基于变量影响分析与数据变异的回归测试用例生成

杨 波[1),2)] 吴 际[2)] 刘 超[2)]

[1)](北方工业大学计算机学院 北京 100144)

[2)](北京航空航天大学计算机学院 北京 100191)

摘 要 故障修复后,程序员还需验证那些与修复故障相关的区域是否还存在故障或者引入了新的故障,这时有可能需要补充新的测试用例.现有研究大多依赖符号执行等技术,这样可能导致状态空间过大.且现有的研究统一地来考虑程序中控制流相关的故障和数据流相关的故障,但程序中控制流相关的故障与数据流相关的故障存在区别.由于程序运行时的行为与变量密切相关,并且在人工程序调试时程序员经常会关注变量的状态变化.因此该文提出了一种基于变量影响分析和数据变异的回归测试用例方法,通过分析程序中动态执行的变量状态的变化,提炼出了一种变量行为模型,该模型描述了变量自身的状态变化和变量间的依赖关系.利用变量行为模型来找到影响故障的语句集合,基于该语句集合并利用数据变异的方法可以有针对性地补充测试用例.经过实验验证,基于变量影响分析和数据变异的回归测试用例方法,在针对数据流相关的故障修复情况进行验证时效果明显,且要优于随机测试用例生成方法和基于路径覆盖的测试用例生成方法.

关键词 故障修复;影响分析;变量;数据变异;回归测试

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02372

An Approach of Regression Test Case Generation Based on Variable Impact Analysis and Data Mutation

YANG Bo[1),2)] WU Ji[2)] LIU Chao[2)]

[1)](*College of Computer Science, North China University of Technology, Beijing* 100144)

[2)](*School of Computer Science and Technology, Beihang University, Beijing* 100191)

Abstract Programmers need to verify program correctness after fault repairing. They often need to add new test cases to test programs for fault repairing, which may bring out new faults. There are many technologies depended on symbol execution which will cause state space explosion, and furthermore these technologies are unified to consider the control flow related faults and data flow related faults in program. However, the control flow related faults and data flow related faults are different. The behaviors of the program and the variables are closely related. Programmers often pay attention to variables state changes in the process of manual debugging. This paper proposes an approach of regression test case generation based on variable impact analysis and data mutation because of these factors. First, it analyses dynamic variable state change and gets a variable behavior model, which describes state change of the variables and the dependence relationship of variables. Then, statements related to fault can be obtained by analyzing the variable behavior model. Last, test cases can be getting by using these statements and data mutation technology. The main research of this paper is verified by experiments. The experimental

收稿日期:2015-07-31;在线出版日期:2016-04-10.本课题得到国家自然科学基金(61502011,61370051,61402016)、北京市教委科技计划(KM201610009007)、北京市大学生科研训练计划深化项目(XN003-16)、北京工业大学科研启动基金项目、北方工业大学开放实验项目资助.**杨 波**,男,1981 年生,博士,讲师,中国计算机学会(CCF)会员,主要研究方向为软件测试、数据挖掘.E-mail:yangbo090313@163.com.**吴 际**,男,1974 年生,博士,副教授,主要研究方向为软件测试、软件可靠性分析.**刘 超**,男,1958 年生,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为软件工程、软件测试.

results show that the proposed approach is good for finding data related fault and achieving better results comparing with random approach and paths coverage approach.

Keywords　fault repair; impact analysis; variable; data mutation; regression testing

1　引　言

有统计数据表明，回归测试占软件维护费用的50%以上[1]，因此提高回归测试的效率非常必要. 在需要进行回归测试的场景中，有一类是发现故障且修复了故障之后的回归测试，在这类情况下可能需要补充新的测试用例. 这其中包含两个方面的考虑，一方面是原有的故障是否已经修复正确，另一方面是原有的故障修复之后，是否会带来新的故障. 程序员验证故障修复时，会对验证修复的目标进行设定，比如设定需要覆盖的程序单元的集合. 然后再用已有的测试用例做回归测试，如果回归测试的结果反映出没有达到验证修复的目标，例如没有完全覆盖到设定的语句集合，此时补充新的测试用例非常有必要.

由于程序中可能存在复杂的函数调用关系，以及逻辑分支、循环、嵌套等复杂的逻辑结构，并且包含相当多的变量，变量之间还存在相当复杂的依赖关系，因此想要在修复故障之后进行全面的回归测试显然花费太大. 并且代码的规模越大，进行全面回归测试的复杂度就越高. 因此如何提高故障修复之后回归测试的效率，是非常有价值的研究问题.

在进行回归测试之前，程序或软件可能会产生新的测试需求，对于这类需求，研究者提出了回归测试中的测试用例方法[2-3]，希望利用这些方法生成新的测试用例来满足新的测试需求. Santelices 等人[2]提出了一种结合控制流图、数据流图和符号执行技术的测试用例生成方法，该方法认为新生成的测试用例集应该能够覆盖所有以修改点为起始点的依赖关系. 除了提出该方法之外，Santelices 等人[3]还将测试用例增加技术与 PIE 模型相结合，提出了传播分析和谓词约束求解的测试用例生成方法. 利用上述方法可以获得测试输入，且能同时记录所满足的测试需求. 当新的测试需求全部被满足时，测试用例生成就宣告完成.

Jiang 等人[4]提出一种回归测试中的测试用例增加方法，该方法首先对程序建立图模型，并且考察图模型中的边是否被测试用例覆盖，如果没有被覆盖，则搜索能连接初始节点与未覆盖边源节点的一条最短路径，并将搜索到的最短路径进行组合，最终形成子图. 根据这些子图来产生测试用例. Xu 等人[5]通过对比测试用例在新旧两个程序版本的执行，得到修改版本中未被测试用例覆盖的分支集合，然后利用符号测试等技术产生新的测试用例. Maragathavalli等人[6]利用覆盖的路径相似为准则，利用已有的测试用例来选择合适的测试用例. Nachiyappan 等人[7]利用进化算法来缩减回归测试用例，Zhang 等人[8]分析程序对应的树形结构找到发生改变的路径，并确定影响路径改变的分支，依据此来生成测试用例. Ye 等人[9]分析程序修改前后对应的活动图，找出其他受到影响、没有受到影响的路径，并依据此选择测试用例. Kumar 等人[10]分析修改前后程序的路径，选择新增加的路径及发生改变的路径，作为需要覆盖的路径. Yoo 等人[11]利用已有的测试用例，通过分析不同的测试变量搜索域和交互水平，采用基于搜索的方法来生成新的测试用例. 吴川等人[12]根据测试目标从已有的测试用例中进行选择，提出了一种基于路径相关的测试数据进化生成方法. 谢晓园等人[13]提出了基于相似度度量的适应值函数，来生成覆盖指定路径的测试用例.

但目前基于影响分析的回归测试用例生成方法还存在以下几点不足：

(1) 现有的基于影响分析的回归测试用例增加方法主要依赖符号执行等传统生成技术，这样使得这些技术会面对状态空间爆炸的问题[14].

(2) 现有的研究统一地来考虑程序中控制流相关的故障和数据流相关的故障，但其实这两种故障存在自身的特点，Santelices 等人[15]基于对程序频谱的分析，指出程序频谱分析应分别针对不同类型的故障来进行，因此对于不同类型的故障，需要有专门验证其修复的方法；

针对上述研究问题，本文专门总结了数据流相关故障的特点，并细致地分析了程序运行过程中变量的动态行为，在其中提炼出能够反映变量行为的模型. 其次，利用变量行为模型，找出与故障相关的语句集合，接下来借助数据变异技术，自动生成与故障相关的测试用例. 该方法还涉及到测试用例的判

定问题,在文中都进行了详细的介绍.

本文第 2 节对数据流相关的故障给出简短的描述,并基于对变量行为的分析提出变量行为模型,其中包括变量自身的变化和变量间的依赖关系,并且给予详细描述,另外介绍将变量自身变化与变量间依赖关系统一在一起的变量行为模型;第 3 节主要讲述如何利用变量行为模型和数据变异,来有针对性地对修复故障之后进行回归测试用例的生成;第 4 节详细地介绍用变量影响分析及数据变异方法进行的实验,并分析实验结果;第 5 节介绍测试用例生成方法、数据流分析和切片分析的相关工作;最后在第 6 节对研究工作进行总结与展望.

2 变量行为分析及建模

2.1 数据流相关的故障

程序中的故障可以分为两大类:分别为与数据流相关的故障和与控制流相关的故障[16],这两类故障存在明显的区别,与数据流相关的故障和程序中的变量联系紧密,变量使用错误与变量定义错误都能导致程序触发此类故障.与控制流相关的故障和程序的结构相关,语句执行顺序的改变、谓词条件的错误定义等都可导致程序触发此类故障.

值得注意的是,现有的基于影响分析的回归测试用例生成方法都没有能够充分地利用程序中变量的取值变化信息、变量间的依赖关系和程序的执行逻辑信息.程序中与数据流相关的故障和这些信息密切相关,且在人工程序调试中,这些信息也是需要重点关注的.

与数据流相关的故障有其自身独特的特点,表现在以下几个方面[17]:其一,与数据流相关的故障未必会导致程序控制流错误,即程序执行错误的路径.这样的故障往往使得程序在执行过程中仅出现一个或多个变量的值表现出异常,更特殊的情况是:程序可能执行到最终结果输出时,才出现输出结果与预期值有差异.其二,与数据流相关的故障有时需要对某个或多个变量的值多次进行计算后才会被触发,例如在某循环内部,某变量值出现越界、溢出、误差过大等情况的前提条件是对该变量的值进行了多次迭代计算.对于此类故障,若能保存变量的执行轨迹,将有助于对此类故障的理解和修复.其三,与数据流相关的故障发生时往往是因为某个变量的取值错误,使得其他变量(某个变量或多个变量)也出现错误,这是变量间的依赖关系导致的,因此在对这类故障进行理解和修复时,需要考虑这种依赖关系.

从上面的讨论可知,与数据流相关的故障受两大方面的影响:变量的取值变化与变量间的依赖关系.这两个方面的影响混杂在一起,使得对这类故障的修复验证相对比较复杂.

本文专注于研究由数据流引起的故障的修复验证,所提出的基于变量影响分析和数据变异的回归测试用例生成方法也主要是针对数据流相关的故障.

2.2 变量行为分析

变量行为分析与程序中的数据流密切相关,程序中的数据流指的是:设 x 是程序 P 中的变量,P 沿所有可能路径所构成的对 x 的所有操作序列都叫作变量 x 的数据流.P 中所有变量的数据流集合叫作程序 P 的数据流[18].

目前已经存在多种针对程序中变量的行为分析的方法,其中典型的包括数据流分析方法、数据流测试技术和切片技术.数据流分析是一种典型的变量行为分析方法,其中包括静态数据流分析与动态的数据分析方法.切片也是一种典型的变量行为分析技术,Weiser 博士[19]定义程序切片为程序中所有能够影响某个输出的语句集合,切片包括动态切片与静态切片,动态切片与静态切片相比有很多优点,动态切片仅考虑在某个特定输入条件下,程序的运行情况,因此有很多情况,例如数组下标、循环依赖关系、指针等信息能在程序运行时动态确定,这样得出来的结果会更具体和精确.而在程序调试过程中,通常只关注程序在某个输入下的执行行为,此时动态切片比静态切片有更高的准确度,更能够减少切片集合规模,缩小调试分析范围.

另外还有一些基于程序图谱的行为分析方法[20],程序图谱的概念是由 Reps 等人[20]在调试解决千年虫的问题时首先提出的.程序图谱包含了程序实体在测试用例运行过程中的覆盖信息,其中程序的元素可以是语句、语句块、谓词、方法、变量等.程序图谱用来描述或提供程序行为的表示方式.其在程序理解、回归测试等领域得到了非常广泛的应用.

上述的程序行为分析方法是进行变量影响分析的基础,这些方法侧重点不一样,解决问题的特点也不相同.因此在进行影响分析时,需要分析待解决的问题所具有的特点,从而提出合适的程序行为表示方法.动态切片技术由于关注程序在给定输入下的行为,因而在调试方面获得了更好的分析准确度,这

也是本文利用程序运行时行为开展影响分析研究的重要原因.然而,动态切片技术没有跟踪相关变量在程序运行时的取值变化等信息,因此无法有效发现程序运行过程中的错误状态.分析程序运行时变量的行为,将有助于对那些与数据相关的故障进行定位.

程序的行为与程序中的变量有着密切的关系,程序中变量行为的变化在一定程度上能够反映出程序的行为变化.因此对程序运行过程中变量的行为进行记录、建模、分析,能够从程序中数据流的角度来反映出程序的行为.并且对于那些由变量引起的程序异常进行检测,基于程序变量的行为分析方法的优势就能够更好地体现出来.

为了对数据流相关的故障进行分析并补充测试用例,需要重点关注程序运行过程中变量的变化信息,与此同时还需关注变量间的依赖关系.而已有的数据流分析、数据流测试的方法和技术,虽然反映了变量的定义、使用的特征,不过却难以形成程序执行过程中变量的动态行为的完整"轨迹",这给针对影响分析的回归测试用例生成带来了一定的难度.本文将变量在程序运行时的行为特征进行了抽象,重点关注变量的操作状态和变量之间的依赖关系,这样抽象出来的结构就是本文所提出来的变量行为模型.

2.3　变量行为模型

(1)变量

对于程序 P,其变量集合可以表示如下:

$$Var(P)=\{var_x,var_y,\cdots,var_z\}.$$

每一个变量都有相应的类型.这些类型可以是基本类型或复杂类型.

(2)变量的赋值与引用

赋值(Define):指变量的值或状态发生改变.

引用(Use):指变量被使用,其值或对象的内部状态没有任何变化.

(3)变量状态

为了能够反应程序的行为特点,我们将变量(包括其所指对象的某个属性)在程序中的每一次被赋值看作是对其值(即状态)的一次改变,故将其称为该变量的一个状态.变量状态包括以下几部分内容:变量标识、该状态的标识(状态的标识直接对应于变量被赋值的语句所在的程序位置,即行号)、在该状态下变量的值、该状态(即对应的赋值语句)的编号,变量在该状态位置所依赖的变量及其状态集合等信息.从另一个角度看,一个变量的状态变化,等效于产生或创建了一个新的变量的状态,并在程序的后续执行中完全取代了该变量的前身(即前一个状态).从这个意义上讲,变量状态的变化过程可以称作变量状态的演化序列,因此变量在程序中的状态演化序列可以看作是一组新变量的产生序列.

对于给定的程序 P,在一组输入 $I=\{in_1,in_2,\cdots,in_n\}$ 的条件下,第 x 个变量在第 i 次被重新赋值所产生的变量 x 的第 i 个状态,记作 var_x^i,定义如下:

$$var_x^i=(v,value,i,s,refs),$$

其中:v 为变量 x 的唯一标识;$value$ 为变量 var_x 的当前值;i 为变量 var_x 被赋值的次数;s 为变量 x 在第 i 次赋值时所在行号;$refs$ 指的是 x 在状态 i 下(即 var_x^i)所依赖的变量的集合.

对于每一个变量 x,其所有的状态的集合可以表示如下:

$$Var(var_x)=\{var_x^0,var_x^1,\cdots,var_x^m\}.$$

(4)变量的状态轨迹

变量的状态轨迹,指的是在程序的运行过程中,变量状态的变化序列:

$$Trace(var_x)=\{(var_x^i,var_x^{i+1})\mid var_x^i\in var_x,\ i,j\in N\}(N\text{ 为非负整数}).$$

(5)变量之间的依赖关系

变量之间的依赖关系:存在两个变量 x 与 y,若变量 y 的变化会对变量 x 产生影响,则称变量 x 依赖 y.依赖关系主要是指针对变量 x 赋值语句,x 出现在赋值语句的左侧,而 y 出现在赋值语句的右侧[14].这里,赋值语句也包括以其他方式影响 x 值的语句,比如对变量 x 的值会产生影响的函数调用(如 x 为其"输出型"参数,或 x 是会受到函数影响的全局变量,即其值可能被所调用的函数修改)等.变量的依赖关系将若干变量在值的层面上关联起来.

共有两种变量依赖关系:第 1 种是两个不同变量之间的依赖关系,如 $c=a+b$;第 2 种是变量自身的依赖关系,如 $a=a+1$.

变量状态之间的依赖关系:两个不同变量状态(以 var_x 和 var_y 为例)之间的依赖关系可以表示如下:

$$Vdr(var_x,var_y)=\{(var_x^i,var_y^j)\mid var_y^j\in Var(var_x^i).refs\},$$

其中:i 和 j 分别是变量 var_x 和 var_y 的变化次数,其中 var_x^i 为依赖关系的前驱节点,var_y^j 为依赖关系的后继节点.$Var(var_x^i).refs$ 中表示 $refs$ 是 $Var(var_x^i)$ 的子域.

在上式中,当 y 即 x 本身时,表示变量 x 自身的值之间的依赖关系,表示如下:

$$Vdr(var_x)=\{var_x^i,var_x^j \mid var_x^j \in Var(var_x^i).refs\}.$$

(6)变量行为模型

对于给定的程序 P,对于输入空间 I,其执行中涉及到的变量集合为 $Var(P_I)$,其变量的数据流轨迹可以表示为 $Trace(Var(P_I))$,各变量状态之间的依赖关系可以表示为 $Vdr(Var(P_I))$,其变量行为模型 $CM(P_I)$可以表示如下:

$$CM(P_I)=(Trace(Var(P_I)),Vdr(Var(P_I)))=\{(a,b) \mid (a,b)\in Trace(Var(P_I)) \| (a,b)\in Vdr(Var(P_I))\}.$$

2.4 变量行为模型与定义-使用对的区别

变量行为模型的一些相关概念来自于数据流测试,特别是利用了定义-使用对中的变量定义和变量使用的概念,但是两者之间也存在明显区别.

变量行为模型针对特定的输入,来找到运行中的变量状态的变化轨迹.静态数据流测试考虑到的是所有的情况,而且有时候,有些根据静态数据流测试找到的潜在的故障可能在实际的运行中根本就不会触发.虽然动态数据流测试更多地考虑变量在执行时所经过的程序执行路径,相比静态数据流测试更有优势,但变量行为模型则不仅会考虑变量自身在程序执行过程中的定义状态变化情况,还会考虑变量状态之间的依赖关系.此外,数据流测试中将变量单独来看待,没有关注变量间的依赖关系.

需要注意的是,变量行为模型与本文作者提出的数据链模型大致相似[17],两者都考虑变量的变化与依赖关系.主要区别在于数据链模型中有许多专门针对故障定位的方法,而变量行为模型中只考虑变量因为变化和依赖关系对故障修复时产生的影响.

3 基于变量影响分析和数据变异的回归测试用例生成

3.1 方法概述

此前分析了程序执行过程中变量的行为,本章以这些分析为基础,同时利用变量影响分析信息,提出了一种基于变量影响分析与数据变异的回归测试用例生成方法.基于变量影响分析和数据变异的测试用例生成方法的总体框架如图 1 所示.

从图 1 可以看出,首先需要收集故障修复前后测试用例的运行情况,根据测试用例的运行情况再

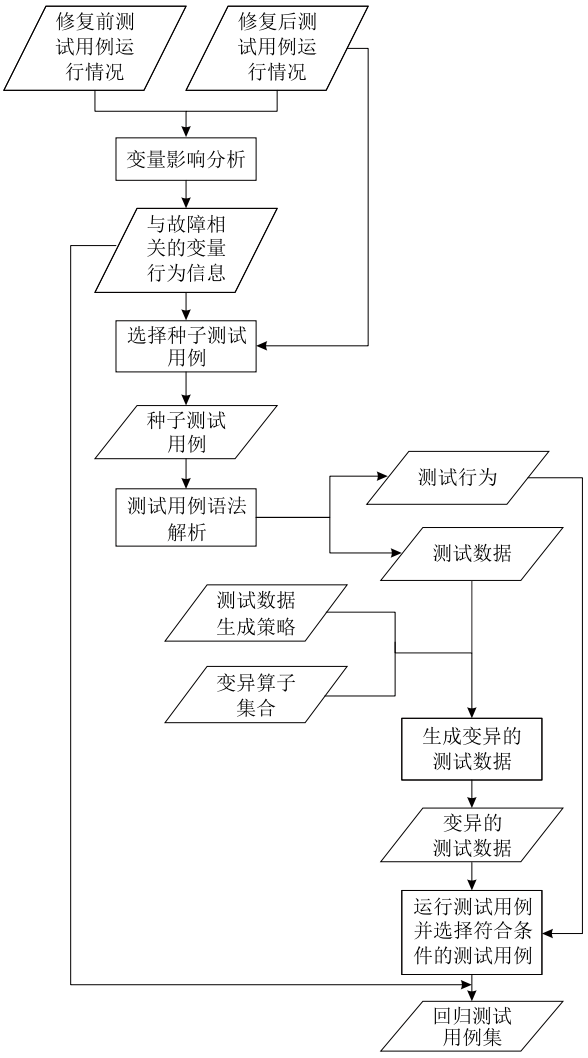

图 1 基于变量影响分析和数据变异的回归测试用例方法框架

结合变量行为模型,对程序修复故障前后的变量进行影响分析.影响分析的结果可以从初始的测试用例集合中选择一个或者多个测试用例作为种子测试用例,也可以在数据变异时选择合适的测试用例.

选择出来的种子测试用例将作为数据变异的基础.利用测试用例语法解析器来解析这些测试用例,从而得到种子测试用例中的测试行为、测试数据和测试配置等信息.

测试行为用于描述在测试用例中的各种行为.测试行为包括消息激励、消息响应及测试结果类别等,其中消息激励和消息响应中定义了需要的输入数据和预期响应数据类型和名称信息.测试数据是在测试过程中需要的相关数据,这些数据可作为输入施加给被测系统,并可作为被测系统输出进行判定的依据.测试配置是对测试系统静态组成结构的描述.

利用种子测试用例的测试数据，可以获得这些测试数据的类型信息，再结合测试数据生成策略和变异算子集合，生成新的测试数据，然后再结合之前得到的测试行为，来组合成新的测试用例. 这样生成的测试用例集有可能存在冗余，因此需要进行缩减，缩减的依据来自于与修复的故障相关的变量影响分析信息. 利用这些变量影响分析信息对产生的测试用例进行选择，可以得到最终补充的测试用例集合.

这种方法的主要技术特点有以下几点：首先，这种方法生成的测试用例主要用在故障修复之后的回归测试中，用于对修复后的程序进行检验. 其次，这种方法利用了此前修复的故障相关的变量影响分析信息，这使得测试用例生成更具有针对性，同时也能够将测试集中在那些可能会产生故障的地方，这样能够更有效并且更切合实际地发现更多的故障. 第三，这种方法与此前的方法不一样的是，该方法采用了数据变异的技术，通过数据变异，借助原有的测试用例来生成新的测试用例.

3.2　变量影响分析

程序中存在的故障被修复后，进行回归测试时需要重点测试该故障修复之后是否引入了新的故障. 而这时候与该故障相关的信息相当关键，这些信息能够帮助测试人员有目的地开展回归测试.

在利用变量影响分析进行测试用例生成的过程中，可以收集故障相关的变量影响分析信息，来作为选择种子测试用例的依据. 本文提出了与故障相关的变量影响分析信息收集算法，该算法的输入与输出如表 1 所示.

表 1　与故障相关的变量影响分析信息收集算法的输入与输出

输入参数序号		输入参数	描述
1	$CM(V)$	$CM_o=\{cm_1,\cdots,cm_n\}$, $n\in N$ $CM_m=\{cm'_1,\cdots,cm'_n\}$, $n\in N$	$CM(V)$指的是修复故障前后程序对应的变量影响分析，$CM(V)$包括 CM_o、CM_m、var_x^i 和 $var_x^{i'}$ 共 4 部分内容，CM_o、CM_m 分别指修复故障前的变量影响分析和修复故障后的变量影响分析.
2	$var_x^i=(v,value,i,s,refs)$		var_x^i 指修复故障前变量状态
3	$var_x^{i'}=(v',value',i',s',refs')$		$var_x^{i'}$ 指修复故障后变量定义状态
输出参数序号		**输出参数**	**描述**
1	S	$S=\{s_1,\cdots,s_l\}$, $l\in N$	S 指的是需要覆盖到的语句集合，$S=\{s_1,\cdots,s_l\}$，其中 s_i 指程序中需要覆盖的语句.

从表 1 可以看出，该算法的输入为 $CM(V)$、var_x^i 和 $var_x^{i'}$. $CM(V)$包括修复故障前后程序所对应的变量影响分析，输出则需要覆盖到语句集合 S. 整个与故障相关的变量影响分析信息收集算法如算法 1 所示. 该算法是以变量影响分析定义为基础实现的，其中还用到了变量状态影响链和变量状态转移轨迹. 变量状态影响链指的是受到变量状态影响的语句. 该算法主要由两个主要的步骤组成，分别是获取故障修复前受影响的语句集 So 和获取故障修复后受影响的语句集 Sp. 获取故障修复前受影响的语句集 So 位于第 5 行至第 14 行，var_x^i 包含故障的变量状态，通过寻找以 var_x^i 为起点的定义操作状态转移轨迹，可以找到这些转移轨迹对应的语句集. 通过寻找以 var_x^i 为起点的影响链，可以找到影响链对应的语句集. 对这两部分语句集进行去重处理，将会得到故障修复前受影响的语句集 So.

算法 1.　与故障相关的变量影响分析信息收集算法.

输入：$CM(V)$，var_x^i，$var_x^{i'}$　//变量行为模型，修复前后变量状态

输出：S　//需要覆盖的语句集

过程：

```
1.  CM_o=CM(V).getOri();
        //得到修复故障前的变量行为模型
2.  CM_p=CM(V).getMod();
        //得到修复故障后的变量行为模型
3.  S_o=null; //修复前受影响的语句集
4.  S_p=null; //修复后受影响的语句集
5.  foreach CM_o.trace(v)!=null &&
        CM_o.trace(v).hasNext()
        //遍历变量 v 的转移轨迹
6.  trace_item=CM_o.trace(v).next;
7.  if trace_item.getN()>i then
    //判断轨迹中变量状态变化数是否比 i 大
8.      S_o.add(trace_item.getL());
    //将对应的语句加入修复前受影响的语句集中
9.  end if
10. end for
11. foreach CM_o.affectvar_x^i!=null &&
        CM_o.affectvar_x^i.hasNext() //遍历影响链
12. affect_item=CM_o.affect var_x^i.next;
13. S_o.add(affect_item.getL());
        //将对应的语句加入修复前受影响的语句集中
14. end for
15. foreach CM_p.trace(v')!=null &&
    CM_p.trace(v').hasNext() //遍历变量v'的转移轨迹
```

16. $trace\_item=CM_p.trace(v').next$;
17. if $trace\_item.getN()>i'$ then
    //判断轨迹中变量状态变化数是否比 i' 大
18. $S_p.add(trace\_item.getL())$;
    //将对应的语句加入修复前受影响的语句集中
19. end if
20. end for
21. foreach $CM_p.affectvar_x^{i'}$!=null &&
    $CM_p.affectvar_x^{i'}$.hasNext() //遍历影响链
22. $affect\_item=CM_p.affectvar_x^{i'}.next$;
23. $S_p.add(affect\_item.getL())$;
    //将对应的语句加入修复前受影响的语句集中
24. end for
25. $S=S_o-S_p$ //S 中的语句在 S_o 中,且都不在 S_p 中
26. return S;

其中第 15 行和第 24 行是获取变量影响分析模型的故障修复后受影响的语句集 $Sp.var_x^{i'}$ 是 var_x^i 修复之后对应的变量状态.通过寻找以 $var_x^{i'}$ 为起点的定义操作状态转移轨迹,可以找到这些转移轨迹对应的语句集.通过寻找以 $var_x^{i'}$ 为起点的影响链,可以找到影响链对应的语句集.对这两部分语句集进行去重处理,将会得到 Sp.最后求 So 与 Sp 的差集 $So-Sp$,即可以得到与故障相关的语句集 S.

3.3 选择种子测试用例

在对程序运行时的变量进行影响分析后,可以利用这些信息及故障修复后测试用例运行时对应的语句集合,来选择种子测试用例.种子测试用例选择算法如算法 2 所示,该算法首先需要遍历测试用例集,然后判断测试用例集中的每一个测试用例是否与需要覆盖的语句集合有交集,如果存在交集,则将该测试用例添加至种子测试用例中.

算法 2. 种子测试用例选择算法.

输入:T //测试用例集
　　S //需要覆盖的语句集
　　$Info$ //故障修复后测试用例运行时对应的语句集合
输出:T' //选择出的种子测试用例
过程:
1. foreach $i=0$; $T(i)\in T$; $i++$ //遍历测试用例集
2. if $info.getStmt(T(i))\cap S$!=null then
   //判断测试用例运行时经过的语句与 S 有交集
   $T'.add(T(i))$;
   //将测试用例加入到种子测试用例集中
3. end if
4. end for
5. return T';

3.4 数据变异

数据变异在本文中是用来结合种子测试用例生成新的测试数据的一种方法.数据变异是变异测试方法的一种,和一般变异测试需要修改程序的源代码不一样的是,数据变异不需要修改程序的源代码,它主要利用的是测试用例中的测试数据信息[12-13].

数据变异有以下一些特点:

(1)数据变异是一种类似黑盒测试的方法,不需要知道源代码的信息,也不需要知道被测系统内部的逻辑结构和实现.

(2)数据变异是一种动态测试方法,可以根据数据类型的不同,依据变异的规则来动态生成测试数据.

(3)数据变异生成的测试用例执行也可以自动化.

数据变异最关键的部分是变异算子,数据变异正是依靠变异算子,才能够实现对种子测试用例里面的测试数据进行变异,从而产生出新的测试数据.

定义 1. 变异算子(Mutation Operator,MO).

数据变异的变异算子指的是在符合语法规则的前提之下,对种子测试用例的测试数据进行修改的一些规则.变异算子可以定义为一个函数 MO,该函数能够对测试用例的输入数据进行变异.通过变异算子产生的输出为 $MOUT=MO(TD,FLD)$.在这里 TD 指的是某测试用例的测试数据,FLD 指的是测试数据中的一个字段.

在本文中针对的主要是 Java 程序,测试用例一般是 Junit 类型的.因此研究了针对 Java 程序的变异算子,并且具体提出了 52 种变异算子.在附录 1 中列出了其中部分变异算子,有些变异算子大同小异,比如 Float、Double 和 Integer 的变异算子有些类似.

定义 2. 变异规则(Mutation Rule,MR).

变异规则给出了要变异的数据字段、所选择的变异算子,以及为选择的变异算子分配变异的参数等信息.变异规则可以定义为一个函数 MR,该函数能够对需要变异的数据字段、变异算子进行选择.变异规则的形式化定义为 $MR=\langle TD,FLD,MO,MOR\rangle$.在这里 TD 指的是测试数据,FLD 指的是测试数据中的一个字段,MO 指的是变异算子,MOR 则指的是变异参数,变异参数代表了变异算子中动态可变的部分.

3.5 测试数据生成

传统的变异测试的测试用例生成方法重点关注的是得到高质量的测试用例,同时还能够有效地检测出所有非等价变异体.而数据变异测试也能够指

导测试用例的生成.不过和传统的变异测试的测试用例生成方法不一样的是,数据变异测试用例生成无需关注程序内部的细节.下面以典型三角形的例子来说明基于变量影响分析的数据变异的测试数据生成.

假设三角形的例子中已经存在了如下的测试用例:

$T1$:输入 $x=8,y=8,z=8,x,y,z$ 的类型都是整型,期待的输出是全等三角形;

$T2$:输入 $x=8,y=8,z=10,x,y,z$ 的类型都是整型,期待的输出是等腰三角形;

$T3$:输入 $x=4,y=5,z=10,x,y,z$ 的类型都是整型,期待的输出是非三角形.

在基于变量影响分析的数据变异的测试用例生成方法中,首先会进行变量影响分析,得到修复故障前后受到影响的语句集合,然后对照修复故障后测试用例的执行情况,选择那些能够执行到受影响语句集合的测试用例作为种子测试用例.

假如修复故障之后,$T1$ 执行时的语句集合中包含了受影响的语句,这时可将测试用例 $T1$ 作为种子测试用例.然后分析测试用例 $T1$ 的输入数据的类型,得到的数据类型是整型.因此可以用到 Integer 的变异算子来对该参数进行变异.

在寻找变异规则时,需要根据测试用例执行的上下文环境,观察测试数据中整型类型的数据是否存在长度限制或者取值的限制,如果存在,则从变异算子集合中挑选相关的变异算子.此外变异规则还有其他的几种模式,分别是随机选择、顺序轮换、基于权重的选择、反馈引导的选择.

随机选择指的是随机地选择变异规则.顺序轮换是按照变异算子集合中的顺序,来随机地选择变异规则.基于权重的选择是指为变异算子赋予一定的权重,然后根据权重的高低来选择变异算子.反馈引导的选择则是基于之前的测试,来为新的变异算子的选择提供依据.

这几种模式的选择可以根据测试人员的偏好,以及测试任务的要求等特定的场景确定.在选择好了变异算子之后,接下来需要利用变异算子对应的变异策略,来确定每个变异参数的取值.

比如在用测试用例 $T1$ 作为种子测试用例之后,选择出变异算子分别为 *MO-IntAdd*、*MO-IntSub*、*MO-IntZero*、*MO-IntNegVal*.假设 *MO-IntAdd* 算子中增加的值为 5、*MO-IntSub* 算子中减少的值为 5,*MO-IntNegVa* 算子中给予的负整型的值为 -1.那么通过数据变异可以得到如下测试用例(以 *MO-IntAdd* 算子为例).需要注意的是,在进行变异的时候没有考虑参数组合的情况.

$T1(1)$:输入 $x=13,y=8,z=8,x,y,z$ 的类型都是整型;

$T1(2)$:输入 $x=8,y=13,z=8,x,y,z$ 的类型都是整型;

$T1(3)$:输入 $x=8,y=8,z=13,x,y,z$ 的类型都是整型;

$T2(1)$:输入 $x=13,y=8,z=10,x,y,z$ 的类型都是整型;

$T2(2)$:输入 $x=8,y=13,z=10,x,y,z$ 的类型都是整型;

$T2(3)$:输入 $x=8,y=8,z=15,x,y,z$ 的类型都是整型;

$T3(1)$:输入 $x=9,y=5,z=10,x,y,z$ 的类型都是整型;

$T3(2)$:输入 $x=4,y=10,z=10,x,y,z$ 的类型都是整型;

$T3(3)$:输入 $x=4,y=5,z=15,x,y,z$ 的类型都是整型.

从以上可以看出,每一个变异算子应用到种子测试用例中测试数据的每一个参数时,都会产生很多变异之后的测试用例.而这其中还没有考虑参数组合的情况,此外新生成的测试数据中还包含一些非法的测试数据,因此数据变异会产生大量的新的测试用例,所以很有必要对这些测试用例的选择.

3.6　测试选择

测试用例在执行的时候,都会产生一个轨迹,这个轨迹的粒度可能是模块级别的,也可以是函数调用级别的,还有可能是语句级别的.这些轨迹经常用来作为统计覆盖率的依据.本文采用了收集语句级别的轨迹数据,利用这些数据来进行测试用例的选择.

测试用例选择的方法首先需要记录测试用例在执行时所经过的语句轨迹,对于修改后的新版本,哪些语句被执行到的信息都能够准确地得知,这样我们就可以知道哪些测试用例覆盖到了与故障相关的数据链信息,而对于那些丝毫没有执行到与故障相关的变量行为信息,将不会被选择出来.

3.7　测试判定

利用数据变异产生大量的测试数据后,接下来可对种子测试用例的测试行为进行复用,从而能够

自动化地应用数据变异测试产生的测试数据.不过对于数据变异来说,测试判定是一个难点问题,在传统的功能测试中,测试判定一般是通过比较被测系统的实际输出是否与预期一致.如果被测系统的实际输出与预期是一致的,那么则认为测试判定为pass;如果被测系统的实际输出与预期是不一致的,那么则认为测试判定为fail.

不过在数据变异测试中,被测系统对于变异之后的测试数据如何响应有时候是难以得知的,尤其是生成的测试数据是非法的测试数据时.因为有时候往往不知道程序源代码或者内部功能结构.这里有多种策略来处理这样的问题,第1种策略每次在运行一条变异测试产生的测试数据之后,再次运行所有的测试用例,看被测系统是否能够正常运行.此外一种策略是在运行一组变异测试产生的测试数据之后,检测被测系统对变异测试产生的测试数据的耐受情况,从而一起做出判定.第3种策略是在已知程序的源代码的情况下,观察程序的一些输出,从而主观地给出测试判定.

对于第3种策略,一般是采取收集程序在运行时的一些信息,来对测试进行判定的方法.例如Csallner等人[21]提出的针对鲁棒性测试时的测试判定方法,该方法主要是收集Java程序在运行时可能出现的错误或者异常信息,然后通过分析异常信息来对测试进行判定.附录2列出了部分Java程序异常信息描述.

从附录2可以看出,Java程序在运行出现异常时,有可能会抛出两大类信息中的一种.第一大类是错误(Error)信息,第二大类是异常(Exception)信息,在异常(Exception)信息中,又包括运行时异常(Runtime Exception)信息和检查异常(Checked Exceptions)信息.

由于检查异常(Checked Exceptions)通常在编译的时候会出现,指一些可以预知的,当发生异常后知道如何处理的异常,因此难以靠这样的异常信息来进行测试判定.而错误(Error)信息和运行时异常(Runtime Exception)信息则可以作为测试判定的依据.

出现错误(Error)信息的时候,表示Java程序出现了很严重的问题,而这样的问题有可能是程序中存在很严重的故障.

出现运行时异常(Runtime Exception)信息时,还需要进行进一步判断.如果运行时出现的异常(Runtime Exception)信息是底层的异常信息,那么这样的异常信息出现之后,可以认为触发了程序中的故障.如果运行时出现的异常(Runtime Exception)信息是测试用例中调用的方法或者构造器出现的异常而导致的,比如参数不合理等.这种情况有可能是数据变异时产生的数据导致了这样的问题,因此难以判断程序中是否存在故障.而如果运行时出现的异常(Runtime Exception)信息不是测试用例中调用的方法或者构造器出现的异常而导致的,那么这时候可以认为程序中存在故障.当认为程序中存在故障的时候,给出的测试判定为fail.反之给出的测试判定为pass.

本文采取的也是第3策略,并且借鉴了Csallner等人[21]提出的针对鲁棒性测试时的测试判定方法.此外,我们还在程序执行时设置了对于执行路径中方法的观察点,来收集方法的返回值(给定相同的输入,来比较原始版本的输入和修改版本的输入),通过那些观测值来辅助给出测试判定.当然,采用这样的办法有可能依然难以得到测试判定,这时候采用的办法是舍弃该测试用例,如果需要可能还要再补充新的测试用例.

4 实验与分析

4.1 实验对象

我们选择了文本比较程序Diff、Siemens基准库的程序xml-security的3个版本、1个网络购物的程序Shopping及NanoXml程序进行了测试,xml-security是xml安全协议的实现代码.这4个例子的代码行数从几百行到上万行,满足变量影响分析所需要分析的代码量.实验程序的相关信息见表2所示.

表2 实验程序

程序名称	功能描述	故障数目	测试用例数	行数
Diff	文本比较	5	50	708
xml-security_v1	XML安全协议实现	1	92	21613
xml-security_v2	XML安全协议实现	1	94	22318
xml-security_v3	XML安全协议实现	1	82	19895
Shopping	网络购物	8	36	518
NanoXml	XML解析器	3	201	4009

4.2 评价准则

为了对提出的方法进行评价,本文提出了3个评价准则,分别为:故障检测率(Fault Detect Rate,FDR)、影响语句覆盖率(Impact Statement Coverage,ISC)和测试用例的故障检测率(Used Test Case,UTC).

故障检测率 $FDR=\frac{d}{ud+d}$,其中 d 指的是检测

到的故障数目，ud 指的是没有检测到的故障数目. FDR 在一定程度上能反映出方法检测故障的能力，FDR 越高，表示该方法检测出故障的能力越强.

影响语句覆盖率 $ISC=\frac{c}{uc+c}$，其中 c 指的是覆盖到的影响语句数，uc 指的是没有覆盖到的影响语句数. ISC 在一定程度上能反映出使用方法获得的测试用例执行时覆盖语句的多少，ISC 越高，表示该方法得的测试用例能够覆盖更多的语句.

测试用例的故障检测率 $UTC=\frac{d}{tc}$，其中 d 指的是检测到的故障数目，tc 指的所有测试用例. UTC 指的是发现总的故障数与使用测试用例总数的比值. UTC 在一定程度上能反映出使用方法获得的测试用例检测故障的能力，UTC 越高，表示该方法得到的测试用例检测故障的能力越强.

UTC 与 FDR 侧重点不同，UTC 侧重在检测故障所用的测试用例的多少，FDR 侧重检测出的故障的多少.

4.3　实验结果与分析

4 组程序植入的故障、种子测试用例数、变异测试用例数的相关情况如表 3、表 4、表 5 和表 6 所示. 本文将所提出的方法(IADM)与随机测试用例生成方法(RT)和传统的测试用例生成方法(NGA)[6]进

表 3　Diff 程序中植入的故障情况

版本	故障编号	正确代码	原始故障代码	修复后植入故障数	种子测试用例数	变异测试用例数	所在文件
v1	Fault1	int newmax=newinfo.maxLine+2	int newmax=newinfo.maxLine+**8**	2	3	7	Diff.Java
	Fault2	oldline=newinfo.other[printnewline];	oldline=***oldinfo***.other[printnewline];	2	6	15	Diff.Java
	Fault3	int newlast=−1;	int newlast=**0**;	2	7	13	Diff.Java
	Fault4	int linenum=++pinfo.maxLine;	nt linenum=***pinfo.maxLine***++;	1	3	9	Diff.Java
	Fault5	pnode.linenum=linenum;	***ipnode***. linestate=linenum;	3	9	19	Node.Java

表 4　xml-security 程序中植入的故障情况

版本	故障编号	正确代码	原始故障代码	修复后植入故障数	种子测试用例数	变异测试用例数	所在文件
v1	Fault1	byte[] output=new byte[this._c14nizedBytes.length];	byte[] output=**this._c14nizedBytes**;	5	11	32	SignedInfo. Java
v2	Fault1	String qnameIS="{"+((namespaceIS==null)?"":namespaceIS)+"}"+localnameIS;	String qnameIS="{"+((namespaceIS==null)?"":namespaceIS)+"}"+localnameIS;	6	16	45	ElementProxy. Java
v3	Fault1	Element digestValueElem=this.getChildElementLocalName(0,…)	Element digestValueElem=this. getChildElementLocalName(**1**,…)	5	13	36	signature. Java

表 5　Shopping 程序中植入的故障情况

版本	故障编号	原始代码	植入故障代码	修复后植入故障数	种子测试用例数	变异测试用例数	所在文件
v1	Fault1	discount=0.95	discount=**0.90**	1	2	8	Order. java
	Fault2	discount=0.85	discount=**0.80**	2	3	9	Order. java
v2	Fault1	discount=0.90	discount=**0.92**	2	3	8	Order. java
	Fault2	discount=price*discount−100	price=price*discount−**200**	3	4	13	Order. java
v3	Fault1	discount=0.95	discount=0.90	1	2	5	Order. java
	Fault2	order.getCartPrice()+order.productCount()*100	order. productCount()*50	1	1	3	Order. java
v4	Fault1	discount=price*discount−100	price=price*discount−**200**	2	3	7	Order. java
	Fault2	order.getCartPrice()+order.productCount()*100	order. productCount()*50	1	2	7	Order. java

表 6　NanoXml 程序中植入的故障情况

版本	故障编号	原始代码	植入故障代码	修复后植入故障数	种子测试用例数	变异测试用例数	所在文件
NanoXml	Fault1	Stringvalue=XMLUtil.scanString(reader,'%', false, this.parameterEntityResolver);	Stringvalue=XMLUtil.scanString(reader,'&', false, this.parameterEntityResolver);	1	9	21	NonValidator.java
	Fault2	this.currentSystemID=new URL("file:.");	this.currentSystemID=new URL("***file***:");	1	7	17	StdXMLReader.java
	Fault3	this.encapsulatedException=e;	this.encapsulatedException=**null**;	2	12	12	XMLException.java

行对比,分别从故障检测率、影响语句覆盖率和测试用例的故障检测率进行比较,所得到的结果图 2、图 3 和图 4 所示.

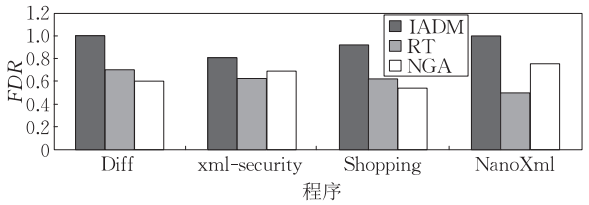

图 2 故障检测率比较

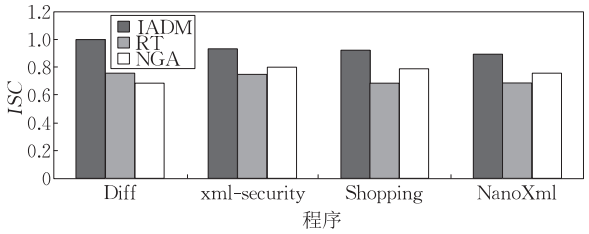

图 3 影响语句覆盖率比较

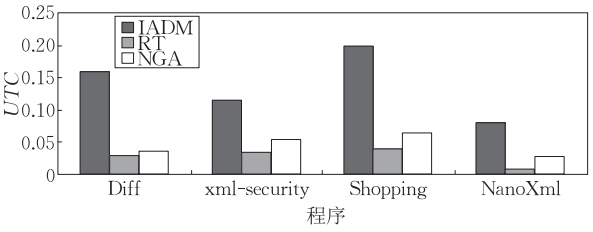

图 4 测试用例故障检测率比较

从图 2 可以看出,基于变量影响分析和数据变异的回归测试用例方法(IADM)在检测故障方面要优于随机测试用例生成方法(RT)和传统的测试用例生成方法(NGA),其中对于 Diff 程序,IADM 的故障检测率达到 1,RT 的故障检测率为 0.7,NGA 的故障检测率为 0.6;对于 xml-security 程序,IADM 的故障检测率达到 0.8125,RT 的故障检测率为 0.625,NGA 的故障检测率为 0.6875;对于 Shooping 程序,IADM 的故障检测率达到 0.9231,RT 的故障检测率为 0.6154,NGA 的故障检测率为 0.538;对于 NanoXml 程序,IADM 的故障检测率达到 1,RT 的故障检测率为 0.5,NGA 的故障检测率为 0.75.

从图 3 可以看出,基于变量影响分析和数据变异的回归测试用例方法(IADM)在影响语句覆盖率方面要优于随机测试用例生成方法(RT)和传统的测试用例生成方法(NGA),其中对于 Diff 程序,IADM 的影响语句覆盖率达到 1,RT 的影响语句覆盖率为 0.76,NGA 的影响语句覆盖率为 0.68;对于 xml-security 程序,IADM 的影响语句覆盖率达到 0.9291,RT 的影响语句覆盖率为 0.748,NGA 的影响语句覆盖率为 0.795;对于 Shooping 程序,IADM 的影响语句覆盖率达到 0.9211,RT 的影响语句覆盖率为 0.6842,NGA 的影响语句覆盖率为 0.789;对于 NanoXml 程序,IADM 的影响语句覆盖率达到 0.891,RT 的影响语句覆盖率为 0.689,NGA 的影响语句覆盖率为 0.756.

从图 4 可以看出,基于变量影响分析和数据变异的回归测试用例方法(IADM)在测试用例的故障检测率方面要优于随机测试用例生成方法(RT)和传统的测试用例生成方法(NGA),其中对于 Diff 程序,IADM 的测试用例的故障检测率达到 0.1587,RT 的测试用例的故障检测率为 0.0301,NGA 的测试用例的故障检测率为 0.0361;对于 xml-security 程序,IADM 的测试用例的故障检测率达到 0.115,RT 的测试用例的故障检测率为 0.0356,NGA 的测试用例的故障检测率为 0.0534;对于 Shooping 程序,IADM 的测试用例的故障检测率达到 0.2,RT 的测试用例的故障检测率为 0.0408,NGA 的测试用例的故障检测率为 0.0654;对于 NanoXml 程序,IADM 的测试用例的故障检测率达到 0.08,RT 的测试用例的故障检测率为 0.0096,NGA 的测试用例的故障检测率为 0.0275.

4.4 影响实验有效性因素

本文将所提出 IADM 与 RT 及 NGA 进行对比取得了较好的效果,其中故障检测率 IADM 比 RT 及 NGA 要高的原因在于:程序中所植入的故障都是与数据流相关的,且这些故障相关的变量行为模型在程序修复前后存在区别,而 RT 和 NGA 都难以检测到所有的区别. 影响语句覆盖率 IADM 比 RT 及 NGA 要高的原因也和故障检测率类似. 测试用例的故障检测率 IADM 比 RT 及 NGA 要高的原因是因为在进行数据变异之后,还有后续的选择策略,使得测试用例的规模减少,从而使得用较少的测试用例能够检测出更多的故障.

尽管采用 IADM 方法能取得不错的效果,但考虑到程序的规模还不够大,缺乏实际的工程项目进行检验. 且本文所针对的故障是数据流相关的故障,在后续的研究中,需要考虑更多大型的工程项目和更多类型的故障.

5 相关工作讨论

5.1 测试用例生成方法

测试用例生成方法有很多,随机测试用例生成

方法是其中一种.随机测试用例生成方法[22]通过在输入域中采用随机选择的方式来产生测试数据.对于简单的程序输入,随机测试用例生成方法,如整数、实数等,可直接生成随机数据.其他数据类型需要进行数据映射和数据转换.此外还有一些随机测试用例生成的改进方法,比如自适应的随机测试用例生成方法[23].该方法根据失效输入的区域特征,提出将测试用例尽可能均匀分布在整个输入空间,这样可以提高测试用例发现错误的效率.由于利用随机测试用例生成方法获得的测试用例不需要利用过多的信息,也不要求开发复杂的程序,因此该类方法的开销较低.但利用随机测试用例生成方法获得的测试用例,发现程序失效的效率不高,且难以确定测试停止的规则或条件.

基于搜索的测试用例生成方法来源于对人工智能的研究[24-25].该类方法根据以往搜索的结果所得到的信息,产生下一步的搜索动作.搜索算法是处理NP-hard问题的有效途径.常用的搜索算法有:爬山算法[26]、模拟退火算法[27]与遗传算法[28].

还有一类测试用例生成方法是基于约束求解的[29].在测试用例生成的过程中,大多数测试覆盖要求可归结为一个或多个逻辑表达式,这些逻辑表达式可看成是一些约束条件.因此,产生达到覆盖要求的测试用例的问题就转化成为求解满足约束条件的值的过程.根据约束问题中变量的值域的不同,约束问题又可以分为布尔约束问题、有限约束问题和数值约束问题.Gotlieb 等人[30]用约束求解法来生成测试数据.Gupta 等人[31]采用迭代逼近的方法,来获得满足所选择的路径上谓词的输入值.然后在每次迭代的过程中执行与选定路径相关的语句,从而得到一个线性约束集.接着利用高斯消去法来求解该约束集,从而获得一个输入增量,再进行迭代,最终产生选定路径的测试数据.

除了上述几种测试用例生成方法之外,还有基于符号执行的测试用例生成方法.符号执行是指在进行程序分析的过程中,使用符号而不是实际的值代入到程序中进行运算的过程.从而能够分析出特定路径的约束条件和计算公式[32].

基于符号执行的测试用例生成方法需要建立约束系统,从前向后替换与从后向前替换是两种典型的建立约束系统的主要方法.

5.2　数据流分析与切片分析

数据流分析是一种典型的变量行为分析方法,在编译优化中得到大量应用,使得编译器在不改变程序功能的前提下,利用数据流分析来对程序中部分代码进行适当的转换,从而提高程序的运行效率[33-37].数据流分析包括静态数据流分析与动态数据流分析,静态数据流分析是指不在计算机运行被测程序的前提下,利用静态分析提供的必要信息,检测变量的取值与使用是否发生不合理现象,即找出被测程序中是否存在变量在使用前未被取值、变量在两次取值之间未被使用、一个赋了值的变量是否未被使用等异常情况[33].动态的数据流分析是指以程序中的数据流关系作为测试的需求,这些数据流的关系指的是变量的定义和使用关系.基于针对数据流的测试覆盖准则,来生成满足覆盖准则的测试数据.

对程序中的方法调用进行的数据流分析称为方法间数据流分析,对全局变量进行的数据流分析称为全局数据流分析.文献[38-41]对这些分析方法进行了深入的研究.在结构化程序中,由于指针传递和引用传递引起的多个变量可以操作内存中同一块数据单元,Burke 和 Choi[42]对其进行了深入的探讨,目前可以解决某些简单形式的问题.数组中的各元素可以被看作独立的变量,但对数组中各元素的操作很难被静态的区分出来.对于数组变量的数据流分析,也有不少的研究,有一种处理方法,可以对数组元素的操作进行部分的区分.静态数据流分析需要提取和分析的信息量很大,所以其时间和空间效率有点低.

动态数据流分析技术则是静态数据流技术的补充.这种技术主要是针对某些特殊的变量(或信息),如数组元素、结构变量(*struct*)、指针变量和动态变量等.和它们相关的具体信息一般只有在程序运行时才能确定.

基于数据流的软件测试的研究工作一直是软件测试研究的重要领域之一,有着极其重要的理论和实际应用价值.从数据流覆盖角度,相关研究设计了一系列的覆盖准则来指导生成满足覆盖准则的测试数据.

切片也是一种典型的变量行为分析技术,Weiser 博士[19]定义了程序切片为程序中所有能够影响某个输出的语句集合,如果删除其他语句,不会对所关注的输出有任何影响.切片分为静态切片、动态切片、前向切片和后向切片等.如果在计算程序切片时不考虑程序的具体输入,只考虑对某个关注点影响的语句集合,得到的切片就是静态切片.如果在计算程序切片时考虑程序的具体输入,在特定输入下,影

响相关变量 v 在 n 点取值的语句集合,得到的切片就是动态切片[43].后向切片是指构造一个集合 $affect(v,n)$,使得这个集合由所有影响变量 v 在 n 点取值的语句组成.后向切片常用于在测试发现错误时来分析前面哪条语句导致了相应错误.前向切片正好相反,是指构造一个集合 $affect(v,n)$,由受到变量 v 在位置 n 取值所影响的所有语句构成.前向切片也有广泛的应用,如可用于检查程序哪些语句受到故障修复的影响.

6 结论与展望

本文对程序中变量的动态行为进行了分析,提出了一种变量行为模型,并且采用变量影响分析和数据变异的方法,对回归测试用例生成进行了研究,实验表明本文所提出的方法有效.但回归测试用例生成是一个颇具难度的问题,仅靠本文所提出的一种方法还不够,特总结了后续可以进一步改进的方面.

完善变量行为模型,即在已有的基础上增加程序控制流相关的信息,这样可针对更多类型故障的修复检验.此外,本文在利用数据变异生成测试用例的方法中,采用了捕获异常信息来给出测试判定的方法,但这样的方法事实上是不够精确的,在后续的研究中,可以对利用数据变异生成测试用例中如何给出测试判定进行深入的研究.

致 谢 感谢北京航空航天大学软件工程研究所模型驱动小组、分布式小组此前所做的工作.感谢百忙之中评阅论文的各位专家!

参 考 文 献

[1] Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment//Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02). New York, USA, 2002: 97-106

[2] Santelices R, Chittimalli P K, Apiwattanapong T, et al. Test-suite augmentation for evolving software//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. Vancouver, Canada, 2008: 218-227

[3] Santelices R. Automated scalable test-suite augmentation for evolving software//Proceedings of the 31st International Conference on Software Engineering-Companion Volume. Vancouver, Canada, 2009: 379-382

[4] Jiang B, Tse T H, Grieskamp W, et al. Regression testing process improvement for specification evolution of real-world protocol software//Proceedings of the Quality Software (QSIC), 2010 10th International Conference on Quality Software. Zhangjiajie, China, 2010: 62-71

[5] Xu Z, Rothermel G. Directed test suite augmentation//Proceedings of the Software Engineering Conference. Penang, Malaysia, 2009: 406-413

[6] Maragathavalli P, Kanmani S, Kirubakar J S, Sriraghavendrar P. Automatic program instrumentation in generation of test data using genetic algorithm for multiple paths coverage//Proceedings of the 2012 International Conference on Advances in Engineering, Science and Management (ICAESM). Nagapattinam, India, 2012, (30-31): 349-353

[7] Nachiyappan S, Vimaladevi A, SelvaLakshmi C B. An evolutionary algorithm for regression test suite reduction//Proceedings of the 2010 International Conference on Communication and Computational Intelligence. Erode, India, 2010: 503-508

[8] Zhang Z, Huang J, Zhang B, et al. Regression test generation approach based on tree-structured analysis//Proceedings of the 2010 International Conference on Computational Science and Its Applications. Fukuoka, Japan, 2010: 244-249

[9] Ye N, Chen X. Automatic regression test selection based on activity diagrams//Proceedings of the 2011 5th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C). Jeju Island, Korea, 2011: 166-171

[10] Kumar A, Tiwari S, Mishra K K, et al. Notice of retraction generation of efficient test data using path selection strategy with elitist GA in regression testing//Proceedings of the 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT). Chengdu, China, 2010: 389-393

[11] Yoo S, Harman M. Test data regeneration: generating new test data from existing test data. Software Testing, Verification and Reliability, 2012, 22(3): 171-201

[12] Wu Chuan, Gong Dun-Wei. Evolutionary generation of test data for regression testing based on path correlation. Chinese Journal of Computers, 2015, 38(11): 2247-2261(in Chinese)
(吴川,巩敦卫. 基于路径相关性的回归测试数据进化生成. 计算机学报, 2015, 38(11): 2247-2261)

[13] Xie Xiao-Yuan, Xu Bao-Wen, Shi Liang, Nie Chang-Hai. Genetic test case generation for path-oriented testing. Journal of Software, 2009, 20(11): 3117-3136(in Chinese)
(谢晓园,徐宝文,史亮,聂长海. 面向路径覆盖的演化测试用例生成技术. 软件学报, 2009, 20(11): 3117-3136)

[14] Zhang Zhi-Yi, Chen Zhen-Yu, Xu Bao-Wen, Yang Rui. Research progress on test case evolution. Journal of Software, 2013, 24(4): 663-674(in Chinese)

(张智轶，陈振宇，徐宝文，杨瑞. 测试用例演化研究进展. 软件学报，2013，24(4)：663-674)

[15] Santelices R，Jones J A，Yu Y，et al. Lightweight fault-localization using multiple coverage types//Proceedings of the IEEE 31st International Conference on Software Engineering. Vancouver，Canada，2009：56-66

[16] Yu K，Lin M，Gao Q，et al. Locating faults using multiple spectra-specific models//Proceedings of the 2011 ACM Symposium on Applied Computing. Taichung，China，2011：1404-1410

[17] Yang Bo，Wu Ji，Liu Chao. Software fault localization based on data chain. Journal of Software，2015，26(2)：254-268(in Chinese)
(杨波，吴际，刘超. 基于数据链的软件故障定位方法. 软件学报，2015，26(2)：254-268)

[18] Rosen B K. High-level data flow analysis. Communications of the ACM，1977，20(10)：712-724

[19] Weiser M. Program Slicing：Formal，Psychological and Practical Investigation of an Automatic Program Abatraction Method[Ph. D. dissertation]. University of Michigan，Ann Arbor，USA，1979

[20] Reps T，Ball T，Das M，Larus J. The use of program profiling for software maintenance with applications to the year 2000 problem//Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC'97/FSE-5). New York，USA，1997：432-449

[21] Csallner C，Smaragdakis Y. JCrasher：An automatic robustness tester for Java. Software：Practice and Experience，2004，34(11)：1025-1050

[22] Chan K P，Chen T Y，Towey D. Restricted random testing// Proceedings of the 7th European Conference on Software Quality. Lecture Notes in Computer Science. Helsinki，Finland，2006：321-330

[23] Jones B，Sthamer H，Yang X，et al. The automatic generation of software test data sets using adaptive search techniques// Proceedings of the 3rd International Conference on Software Quality Management. Seville，Spain，1995：435-444

[24] Tracey N J. A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software[Ph. D. dissertation]. University of York，Yorkshire，UK，2000

[25] Harman M，Jones B F. Search-based software engineering. Information and Software Technology，2001，43(14)：833-839

[26] Harman M，McMinn P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation//Proceedings of the 2007 International Symposium on Software Testing and Analysis. London，UK，2007：73-83

[27] Metropolis N，Rosenbluth A W，Rosenbluth M N，et al. Equation of state calculations by fast computing machines. The Journal of Chemical Physics，2004，21(6)：1087-1092

[28] Pargas R P，Harrold M J，Peck R R. Test-data generation using genetic algorithms. Software Testing Verification and Reliability，1999，9(4)：263-282

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

[30] Gotlieb A，Botella B，Rueher M. Automatic test data generation using constraint solving techniques//Proceedings of the ACM SIGSOFT Software Engineering Notes. Florida，USA，1998：53-62

[31] Gupta N，Mathur A P，Soffa M L. Automated test data generation using an iterative relaxation method//Proceedings of the ACM SIGSOFT Software Engineering Notes. Florida，USA，1998：231-244

[32] Boyer R S，Elspas B，Levitt K N. SELECT：A formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices，1975，10(6)：234-245

[33] Rosen B K. High-level data flow analysis. Communications of the ACM，1977，20(10)：712-724

[34] Ryder B G，Paull M C. Elimination algorithms for data flow analysis. ACM Computing Surveys，1986，18(3)：277-316

[35] Bodik R，Gupta R，Soffa M L. Complete removal of redundant expressions//Proceedings of the ACM SIGPLAN Notices. Florida，USA，1998：1-14

[36] Khedker U P，Dhamdhere D M. A generalized theory of bit vector data flow analysis. ACM Transactions on Programming Languages and Systems，1994，16(5)：1472-1511

[37] Andrew W A，Jens P. Modern Compiler Implementation in Java. 2nd Edition. University Printing House Shaftesbury RoadCambridge CB2 8BS，United Kingdom：Cambridge University Press，2002

[38] Fosdick L D，Osterweil L J. The detection of anomalous interprocedural data flow//Proceedings of the 2nd International Conference on Software Engineering. California，USA，1976：624-628

[39] Fosdick L D，Osterweil L J. Data flow analysis in software reliability. ACM Computing Surveys，1976，8(3)：305-330

[40] Atkinson D C，Griswold W G. Implementation techniques for efficient data-flow analysis of large programs//Proceedings of the IEEE International Conference on Software Maintenance. Washington，USA，2001：52-61

[41] Burke M，Choi J D. Precise and efficient integration of interprocedural alias information into data-flow analysis. ACM Letters on Programming Languages and Systems，1992，1(1)：14-21

[42] Feautrier P. Dataflow analysis of array and scalar references. International Journal of Parallel Programming，1991，20(1)：23-53

[43] Agrawal H，Horgan J R. Dynamic program slicing. ACM SIGPLAN Notices，1990，25(6)：246-256

附录 1 部分变异算子列表.

数据类型	变异算子	描述
Integer	*MO-IntAdd*	增加参数的值
	MO-IntSub	减少参数的值
	MO-IntNull	设置参数的值为空
	MO-IntZero	设置参数的值为 0
	MO-IntUpBod	设置参数值为超越上限的值
	MO-IntLowBod	设置参数的值为超越下限的值
	MO-IntNegVal	设置参数的值为非法的值
String	*MO-StrAdd*	增加参数的值
	MO-StrSub	减少参数的值
	MO-StrNull	设置参数的值为空
	MO-StrOverflow	设置参数的值为非法的值
	MO-StrMaxLen	设置参数的值为非法的值
	MO-StrMinLen	设置参数的值为非法的值
	MO-StrReplace	用其他的值来代替参数之前的值
ArrayType	*MO-ArrAdd*	增加数组中的值
	MO-ArrSub	减少数组中的值
	MO-ArrNull	设置数组的值为空
	MO-ArrReplace	用其他的值来代替参数之前的值
	MO-ArrOversize	改变数组的长度
ReferenceType	*MO-RefAdd*	增加引用变量中子域的值
	MO-RefNull	设置子域的值为 NULL
	MO-RefMod	修改引用变量中子域的值

附录 2 部分 Java 程序异常

异常类型		异常类名	描述
Error		AbstractMethodError	调用抽象方法错误
		ClassFormatError	类文件格式错误
		IllegalAccessError	非法访问错误
		InternalError	Java 内部错误
		LinkageError	连接失败所产生的错误
		ThreadDeathError	线程死亡错误
		UnknownError	未知错误
		UnstatisfiedLinkError	链接条件没被满足的错误
		VirtualMachineError	虚拟机错误
Exception	RuntimeException	ClassCaseException	类强制转换异常
		ArrayIndexOutOfBoundsException	数组索引越界异常
		NullPointerException	空指针异常
		SecurityException	安全异常
	Checked exceptions	ClassNotFoundException	找不到类异常
		IllegalAccessException	违法的访问异常
		NoSuchFieldException	属性不存在异常

YANG Bo, born in 1981, Ph. D., lecturer. His main research interests focus on software testing and data mining.

WU Ji, born in 1974, Ph. D., associate professor. His main research interests include software testing and software reliability analysis.

LIU Chao, born in 1958, Ph. D., professor, Ph. D. supervisor. His main research interests include software engineering and software testing.

Background

Program contains a lot of faults are tight related with data flow, which is specified as the sequence of definitions and usages of variables. Manual debugging is often used to find fault by watching variable definition, variable use, dependencies between variables and a variety of operating impacts on the value of a variable. Some studies have used program variables as an important element of concern in test case generation, such as DU-Pairs coverage approach and program slicing approach. DU-Pairs coverage approach analyzes the covered DU-Pair in the program execution process and statistic the frequency of the appearance of DU-Pair. However, DU-Pairs coverage approach does not take account of the relationship between different variables, which affects the test case efficiency. Program slicing is a kind of analysis techniques to identify the program statements with dependency analysis. However, the data obtained from program slicing technology are enormous and disordered. This method is often cost-consuming.

This paper presents a regression test case generation approach based on a model of variables at runtime-variable behavior model, the model will take account of variables change information and dependencies information. On the basis of this model, this paper proposes an approach to guide the automatic test case generation. These test cases make the regression testing better. We evaluated the proposed approach by performing experimental studies on four programs and making a conclusive evaluation that the proposed approach can effectively generate test cases with high defect detection ability to the data flaw related faults.

This research is supported by the National Science Foundation of China (Nos. 61502011, 61370051, and 61402016), the Scientific Research Project of Beijing Educational Committee (No. KM201610009007).