoTesCo 生成了 10 条路径以覆盖所有的转换,并生成每个状态上的无效输入.路径和无效方法调用组合生成了 114 个测试用例.运行后,发现了两种类型的程序失效:自定义的 BufferBrokenException 及未检异常 NullPointerException 和 ArrayIndexOutOfBoundsException.经过调试我们发现,构件设计者乐观地假定传入的参数很理想.比如,从 CircularBuffer.java 的 100 行抛出 NullPointerException 的语句 *System.arraycopy*(*buffer*,*out*,*buf*,*off*,*maxreadlen*)被调用前并没有对参数合理性的检查.BufferBrokenException 揭示了构件的一个隐藏缺陷.调用栈帮助定位到第 58 行,返回缓存中实际存储的字符数 *available*:当两个指针顺序颠倒,即 *in* 比 *out* 小时,返回一个大于缓存最大容量 *size* 的值.基于该方法的调用可能会覆盖掉已有数据,调用方不会得到错误信息.

3.3 实验结果及评价

为了与已有方法进行比较,我们实现了基于状态机的功能性测试算法(stat-based functional,简称 SF)以及纯随机的健壮性测试算法(fuzzy testing,简称 Fuz)[5].SF 的测试用例是健壮性测试用例的第 1 部分,即有效调用序列.当路径足够长时,SF 能够发现 CB 中指针倒置的错误.但因为只关注功能性问题,不能发现因为无效参数调用激发的错误.

Fuz 对无状态的对象用随机和预设的无效输入值进行测试.它测试 CB 发现了 NullPointerException,但另一错误未被激发,因为所有的测试都基于无状态的空缓冲区.

我们分别用这 3 种方法测试表 1 中的 10 个构件,结果见表 2.一共 4 大列数据,包括所有测试用例、产生失效的测试用例、未检异常数和失效标志异常数.每列中分别有 3 小列分别为 RoTesCo,SF 和 Fuz 方法的实验结果,用 R,S 和 F 来标注,表示这项数据在 RoTesCo,SF 和 Fuz 方法中的值.如构件 CB 的第 2 列 114,10 和 8 分别表示 3 种算法生成的测试用例数为 114,10 和 8 个.第 4 大列和第 5 大列分别是测试脚本捕获的未见异常和检验异常的数目.其中,列 A 来标注所有(all)方法截取的异常数的总和.3 种算法所检获 CB 构件的所有未检异常的数目为 4.

Table 2 Experimental results

表 2 实验结果

Component	All test cases			Failed test cases			Unchecked exceptions				Checked exceptions			
	R	S	F	R	S	F	R	S	F	A	R	S	F	A
CircularBuffer	114	10	8	107	3	8	4	0	1	4	2	1	1	2
ArrayStack	48	9	2	29	2	2	5	2	2	6	0	0	0	0
BinaryHeap	33	19	0	14	0	0	1	0	0	1	0	0	0	0
FastArrayList	189	33	8	170	25	8	10	4	2	10	0	0	0	0
FastHashMap	35	20	0	0	0	0	0	0	0	0	0	0	0	0
BufferedInputStream	150	14	8	76	0	4	1	0	2	2	3	0	0	3
BufferedWriter	75	11	4	64	0	4	5	0	2	5	4	0	1	5
StringBufferInputStream	49	7	6	28	0	4	2	0	2	3	0	0	0	0
IMAPConnection	1 008	31	21	902	14	16	2	0	4	4	69	4	2	69
POP3Connection	152	13	18	104	6	8	0	0	0	0	21	2	4	21

图 9 和图 10 更直观地显示了 3 种不同方法在健壮性测试平台上的表现.嵌菱形线、嵌三角形线和嵌圆形线分别代表 RoTesCo,SF 和 Fuz 的数据.横轴为被测构件,纵轴为某方法所测得失效占总失效数的比例,比例越高,说明检测能力越强.最高为 1,即已发现失效都被覆盖;最差为 0,即其他方法检测到失效,该方法未发现.表 2 的数据存在 3 种方法均未测出失效的情况,在图中,我们以中值 1/2 来代替无意义的 0/0,表示对该方法的能力不予置评.从检验健壮性角度来看,RoTesCo 在激发健壮性缺陷、抛出未检和检验异常的能力明显优于 SF 和 Fuz.SF 方法因为侧重于功能性测试,对健壮性的检验能力最弱.Fuz 方法处于两者之间.在有些情况下,如图 9 中的构件 BufferedInputStream 和 IMAPConnection,Fuz 检验到最多的问题.但与 RoTesCo 相比,Fuz 很不稳定.综合而言,RoTesCo 能够高效且稳定地检验构件健壮性.

前文对测试数据作统计分析,但并未对构件的健壮程度作定量分析.第 1.2 节的定义中只区分构件健壮与

否.然而,作为构件及更复杂的构件系统,实际上可以再作细化,通过量化评估对构件系统的健壮性程度(degree)作更精确的评定.这可以作为进一步的工作.

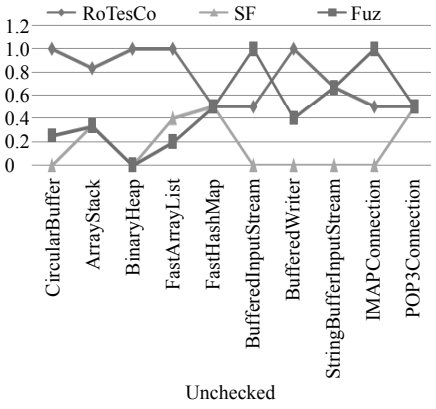


Fig.9 Results diagram of unchecked exceptions

图 9 未检异常结果比较图

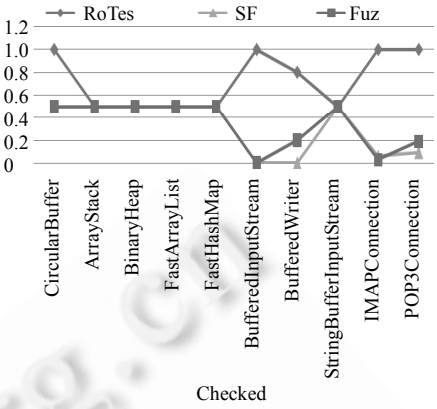


Fig.10 Result diagram of checked exceptions

图 10 检验异常结果比较图

3.4 方法讨论

RoTesCo 完成了测试生成和执行的自动化.测试人员只需提供 UML2 模型文件和接口类,就可以自动生成测试用例,并执行生成的测试驱动.测试基准是隐式的,若当测试脚本截获到特定异常则报错.用户自定义的异常需要手工给出,这一步是可选的,加入之后会使测试更加具有针对性.测试方法的完整性(completeness)受限于测试生成和选择策略.后续工作可以对比不同的覆盖准则,以获取针对健壮性测试更有效的方法调用序列.测试框架不假定具体的构件架构平台,但要将 RoTesCo 移植到其他平台需要重新定义和实现两部分:构件健壮性失效到具体程序现象的映射以及从路径到测试脚本的自动生成模块.

RoTesCo 的不足之处如下:首先,当构件有很多状态时,路径和无效输入直接组合会产生大量的测试用例.类似文献[22]的选择策略,可以用来精简测试用例集,降低测试的时间成本.其次,模型来源也是基于模型测试(model based testing)的常见问题.如果被测构件缺乏状态机模型,则需要测试人员手工提供.这个过程耗时且容易引入错误.但 UML 在研究领域及工业界得到广泛应用,建模已经成为软件开发的主要工作之一.另外,NetBeans,Rational Software Architect 等工具集支持从需求、设计到代码开发的全过程,它们的应用势必产生对测试有用的模型.

4 相关工作

相关的研究可以分为以下 3 类:基于状态机的测试、构件软件的测试以及健壮性测试.

有穷状态机是软件工程领域研究很多的模型.文献[7]用状态机对反应式系统模型检验.Tretmans 等人[9]用输入输出自动机(input-output labeled transition system,简称 IOLTS)生成测试用例.他们关注构件实现与抽象模型之间的功能一致性.

构件软件测试的主要困难来源于构件所提供信息非常有限——它们通常不提供源代码,实现平台也可能与调用方的平台异构.传统的黑盒测试可以对构件进行功能性测试.对于构件间交互,测试可以基于 UML 顺序图或交互图[23]或者构件交互图[24].Gallagher 等人[8]用带有类变量的交互状态机生成组合状态以及构件控制流图,并基于此生成集成测试用例.Ali 等人[6]将交互图和状态机相结合,提出状态交互测试模型(state collaboration test model)用于测试.这些方法扩展 UML 进行集成测试,但都假定构件本身是正确的.而本文则专注于测试单个构件的健壮性.

针对 API 的健壮性测试已有工具,如 Fuzz[25]和 Ballista[4]分别对 Unix 和 POSIX 的实现平台测试,Jcrasher[5]

为每个数据类型预设一些值来测试 Java 类.这些工具的优点在于只需提供被测软件的 API.但是它们不考虑状态信息,只能找到较浅层的错误,无法发现那些只在特定状态下才被激发的缺陷.

对健壮性和基于状态的测试都有过相关工作,但将二者有机结合、基于状态信息对构件的健壮性做深入的测试的工作并不多见.文献[14]对构件健壮性测试给出了一个初步的基于状态机的途径.本文在此基础上,基于构件演算理论对其健壮性重新进行定义,为构件健壮性测试和测试结果的自动分析提供了初步的理论基础和技术支持;实现了原型工具 RoTesCo 以及 Fuzz 和 SF 两种算法;选取 Kaffe 虚拟机、Apache 标准库等开源系统中的典型构件进行建模和实证分析,证实了测试框架的有效性.

5 总 结

本文提出一个基于状态机的构件健壮性测试框架,并实现了原型工具 RoTesCo,测试构件处理无效输入和不当调用的能力.我们在构件语义框架下定义了健壮性.RoTesCo 以构件的规约及其 Java 字节码为输入,生成 JUnit 测试用例.用例驱动构件沿着状态机的路径进行状态转换,在最后用无效输入或者不当调用试图引发构件失效.RoTesCo 被用于一组公开构件的测试.结果显示,它比传统算法更能有效检测健壮性错误.

关于后续工作,我们计划增强工具处理规约的能力,生成更有意义的用例;其次,有许多开源构件,可以将它们引入到本框架进行测试;最后,本文关注单个构件的健壮性测试.但构件只是系统的可复用单元,构件集成和构件系统涉及到更多的缺陷引入点.因此,针对构件集成方法和交互协议的测试以及整个构件系统健壮性的测试、评估方法都将是有意义的课题.

致谢 本文的很多工作是雷斌作为 Fellow 在 UNU-IIST 访问学习期间完成的.文中很多想法来源于与刘志明高级研究员和 Charles Morisset 博士后的讨论,在此对他们表示衷心的感谢.

References:

[1] Sun Microsystems, EJB 3.0 Expert Group. JSR-000220 enterprise JavaBeans 3.0 final release. 2006.

[2] Microsoft. COM: Component object model technologies. 2008. http://www.microsoft.com/com/default.mspx

[3] Chen ZB, Liu ZM, Ravn AP, Stolz V, Zhan NJ. Refinement and verification in component-based model driven design. Research Report, 388, Macao: Int'l Institute for Software Technology, United Nation University, 2007.

[4] Kropp NP, Koopman PJ, Siewiorek DP. Automated robustness testing of off-the-shelf software components. In: Arlat J, Chillarege R, eds. Proc. of the 28th Annual Int'l Symp. on Fault-Tolerant Computing. Washington: IEEE Computer Society Press, 1998. 230−239.

[5] Csallner C, Smaragdakis Y. Jcrasher: An automatic robustness tester for java. Software—Practice & Experience, 2004,34(11): 1025−1050. [doi: 10.1002/spe.602]

[6] Ali S, Briand LC, Rehman MJ, Asghar H, Iqbal MZZ, Nadeem A. A state-based approach to integration testing based on UML models. Information and Software Technology, 2007,49(11-12):1087−1106. [doi: 10.1016/j.infsof.2006.11.002]

[7] Drusinsky D. Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking. London: Newnes, 2006. 96−97.

[8] Gallagher L, Offutt J. Test sequence generation for integration testing of component software. The Computer Journal, 2009,52(5): 514−529. [doi: 10.1093/comjnl/bxm093]

[9] Tretmans T. Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools, 1996,17(3):103−120.

[10] IBM. Web services architecture overview. 2008. http://www.ibm.com/developerworks/webservices/library/w-ovr/

[11] Meyer B. The grand challenge of trusted components. In: Dillon J, Tichy W, eds. Proc. of the 25th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society Press, 2003. 660−667.

[12] He JF, Li XS, Liu ZM. A theory of reactive components. Electronic Notes in Theoretical Computer Science, 2006,160:173−195. [doi: 10.1016/j.entcs.2006.05.022]

[13] IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, 1990.

[14] Lei B, Liu ZM, Morisset C, Li XD. State based robustness testing for components. Electronic Notes in Theoretical Computer Science, 2010,260:173−188. [doi: 10.1016/j.entcs.2009.12.037]

[15] Binder RV. Testing Object-Oriented Systems: Models, Patterns, and Tools. Boston: Addison-Wesley Longman Publishing Co., Inc., 1999. 357−360.

[16] Chilenski JJ, Miller SP. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 1994,9(4):193−200.

[17] Warmer JB, Kleppe AG. Octopus open source project. 2006. http://octopus.sourceforge.net/

[18] The Eclipse Foundation. Eclipse Modeling Framework. 2010. http://www.eclipse.org/modeling/emf/

[19] Hussmann H, Demuth B, Finger F. Modular architecture for a toolset supporting OCL. Science of Computer Programming, 2002, 44(1):51−69. [doi: 10.1016/S0167-6423(02)00032-1]

[20] Sun Developer Network. The reflection API. 2008. http://java.sun.com/docs/books/tutorial/reflect/TOC.html

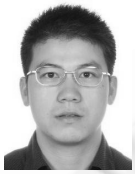
[21] Pick J. Kaffe Java virtual machine. 2008. http://www.kaffe.org/

[22] Chen MS, Qiu XK, Xu W, Wang LZ, Zhao JH, Li XD. UML activity diagram-based automatic test case generation for Java programs. The Computer Journal, 2009,52(5):545−556. [doi: 10.1093/comjnl/bxm057]

[23] Zheng WQ, Bundell G. Model-Based software component testing: A UML-based approach. In: Lee R, *et al*., eds. Proc. of the 6th IEEE/ACIS Int'l Conf. on Computer and Information Science. Washington: IEEE Computer Society Press, 2007. 891−899.

[24] Wu Y, Pan D, Chen MH. Techniques for testing component-based software. In: Andler SF, Offutt J, eds. Proc. of the 7th Engineering of Complex Computer Systems. Washington: IEEE Computer Society Press, 2001. 222−232.

[25] Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990,33(12): 32−44. [doi: 10.1145/96267.96279]



雷斌(1982－),男,江苏洪泽人,博士,主要研究领域为软件工程,软件测试.

卜磊(1983－),男,博士生,主要研究领域为软件工程,形式化方法,软件验证.

王林章(1973－),男,博士,副教授,CCF 高级会员,主要研究领域为软件工程,软件测试.

李宣东(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,软件建模与分析,软件测试与验证.