一种面向缺陷的软件测试方法:变异测试

单锦辉　李炳斌　孙　萍
(中国酒泉卫星发射中心)

摘要 软件测试是保证软件质量的重要手段。变异测试是一种行之有效的软件测试方法,系统地模拟软件中的各种缺陷,然后构造能够发现这些缺陷的测试数据集。通过简要介绍变异测试的基本原理,分析软件故障模型与变异算子的关系,探讨了变异测试中测试数据自动生成的方法,并指出变异测试进一步的研究方向。

关键词 变异测试　测试数据　充分性准则　软件缺陷　软件测试

1 引言

随着社会的不断进步和计算机科学技术的飞速发展，计算机及软件在国民经济和社会生活等方面的应用越来越广泛。作为计算机的灵魂,软件起着举足轻重的作用。软件的失效有可能造成巨大的经济损失,甚至危及人的生命安全。例如,1996 年 Ariane 5运载火箭发射失败是由软件故障引起的[1]。

软件失效机理可以描述为：软件错误→软件缺陷→软件故障→软件失效[2]。软件错误(software error)是指在软件生存期内的不希望或不可接受的人为错误，其结果是导致软件缺陷的产生。软件缺陷(software defect)是存在于软件(文档、数据、程序)之中的那些不希望或不可接受的偏差，其结果是软件运行于某一特定条件时出现软件故障。软件缺陷是存在于软件内部的、静态的一种形式。软件故障(software fault) 是指软件运行过程中出现的一种不希望或不可接受的内部状态。出现软件故障时若无适当措施(如容错)加以及时处理,便产生软件失效。软件故障是一种动态行为。软件失效(software failure)是指软件运行时产生的一种不希望或不可接受的外部行为结果。

软件开发的各个阶段都需要人的参与。因为人的思维和交流不可能完美无缺,出现错误是难免的。与此同时，随着计算机所控制的对象的复杂程度不断提高和软件功能的不断增强，软件的规模也在不断增大。例如,Windows NT 操作系统的代码大约有3200 万行。这使得错误更可能发生。人们在软件的设计阶段所犯的错误是导致软件失效的主要原因。并且软件复杂性是产生软件缺陷的极其重要的根源[2]。

软件测试是保证软件产品质量的重要手段。一方面它能够帮助我们揭示软件中的缺陷。另一方面，当软件测试达到一定的充分度时，它也能够帮助我们掌握软件的质量水平,建立对软件的信心。

变异测试是一种面向缺陷的软件测试方法[3,4],通过使用变异算子系统地模拟软件中的各种缺陷,然后构造能够发现这些缺陷的测试数据集，该测试数据集能够达到一定的测试充分度。变异测试具有排错能力强、方便灵活等优点,但是同时具有消耗资源较多,并且有些工作必须人工进行的缺点。

2 变异测试的基本原理

变异测试的基本原理是，使用变异算子每次对被测程序做一处微小的合乎语法的变动，例如将运算符“>”用“<”替换,产生大量的新程序,每个新程序称为一个变异体;然后根据已有的测试数据,运行变异体,比较变异体和原程序的运行结果:如果两者不同,就称该测试数据将该变异体杀死了。

例如,图 1 给出了一个求给定的两个整数 x 和 y 中较大者的 C 语言的源程序和它的一个变异体。

```
int max(int x, int y)
{
  int mx;
  if (x > y)
    mx = x;
  else
    mx = y;
  return mx;
}
```

(a)

```
int max(int x, int y)
{
  int mx;
  if (x < y)
    mx = x;
  else
    mx = y;
  return mx;
}
```

(b)

图 1　一个典型的 C 语言源程序和它的一个变异体

测试数据 x = 3, y = 4 能够将变异体杀死,而测试数据 x = 3, y = 3 不能将变异体杀死。

导致变异体不能被杀死的原因有两个:(1) 测试数据集还不够充分,通过扩充测试数据集便能将该变异体杀死;(2) 该变异体在功能上等价于原程序,称这类变异体为等价变异体。例如在对图 1(a)中的程序进行变异时将">"用">="替换,则该变异体就与原程序在功能上等价。

杀死变异体的过程一直执行到杀死所有变异体或变异充分度已经达到预期的要求。变异充分度是已杀死的变异体数目与所有已产生的非等价变异体数目的比值,即变异充分度 = D / (M - E),其中 D 为已经被杀死的变异体个数,M 为所有已产生的变异体总数,E 为所有已产生的变异体中与原来程序等价的变异体个数。

变异测试的结果(或目标)是得到一个有效的测试数据集,它能够杀死大多数变异体,使变异充分度达到预期的要求。

2.1 杀死变异体的条件

假设 M 是对程序 P 的语句 s 进行变异后的一个变异体,如果一个测试数据 t 满足以下三个条件,则其一定能杀死 M[5,6]:(1) 可达性条件 Cr:以 t 运行 M 时能够执行到变异语句 s。这是因为 M 和 P 只在 s 处不同,若执行不能到达 s,则 t 在 P 和 M 上的运行结果必然相同,因而 t 不能杀死 M。(2) 必要性条件 Cn:t 必须在到达 s 后使 M 产生一个不同于 P 的状态。若以 t 执行 P 和 M 到达 s 后的状态一致,则 P 与 M 的最终状态必然一致;这是因为 P 与 M 仅在 s 处不同,在 s 后的语句都是一致的。(3) 充分性条件 Cs:M 与 P 的最终状态不同。也就是说 P 与 M 在 s 处产生的不一致状态必需能够传递到程序的结束处。

变异测试是一种行之有效的测试方法,具有以下优点[4]:(1) 排错能力强。在变异测试的过程中,我们可能会发现被测程序在一个输入数据上的输出是错误的,而一个变异体却能够给出正确的结果。变异体与源程序的差别可以为我们排错提供有用的信息。(2) 灵活性好。在测试的过程中测试人员可以通过与测试工具的交互,有选择地使用变异算子的一个子集来完成不同层次的测试分析,还可以有选择地对程序中的一个片段作用变异体算子,对程序中的一个部分进行重点测试。(3) 自动化程度高。现有的变异测试工具可以自动地产生变异体并自动地运行变异体和源程序,自动地发现被杀死的变异体。

但变异测试也存在着一些缺点[7]:(1) 需要大量的计算机资源来产生、存储、运行变异体,比较运行结果,分析测试充分性。研究表明一个程序所产生的变异体数量与它含有的数据引用个数与数据对象个数的乘积成正比[8,9]。(2) 需要大量的人工操作。一方面判断一个活的变异体是否等价于原来的程序是一个不可判定的问题[5],必须人工进行。另一方面,为了杀死变异体,需要设计测试数据,这些工作目前大部分都是由人手工完成。

变异测试既可以用来揭示软件中的缺陷,又可以用来衡量测试数据集的揭错能力,评估测试的充分性。因此,变异测试受到人们的重视,并开发出 PIME, Mothra[6]等变异测试工具[3,7]。

3 软件故障模型与变异算子

软件人员在开发软件的过程中所犯的错误造成软件中存在缺陷,例如数组超界等。这种有缺陷的软件运行于某一特定条件时出现软件故障。人们对软件故障的各种现象进行研究,建立软件故障模型,以便查找和消除导致软件故障的软件缺陷,进而排除软件故障。变异测试是一种面向软件缺陷的测试方法,变异算子模拟了软件中的各种缺陷。人们针对软件单元、模块接口、面向对象软件和软件合约中可能出现的各种故障进行研究,提出相应的变异测试方法和变异算子。

3.1 软件故障模型

文献[10]针对科学计算程序的软件建立以下几种故障模型:计算型故障、分支型故障、循环型故障、功能型故障、死锁型故障、测试型故障。

(1) 计算型故障模型。包括变量的定义与引用方面的错误;数据的冗余;全局变量与局部变量、静态变量与动态变量的混淆;数组变量的越界错误;用错指针变量;数据类型不匹配的错误;数据操作方面错误,包括函数调用参数传递错误;赋值语句错误等。

(2) 分支型故障模型。包括谓词的错误;判定变量被赋了错误的值;谓词操作符不正确或少操作符;谓词中的变量不正确或少变量;谓词的结构不正确(if 与 else 不匹配,分号位置不对);少默认情况或默认情况不对;少动作或动作不对;出现额外动作等故障。

(3) 循环型故障模型。包括永不循环故障和死

循环故障等。这主要是由循环变量引起的,即循环谓词中循环变量被赋予了错误的值; 循环谓词中操作符不正确或缺少操作符; 谓词中的变量本身不正确或少变量;少动作或动作不对。

(4) 功能型故障模型。是指一些软件功能或性能不完善的故障,具体包括功能或者性能规定错误、遗漏了某些功能、规定了某些冗余的功能、为用户提供的信息有错误、对意外情况的处理有错误、设计界面用户不满意、功能的结果不完善、提供的用户接口不完善等。

(5) 死锁型故障模型。指各并发进程或线程彼此互相等待对方所拥有的资源, 且这些并发进程或线程在得到对方的资源之前不释放自己所拥有的资源。

(6) 测试型故障模型。包括:测试目的不明确;测试计划有误;测试步骤错误;测试实施有误;测试文档有误;测试用例不充分;测试环境有误等。

文献[11]针对 C 语言程序建立以下几种故障模型:坏的存储分配、内存泄漏故障、初始化变量错误、指针和越界指针的引用错误、数组越界错误、非法的算术运算错误(包括除数为 0、负数开方、对数幂变量为 0 或负数等)、整数或浮点数的上溢/下溢错误、非法类型转换错误、不可达代码错误等。

3.2 变异算子

在变异测试的过程中, 变异体是将变异算子作用到源程序上得到的。研究结果表明,变异体与真实的缺陷非常相似,并且与手工播种的缺陷有区别,手工播种的缺陷比真实的缺陷更加难于检测[12]。

变异算子模拟了软件中的各种缺陷。因此,变异算子的设计在变异测试中是非常重要的一步。变异算子的设计依赖于程序设计语言。经过多次改进和提高,变异测试工具集 Mothra 针对 Fortran 语言定义了以下 22 种变异算子[6]:(1) AAR,数组元素替换;(2) ABS,插入绝对值符号;(3) ACR,用数组元素替换常量;(4) AOR,算术运算符替换;(5) ASR,用数组元素替换变量;(6) CAR,用常量替换数组元素;(7) CNR,可比较的数组替换;(8) CSR,用常量替换变量;(9) DER, 替换 DO 语句的结束语句;(10) LCR,逻辑运算符替换;(11) ROR,关系运算符替换;(12) SAR,用变量替换数组元素;(13) SCR,用变量替换常量;(14) SVR,变量替换;(15) CRP,常量替换;(16) DSR,修改 DATA 语句;(17) GLR,替换 GOTO 语句的标号;(18) RSR,RETURN 语句替换;(19) SAN, 语句分析;(20) SDL, 删除语句;(21) SRC,源常量替换;(22) UOI,插入一元操作符。

例如,对程序语句"a[0]=c+1",作用变异算子"CSR",用常量 2 替换同类型的变量 c,得到"a[0]=2+1"。又如,对程序语句"if((d>a[0])&&(c>1))",作用变异算子"LCR",用逻辑操作符||替换 &&,得到"if((d>a[0])||(c>1))"。

变异算子可以模拟软件中的各种缺陷。这种有缺陷的软件运行将有可能导致产生某些软件故障。例如, 利用 CSR、SCR、CRP 等变异算子可以模拟变量的定义与引用、数组变量的越界、非法的算术运算等方面的计算型故障。利用 AOR、LCR、ROR 等变异算子可以模拟分支型故障和循环型故障。利用变异算子 SDL 删除动态内存释放语句可以模拟内存泄漏故障等。

3.3 接口变异测试

接口变异是对传统变异测试的扩展, 在模块的接口处作用变异算子产生变异体。研究集成错误是为了设计接口变异算子, 接口变异算子作用后生成的变异体应该尽量包括所有的集成错误。可以针对 C 语言设计两组接口变异算子[13]:

(1) 第一组变异算子作用于被调用函数内部。该组变异算子包括:(a) 将接口变量替换为 其他 的变量。(b)通过改变 其他 的变量或常量,从而影响接口变量的值。在下列两种情形下使用该组变异算子:要改变的变量或常量位于 Return 语句中;要改变的变量或常量所在的语句中包含接口变量。(c)对接口变量或类似于(b)中的非接口变量进行++或—操作。(d)对接口变量或类似于(b)中的非接口变量进行-,!,~操作。(e)改变通过 return 语句返回的值等。

(2) 第二组变异算子作用于调用函数内部。该组变异算子包括:(a)参数替换;(b)交换参数,即改变调用的参数顺序;(c)删除参数(该类错误能被编译器检测出来);(d)插入一元操作符,即-,!,~等;(e)删除调用语句。

对上述接口变异算子进行实验研究,结果表明,接口变异产生的变异体数目比传统的变异产生的少得多。接口变异是一种对接口(包括函数调用、参数传递和全局共享变量等)进行测试的有效手段。接口变异充分性准则是集成测试中一个非常有效的准则。

3.4 面向对象软件的变异测试

面向对象软件引入了类、对象、继承、封装、信息

隐蔽、多态、动态绑定等新特征。类变异是传统变异测试的一个延伸，关心那些传统的变异测试系统无法检测的面向对象软件特征(如继承、多态等)相关的软件缺陷。类变异使用特定的变异算子将带有面向对象软件特征的错误植入到原程序中以产生变异体。

文献[14]和[15]针对 Java 语言定义类变异算子，通过类变异算子植入与 Java 面向对象特征(类定义和引用、单继承、信息隐藏、多态等)相关的错误。所定义的类变异算子包括：将一个对象的类型改变成兼容的类型；改变运行时刻对象类型，在实例创建语句中以兼容类修改原来的类；修改方法定义中的参数的顺序；删除重载(overloading)方法的定义；改变方法调用表达式中的参数顺序；减少调用表达式中参数个数；在子类中删除复写(overriding)的方法；在子类中删除与父类(或父接口)中同名的成员变量；在子类中增加一个出现在父类或接口中的变量；将访问控制符(public, private, protected 和 default)修改成 其他 类型；将 static 变量修改成非 static 变量，或将非 static 变量修改成 static 变量；当有多个 catch 语句的时候，删除一个 catch 子句；当有一个 exception 子句和一个 finally 子句时，删除一个 catch 子句；交换操作异常的方法，即原来是用 try{}catch{} 改成 throws，或进行相反的操作。

文献[16]针对上述类变异算子进行实验研究。结果表明，类变异测试是一种对面向对象软件进行测试的有效手段。

3.5 合约变异测试

传统的变异测试针对程序代码进行变异，可以对功能规约进行变异。规约变异的目的不是为了发现规约中的错误，而是为了发现由于规约被错误理解或实现所导致的软件缺陷[17]。由于合约是功能规约的部分体现，因此，合约变异也可被视为规约变异的一种特例。文献[17]中所指的合约包括可执行的代码以及抽象的软件规约。与传统的变异测试不同，合约变异所针对的不是程序的代码，而是软件的合约。实际上，变异后的合约代表了两类错误。对抽象的规约来说，它是软件开发或设计人员对需求可能产生的误解；对可执行的合约来说，它是由合约错误导致的实现错误。

文献[18]和[19]定义了五类合约变异算子：合约取反；条件交换，即交换前置条件和后置条件；前置条件弱化；后置条件强化；固定合约值等。变异预言(Mutation Oracle) 是区分原程序与变异体的程序或条件。在传统的变异测试中，一般采用程序的运行结果作为区分变异体与原程序的条件。文献[17]虽然对合约进行变异，但它没有明确给出变异预言。文献[19] 将合约检查结果的一致与否作为判断变异体是否被杀死的条件。实验结果表明，合约变异算子产生的变异体数目要远远少于传统的变异算子，而这些变异算子能达到与传统变异算子几乎完全相同的效果[19]。

4 变异测试中测试数据自动生成

变异测试的结果(或目标)是得到一个有效的测试数据集，它能够杀死大多数变异体，使变异测试充分度达到预期的要求。因此，自动生成能够杀死变异体的测试数据对于变异测试是至关重要的。2.1 节指出了能够杀死变异体的测试数据应该满足可达性、必要性和充分性条件。

寻找满足充分性条件的测试数据在实践中是非常困难的[6]，现有的变异测试中测试数据生成方法都是寻找那些满足必要性条件和可达性条件的测试数据，这样得到的测试数据不保证一定能杀死变异体，但有实验表明这样得到的测试数据集具有较高的变异测试充分度[20]。

目前主要有两种自动生成能够杀死变异体的测试数据的方法：基于约束的生成方法 (Constraint-Based Test data generation，以下简称 CBT 方法)[6]和动态域削减方法 (Dynamic Domain Reduction test data generation，以下简称 DDR 方法)[21]。

CBT 方法采用符号执行方法构造由变异体的可达性条件和必要性条件组成的约束系统，然后采用域削减、选择域最小的变量随机赋值和回代相结合的方法求解约束系统。实验结果表明 CBT 方法比较有效[20]，但是 CBT 方法存在以下几个问题：(1) 符号执行方法通常要进行复杂的代数运算，并且难于处理依赖于输入变量的循环条件、数组元素下标和模块调用；(2) 该方法需要在约束满足之前将整个约束系统都表示出来，而约束系统通常是非常大的，需要很大的存储空间；(3) 约束求解技术有可能导致有解的约束系统无解。

针对 CBT 方法存在的不足，文献 [21] 提出了 DDR 方法，其动态特性有助于改进对数组、循环、表达式等的处理，并且动态地沿着控制流图移动使得路径约束可以立即被求解，这在时间和空间方面都

更加有效。但是动态域缩减方法在求解约束系统的过程中需要进行回溯,所以即使程序路径 W 上所有谓词函数都是输入变量的线性函数，它也要进行大量的迭代。如果 W 上几个分支谓词依赖于公共的输入变量,那么在回溯时会造成大量的浪费。并且由于动态域缩减方法需要选择分裂点对变量的域进行分裂,故当变量的域为无穷区间时,即使 W 上所有谓词函数都是输入变量的线性函数，并且各输入变量均无整数限制,并且 W 是可行的,该方法仍然有可能找不到使 W 被经过的测试数据。

5 结论

软件测试是保证软件质量的重要手段之一。变异测试是一种面向缺陷的软件测试方法，具有排错能力强、方便灵活等优点，既可以用来揭示软件缺陷,又可以用来衡量测试数据集的揭错能力,评估测试的充分性。当前在单元测试、接口测试、面向对象软件的测试和合约测试等方面都得到了应用。但是变异测试需要消耗大量的计算机资源，并且有些工作必须人工进行。

变异测试通过变异算子系统地模拟软件中的各种缺陷,然后构造能够发现这些缺陷的测试数据集。变异测试技术具有广阔的应用前景。为提高变异测试方法的实用性,需要进一步减少变异测试的开销,提高变异测试的自动化程度，包括用计算机辅助检测等价变异体和变异测试中测试数据的自动生成,即自动生成能够杀死变异体的测试数据。这些都是有意义的研究方向。 ◇

参考文献

[1]Weyuker E J. Testing component-based software: A cautionary tale. IEEE Software, 1998, 15(5): 54-59.

[2]Cai Kai-Yuan. The Foundations of Software Reliability Engineering. Tsinghua University Press, Beijing, 1995, in Chinese.

[3]Zheng Ren-Jie. Computer Software Testing Technologies. Tsinghua University Press, Beijing, 1992, in Chinese.

[4]Zhu Hong, Jin Ling-Zi. Quality Assurance and Testing of Software. Science Press, Beijing, 1997, in Chinese.

[5]Offutt A J, Pan J. Detecting equivalent mutants and the feasible path problem. In: Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96), Gaithersburg MD, June 1996. 224-236.

[6]DeMillo R A, Offutt A J. Constraint-based automatic test data generation. IEEE Trans on Software Engineering, 1991, 17(9): 900-910.

[7]Offutt A J, Untch R. Mutation 2000: Uniting the orthogonal. Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, October 2000, 45-55.

[8]Wong W E, Mathur A P. Reducing the cost of mutation testing: An empirical study. Journal of Systems and Software, 1995, 31(3):185-196.

[9]Offutt A J, Rothermel G, Untch R H, Zapf C. An experimental determination of sufficient mutant operator. ACM Transactions on Software Engineering and Methodology, 1996, 5(2):99-118.

[10]朱 荣, 徐拾义. 软件测试中故障模型的建立. 计算机工程与应用, 2003, 17: 69-71,91.

[11]宫云战. 一种面向故障的软件测试新方法. 装甲兵工程学院学报, 2004, 18(1): 21-25.

[12]Andrews J H, Brand L C, Labiche Y. Is Mutation an Appropriate Tool for Testing Experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). St. Louis, MO, USA, May 15-21, 2005. 402-411.

[13]Delamaro M E, Maidonado J C, Mathur A P. Interface mutation: an approach for integration testing. IEEE Trans on Software Engineering, 2001, 27(3): 228-247.

[14]Ma Y-S, Kwon Y-R, Offutt, J. Inter-class mutation operators for Java. In: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002). Annapolis, MA, USA, November 12-15, 2002. 352-363.

[15]Kim S, Clark J A, McDermid J A. Class mutation: Mutation testing for Object-Oriented programs. In: Proceedings of OOSS: Object-Oriented Software Systems, October 2000.

[16]Lee H-J, Ma Y-S, Kwon Y-R. Empirical evaluation of orthogonality of class mutation operators. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference. Busan, Korea, November 30 - December 3, 2004. 512-518.

[17]Aichernig B K. Contract-based mutation testing in the refinement calculus. Electronic Notes in Theoretical Computer Science, 2002, 70(3).

[18]姜 瑛. 基于接口合约的软件构件易测试性研究. [博士学位论文], 北京大学研究生院, 2005.

[19]Jiang Y, Hou S-S, Shan J-H, Zhang L, Xie B. Contract-based mutation for testing components. In: Proceedings of the 21st International Conference on Software Maintenance (ICSM' 2005). Budapest, Hungary, September 26 - 29, 2005. 483-492.

[20]DeMillo R A, Offutt A J. Experimental results from an automatic test case generator. ACM Transactions on Software Engineering Methodology, 1993, 2(2): 109-127.

[21]Offutt A J, Jin Z, Pan J. The dynamic domain reduction procedure for test data generation. Software: Practice and Experience, 1999, 29 (2): 167-193.