文章编号：2095-2163(2023)06-0142-07 中图分类号：TP311.5 文献标志码：A

基于历史覆盖信息的回归测试用例动态生成

栗晓雪[1]，赵逢禹[2]
（1 上海理工大学 光电信息与计算机工程学院，上海 200093；
2 上海出版与印刷高等专科学校 信息与智能工程系，上海 200093）

摘 要：随着软件产品的演化频率越来越高，软件的每次修改或集成都需要大量的回归测试，以验证软件维护是否带来了新的问题。针对于回归测试用例集构建的问题，本文提出一套回归测试用例集生成方法，该方法包括回归测试用例集选择与回归测试用例集动态生成两部分。基于已有测试用例集的历史覆盖信息，选择与程序更改相关的测试用例，形成回归测试用例集选择；通过对改动的部分程序进行插桩，并执行回归测试用例选择集以获得其动态执行信息，并基于路径约束表达式的求解，生成了新的测试用例。通过对 6 个开源程序的实验，验证了所提方法的有效性与合理性。

关键词：回归测试用例选择；回归测试用例生成；代码插桩；约束求解

Dynamic generation of regression test cases based on historical coverage information

LI Xiaoxue[1], ZHAO Fengyu[2]
(1 School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China; 2 Department of Information and Intelligent Engineering, Shanghai Publishing and Printing College, Shanghai 200093, China)

【Abstract】 As software products evolve with increasing frequency, each software modification or integration requires extensive regression testing to verify whether software maintenance introduces new problems. Aiming at the problem of regression test case set construction, this paper proposes a regression test case set generation method, which includes two parts: regression test case set selection and regression test case set dynamic generation. Based on the historical coverage information of existing test cases, test cases related to program changes are selected to form regression test case selection. The dynamic execution information of the modified program is obtained by staking and implementing regression test case selection set, and new test cases are generated based on the solution of path constraint expression. The validity and rationality of the proposed method are verified by experiments on six open-source programs.

【Key words】 regression test case selection; regression test case generate; code instrumentation; constraint solving

0 引 言

在软件的演化和维护过程中，往往需要修改软件中的错误、增加新的功能、调整软件的配置等需求，导致代码不断被修改。在代码修改完后，需要使用回归测试来检查软件中的缺陷，避免代码的修改给软件带来新的错误。此外，当已有测试用例集不充分时，还需要针对新的功能与代码部分设计新的测试用例。因此，在软件的持续演化过程中，测试用例集合的规模逐渐扩大，导致回归测试用例集的构建成为一项复杂的工作。有研究表明，回归测试的开销占整个软件测试预算的 80%以上，并占整体维护预算的 50%以上[1]。因而，研究并提出一套有效且经济的回归测试用例集的构建方案是十分有意义的。

不管是回归测试用例的选择还是回归测试用例的生成都是国内外学者关注的课题。文献[2]提出了一套基于测试用例能够检测的故障程度来选择回归测试用例的方法，该文首先运行已有的测试用例，将发现的故障与故障程度记录在日志文件中，然后选择故障程度较高的测试用例进入下一轮的回归测试。文献[3]提出一种根据回归测试目标自动调整优化策略的测试用例选择方法。该方法将测试用例加上标识属性，如缺陷检测数表示该测试用例历史

作者简介：栗晓雪(1996-)，女，硕士研究生，主要研究方向：软件测试；赵逢禹(1963-)，男，博士，教授，CCF 成员，主要研究方向：软件测试。
通讯作者：赵逢禹 Email：zhaofengyv@ usst.edu.cn

收稿日期：2022-07-09

执行中失败的总次数,用重要性因子表示该测试用例对于当前测试需求的重要程度,新旧功能标识表示了该测试用例是否属于新增或删除模块。根据阶段的回归测试目标,将测试用例按照属性标识自动优化排序,根据优化排序结果选择测试用例集。文献[4]提出了一种静态多重关联的回归测试用例构造方案,通过分析方法间的调用关联和隐式数据关联进而构建多重方法关联图,并依据该图中的关联关系,选择因代码更改而受影响的回归测试用例集。文献[5]提出基于偶然正确性概率的回归测试选择方法,删减掉可能发生偶然正确性现象的测试用例,提高测试用例的检错能力,进一步缩减回归测试用例集的规模。所谓偶然正确性是指程序中包含错误的语句被执行但仍通过了测试的现象。

当回归测试中选择的测试用例不充分时,需要生成新的测试用例。符号执行作为一种重要的程序分析技术,可以为程序测试提供高覆盖率的测试用例[6]。符号执行又分为静态符号执行和动态符号执行。静态符号执行使用符号值代替一类实际数值,作为程序的输入并执行程序,在程序执行的过程中收集路径约束和符号状态。最终,在程序执行结束后,得到一条完整的路径约束方程,使用约束求解器对其求解便可得到一条与符号执行时相同路径的测试用例。随着程序的复杂化,出现了“路径爆炸”和复杂的外部调用等导致静态符号执行收集到的路径约束方程无法进行求解。为了解决上述问题,Cristian Cadar[7]提出了动态符号执行的技术。当静态符号执行遇到约束方程无法求解的情况,利用具体值代替符号值,也就是结合具体值与符号值共同执行程序。Artzi 等人[8]开发的 web 缺陷检测工具 Apollo,就使用了动态符号执行的技术生成缺陷检错能力更高的测试用例集。在 Apollo 工具中,Artzi 等对动态符号执行技术做了改进,为了解决动态符号执行中路径爆炸和生成冗余测试用例的问题,对路径约束相似的路径进行剪枝处理,对由约束求解器求得的相似测试用例进行判断是否冗余,缩小了测试用例的规模,提高软件测试的效率。

当前的回归测试用例生成技术多集中于如何高效地从已有测试用例集中选择测试用例来覆盖程序的改动部分。但是,由于代码的修改可能会调整代码的逻辑,仅从历史测试用例集中选取部分测试用例还是不够的,需要在所选测试用例集的基础上,进一步完善测试用例。而实际上,已有测试用例集的执行信息包含了覆盖路径、路径约束等信息,基于这些信息更能够准确地分析所选测试用例集覆盖程序的不完全性,进而构建更完善、准确的回归测试用例集。

本文提出的基于测试用例的执行信息构建回归测试用例集方法,首先从原测试用例集中,根据测试执行时,测试用例覆盖程序的方法,选择一部分覆盖改动方法的测试用例作为回归测试用例选择集;然后,对修改的部分源程序进行插桩,并执行所有已选择的测试用例,由插桩语句获取回归测试用例选择集的动态执行信息;最后,整理并分析插桩得到的信息,找出未覆盖的程序路径,提取未覆盖路径的逻辑表达式组成该路径下的路径约束表达式,并对其进行求解,得到的结果即是新的测试用例。

1　方法描述

1.1　回归测试用例集选择

由于已有的测试用例是已经执行过的,所以本文假设已经获得了已有测试用例覆盖程序方法的信息,根据此信息构建测试用例覆盖程序方法的矩阵。而代码经过修改后,形成了代码的变更信息,可以得到改动后的方法集合。又因为程序代码的修改会导致程序的调用发生变化,所以所有直接或间接调用修改方法(包括增加的新方法)的方法,也属于变更的方法,都应该在回归测试中被重新测试。本文将改动方法与受改动方法影响的方法集合称为该改动相关方法集。

假设在已有的测试用例库 T 中,存在某一测试用例 ti(ti∈T),程序方法 mi(mi∈M),若执行 ti 的测试路径覆盖 mi,测试用例覆盖程序方法矩阵中对应值则为 1,否则为 0。覆盖改动相关方法集的所有测试用例构成回归测试用例选择集。表 1 给出了一个从已有测试用例中选择测试用例的示例。

表 1　测试用例覆盖程序方法矩阵

Tab. 1　Test case overlay method matrix

	t1	t2	t3	t4	t5	t6
m1	1	0	1	1	0	1
m2	0	1	0	0	0	1
m3	0	1	1	0	1	0
m4	0	0	1	1	0	1
m5	0	0	0	0	0	1

假设方法 m3 是程序经过维护后修改的方法,而受调用关系,m2 和 m5 是方法 m3 的相关方法,所以选择的测试用例集需要覆盖的方法集合是{m2,m3,m5}。最终,回归测试用例选择集为{t2,t3,t5,

t6 }。

1.2 动态执行信息的收集

程序经过修改后,代码逻辑发生改变,从已有测试用例集中选择的测试用例很有可能不能完全覆盖程序所有的路径,导致构建的回归测试用例选择集不充分。为了建立更完整的测试用例集,需要先执行回归测试用例选择集,通过动态执行信息分析是否存在未覆盖的程序路径,并基于执行结果生成新的测试用例。所以,收集选择测试用例动态执行信息的目的有两个,一是分析回归测试用例选择集是否有未覆盖的程序路径;二是获取未覆盖路径的路径约束表达式,通过求解生成新的测试用例。

为了收集回归测试用例选择集的动态执行信息,首先需要对源程序进行静态分析识别出程序修改的部分,然后在修改的部分程序中按照一定的规则进行代码插桩,插桩输出的结果包括执行测试用例的执行语句编号、逻辑表达式、逻辑表达式的值、逻辑表达式中各变量的值。本文定义了以下插桩规则以获得和记录上述信息。

为了实现插桩的功能,插桩位置需要设置在修改后的部分程序中的逻辑判断语句处和代码中顺序执行的多个语句后。需要特别说明的是,对于没有改动的部分程序来说,即使存在逻辑判断表达式或循环语句也不需要对此进行插桩。

插桩规则:

(1)顺序执行的语句块:输出语句块标识。

(2)分支语句:输出执行语句编号、逻辑表达式、逻辑表达式的值、逻辑表达式的各变量值。

(3)循环语句:输出执行语句编号、循环中的逻辑表达式、循环中逻辑表达式的值、循环中的变量值。

以下是一段简单的程序代码,并假设 testme 方法中的 if 判断语句即第 6 行代码发生了更改。下面以此例展示本文的插桩方法以及插桩后输出的信息。

```
1   int twice (int v) {
2   return 2 * v;
3   }
4 void   testme   (int x , int y){
5            z=twice(y);
6 if (z==x){
7                  if(x>y+10){
8 printf("%s",error) ; }
9                   else {}
10  }
11          else {}
12  }
13   int main(){
14 printf("please input two number:\n");
15              int x,y ;
16 scanf("%d%d",&x,&y);
17 testme(x , y);
18              return 0;
19 }
```

由于第 6 行代码是更改后的语句,所以 testme 是变更方法。受调用关系影响,主方法 main 是变更方法的相关方法,所以方法 testme 和方法 main 都需要重新被测试,由此选择的部分测试用例也必须覆盖上述两个方法,所以这两个方法也是选择进行插桩的部分程序。

根据插桩规则,main 属于顺序执行的语句块,即在该方法的出口处进行插桩输出该方法的方法块标识。而 testme 方法中调用的 twice 方法不受更改代码的影响,所以不进行插桩处理。最终,testme 方法中需要进行插桩的地方是代码第 6、7、8、9、11 行处,由于上述代码行属于逻辑判断结构,所以由插桩语句输出执行语句编号、逻辑表达式、逻辑表达式的值、逻辑表达式值中各变量的值即可。最终 testme 方法经过插桩后的部分代码如下所示。

```
6 if (z == x)      {printf("%d,%s,%s",6, z == x,T); printf ("z = %d,x = %d",z,x);
7   if (x > y + 10) {printf("%d,%s,%s",7, x > y + 10,TT); printf ("x = %d,y = %d",x,y);
8 printf("%s",error) ; printf{("%d ",8 )}
9   else   { printf("%d,%s,%s",9, x > y + 10,TF);}
10  }
11else   { printf("%d,%s,%s",11, z == x,F) }
12  }
```

以测试用例{$x = 30, y = 15$} 作为输入,执行上述插桩后的程序,得到的插桩结果如下方代码所示:

```
6   z == x      T     z = 30 x = 30
7   x > y + 10   TT   x = 30 y = 15
8
```

如第一行输出数据中,6 表示了当前执行的选择测试用例的语句编号,"$z == x$" 是执行的逻辑表达式,T 表示该逻辑表达式的值为真,"$z = 30, x =$

30”是逻辑表达式中的变量值。

1.3　回归测试用例集生成

本文基于插桩获得的信息分析回归测试用例选择集中各测试用例所覆盖程序路径的情况，通过算法比对得出未覆盖的程序路径，然后对未覆盖的路径生成新的测试用例，形成回归测试用例生成集。

为了找出回归测试用例选择集未覆盖的程序路径，首先需要对插桩得到的信息进行整理，分析被修改的方法中所有逻辑表达式的真假分支是否都被执行，而没有执行的分支就组成了未覆盖的程序路径。

得到未覆盖的程序路径之后，需要根据回归测试用例选择集的动态执行信息，比对、查找出未覆盖路径上的逻辑表达式与逻辑表达式的值。假如未覆盖路径所对应的逻辑表达式的值为 T，那么只需要令当前逻辑表达式为 T，然后对该约束求解，找出满足该约束表达式的各变量值。

通过求解路径约束表达式，可以得到各变量具体的值，进而构建测试用例，使之能测试执行到未覆盖的程序路径。但是，在一些特殊的情况下，例如在路径约束表达式中的变量需要一系列复杂计算或者该变量取值网络上的数据时，在生成测试用例的输入值时，需要进一步分析如何确保路径约束表达式中的变量能够取到限定值，使测试执行到未覆盖路径。

下面以 1.2 节中的程序为例，说明生成新测试用例的过程。现假设表 2 是上述程序经过修改后，通过选择形成的回归测试用例选择集。将这些测试用例作为输入，执行插桩后的程序，然后对插桩输出结果进行处理，得到如表 3 所示的动态执行信息表。

表 2　回归测试用例选择集信息

Tab. 2　Information about selected test cases

测试用例编号	测试用例	测试输出
T1	(0,1)	0
T2	(30,15)	error

表 3　回归测试用例选择集的动态执行信息

Tab. 3　Test case execution information

测试用例	语句号(ID)	逻辑判断条件(S)	真假值(P)	变量值(V)
T1	6,11	A($z == x$)	A(F)	$x = 0, y = 1$
T2	6,7,8	A($z == x$), B($x > y + 10$)	A(T),B(T)	$x = 30, y = 15$

从表 3 中可以看到，插桩信息中出现了两个逻辑表达式分别为 A($z == x$) 和 B($x > y + 10$)，其中逻辑表达式 A($z == x$) 出现了 A(F) 和 A(T) 两个逻辑表达式的值，说明该逻辑表达式中的真假分支均已被执行到，而逻辑表达式 B($x > y + 10$) 只出现了 B(T) 说明该逻辑表达式只有逻辑为真的分支被执行到，逻辑为假的分支 B(F) 并没有被执行到，所以回归测试用例选择集未覆盖的程序路径包含该 B(F) 分支。

为了得到完整的未覆盖路径的约束表达式，可基于回归测试用例选择集的动态执行信息构建路径覆盖图，基于路径覆盖图查找该部分程序块中所有使程序执行到未覆盖路径的逻辑表达式。以表 3 为例，构建的路径覆盖如图 1 所示。由图 1 可知，程序执行到未覆盖的路径经历了两个逻辑表达式分别为 A($z == x$) 和 B($x > y + 10$)，对应的逻辑表达式的值分别为 T 和 F，所以最终组成的未覆盖路径的约束表达式为($z == x$) $\wedge$ [$\neg$ ($x > y + 10$)]，对此进行求解即可得到一组使测试执行到未覆盖路径的变量值，如$\{x = 20, y = 10\}$。

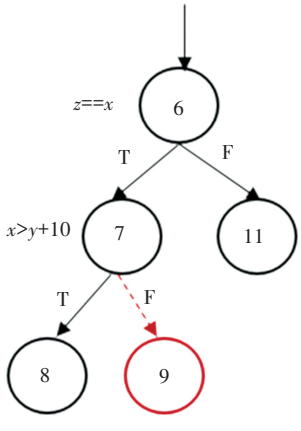

图 1　路径覆盖图

Fig. 1　Path coverage diagram

2　实验研究

2.1　实验对象

前文给出了对修改代码路径覆盖的回归测试用例集生成方法。为了验证本文所提方法的有效性，选取了 4 个 C 语言编写的基准程序和两个 C#语言编写的 web 程序构建实验，这些程序常被用于软件

测试研究领域[9-10]。以上程序均采用了不同的维护方法,形成了各试验程序的不同版本。表 4 列出了每一个实验程序的名称、简要概述、方法个数、代码行数、测试用例数和维护类型。

表 4 实验程序信息

Tab. 4 Experimental program information

程序	描述	方法	代码行数	测试用例	维护类型
Bubble	Bubble Sort	1	29	96	修改
Triangle	Triangle types	3	42	26	修改
Median	Find median	4	37	94	修改
Nextday	Calculate date	1	83	277	修改
Financial management	Account book	40	1 026	1 160	增加
Course management	Course selection	65	1 734	1 493	删除

2.2 实验设计与分析

回归测试用例集生成方法主要有两部分,一部分是当程序经过变更后,从已有测试用例集中选择能够覆盖全部改动方法的测试用例。另一部分是对已选测试用例集所未能覆盖的程序路径生成新的测试用例。所以实验需要验证的目标有两个,一是回归测试用例选择集的合理性;二是针对于程序未覆盖的路径,是否能够生成新的且正确又有效的测试用例,即回归测试用例生成集的正确性。

2.2.1 回归测试用例选择集的合理性

回归测试用例选择集的合理性是指凡是覆盖程序改动相关方法的测试用例都应该被选入回归测试用例选择集,否则不应该选入回归测试用例选择集。

实验步骤:

(1)根据已有测试用例集的历史执行信息构建已有测试用例集覆盖程序方法的矩阵。

(2)根据程序代码的变更信息定位程序改动的方法,并通过分析方法间的调用关系,找出所有与改动相关的方法。

(3)从已有测试用例覆盖程序的矩阵中选择覆盖程序改动部分的测试用例进入回归测试用例选择集。

实验结果:

表 5 是回归测试用例选择集的结果,主要展示信息有:程序的名称、原测试用例集的数量、回归测试用例选择集的数量,其中,回归测试用例选择是在已有测试用例集合的基础上,选择覆盖程序改动部分的测试用例。例如 Financial management 程序,在对程序进行增加类型的维护后,受更改影响的相关方法共有 4 个,根据已有测试用例覆盖程序方法的关系共选择了 90 个测试用例进入回归测试。

表 5 回归测试用例选择集信息

Tab.5 Test case selection set information

实验程序	原测试用例	选择的测试用例
bubble	106	106
triangle	126	53
median	216	108
nextday	377	125
Financial management	1 160	90
Course management	1 493	63

验证分析:

为了验证前文所提测试用例选择方法的合理性,实验采取了对部分改动程序插桩的方法以记录回归测试用例选择集的执行信息,如果回归测试用例选择集中的所有测试用例都执行了程序的改动部分并且未选的测试用例都不执行程序的改动部分,便认为回归测试用例选择集是合理的。

表 6 是测试用例选择集的结果验证,主要信息有程序名称、代码改动指因改动而受影响的程序部分、回归测试选择集覆盖的方法集合。通过收集回归测试用例选择集的执行信息即覆盖程序改动方法的情况,得出了回归测试用例选择集中所有的测试用例都执行了程序的改动部分。并且实验采取了相同的方法收集了未选入回归测试用例选择集的测试用例集合的执行信息,结果显示未入选的测试用例集合皆未执行程序的改动部分。由此得出回归测试用例选择集是合理的。

表 6 回归测试用例选择结果验证

Tab. 6 Test case selection verifies results

实验程序	代码更改	回归测试用例选择集覆盖的方法
bubble	3	Method1(main)
triangle	2	Method1(main),Method2(arrays.sort) Method3 (triangle)
median	3	Method1(main),Method2(medianFinder) Method3(select),Method4(partition)
nextday	6	Method1(main)
Financial management	5	Method1(login),Method2(selectSpend) Method3(addspend),Method4(modifyspend) Method5(Mschange)
Course management	4	Method1(addcourse),Method2(selectcourse) Method3(searchcourse),Method4 (ShowCurriculum)

2.2.2 回归测试用例生成集的正确性

回归测试用例生成集的正确性指生成的测试用例集覆盖了测试用例选择集中没有覆盖的全部路径。因而,通过执行回归测试用例生成集,并记录其执行的路径便可验证回归测试用例生成集的正确性。

实验步骤：

(1)整理回归测试用例选择集的动态执行信息，并通过对动态执行信息的分析找出未覆盖的程序路径。

(2)对程序进行向上回溯，找到所有使程序执行到未覆盖路径的逻辑判断表达式构建未覆盖程序路径的约束表达式。

(3)求解得到的路径约束表达式，若有解即是一条新的测试用例。

实验结果：

表 7 是回归测试用例生成集的结果，主要信息有实验程序名称、未覆盖路径、回归测试用例生成集、回归测试用例生成集覆盖的路径。从表 7 中可以看出，前文的测试用例生成方法针对于每个实验程序中的未覆盖路径都有新生成的测试用例。通过执行新生成的测试用例并记录其执行信息，得到了测试用例的路径覆盖情况。实验证实，新生成的测试用例所覆盖的路径皆是回归测试选择集没有执行过的程序路径。

表 7　回归测试用例生成集结果

Tab. 7　Test case generation set results

实验程序	未覆盖路径	回归测试用例生成集	覆盖的路径
bubble	1	1.(1,2,3,4,5) 2.(1,2,2,3,4)	A(F):array[j]≤array[j+1]
triangle	1	1.(1,1,$2^{1/2}$) 2.($2^{1/2}$,1,1) 3.(1,$2^{1/2}$,1)	A(T):c^2 = a2+b2 B(F): a=b ‖ b=c ‖ a=c
median	1	1.(3,1,2) 2.(5,3,4,6,7)	A(F):len%2=0 B(T):temp=num[j]
nextday	1	1.(2019-01-32) 2.(2019-01-00) 3.(2019-01-31) 4.(2019-01-29)	A(F):year<1 B(F):month<1||month>12 C(F):month=4||month=6||month=9||month=11 D(T):month! =2
Financial management	3	1.(输入超级管理员账号和密码，登录) 2.(超级管理员登录成功，修改某管理员权限) 3.(超级管理员登录成功，录入支出)	A(T):role==′super′ B(T):flag==′super′
Course management	2	1.(管理员登录成功，添加课程的学分信息) 2.(学生登录成功，输入学分查询课程信息)	A(T):flag==′addclass′ B(T):textBoxclass.Text==″xuefen″

3　结束语

本文提出了一套包含测试用例选择和测试生成的回归测试用例构建方案，通过该方法选择的测试用例集不仅能够完全覆盖程序的改动部分，还保持了较小的数据集，能够一定程度上降低回归测试选择集的冗余度。而生成的新测试用例能够弥补已有测试用例的覆盖不足，增加回归测试的检错能力，提高回归测试的效率。

本文实验的程序集规模与程序逻辑都不太复杂，约束方程的求解较容易，但随着实验程序的规模与复杂性的增加，特别是在有复杂的外部调用或者频繁的数据库交互时，会导致约束方程求解困难，使新测试用例生成难度加大。所以本文方法还需要在约束方程的求解方面进一步完善。

参考文献

[1] HAILPERN B, SANTHANAM P. Software debugging, testing and verification [J]. Ibm Systems Journal, 2002, 41(1):4-12.

[2] SRISURA B, LAWANNA A. False test case selection: Improvement of regression testing approach [C]//2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON). IEEE, 2016: 1-6.

[3] 胡鹏，常朝稳，祝现威. 面向持续集成的回归测试优化方法[J]. 计算机应用研究，2021，38(12)：3709-3714.

[4] 刘惠敏，赵逢禹，刘亚. 多重关联的静态回归测试用例集构建研究[J]. 计算机应用研究，2019，36(1)：119-123.

[5] 周小莉，赵建华. 基于偶然正确性概率的回归测试选择方法[J]. 软件学报，2021，32(7)：2103-2117.

[6] 叶志斌，严波.符号执行研究综述[J]. 计算机科学，2018，(S1)：28-35.

(下转第 153 页)