各种覆盖率方法介绍

STIN-GZH 提供　　作者：三原
下载：WWW.51CMM.COM

1 简介

1.1 代码覆盖率分析

这篇文章给出了一个完整的代码覆盖率分析方面的概念。

代码覆盖率分析是这样一个过程：

- 找出程序经过一系列测试而没有执行的部分代码
- 创建一个附加的测试用例来增加覆盖率
- 决定代码覆盖的定量度量。

代码覆盖率分析的一个有效方面是：

- 识别出没有增加覆盖率的无效的测试用例。

覆盖率分析需要被测试程序的源代码，并且经常需要用一个特殊的命令重新编译它。这篇文章讨论你应当考虑你的测试计划中应该如何增加覆盖率分析的细节问题。覆盖率分析有一定的好处和弱点。你应该选择一个测量方法的范围。你应该建立一个覆盖率要达到的最小百分比，来决定你什么时候停止覆盖率分析。覆盖率分析只是许多测试技术的一种，你不能只是依靠它。

1.2 结构化测试和功能测试（Structural testing&Functional testing）

代码覆盖率分析是一种结构化测试技术(AKA glass box testing and white box testing). 结构化测试是比较被测试程序的行为和源代码的外观目的。和功能测试相比(AKA black-box testing), 功能测试是比较被测试程序的行为和确定的需求。结构化测试检查程序的工作，考虑结构中可能存在的逻辑缺陷。功能测试检查被测试程序的完成需求的能力，不考虑它是怎么工作的。

结构化测试也叫路径测试（path testing）， 因为你选择测试用例来通过程序结构的路径。不要和路径覆盖率度量（path coverage）混淆，下面会介绍。

粗略的看，结构化测试似乎不安全，结构化测试不能发现需求疏忽的错误，但是，需求定义有时并不存在，而且并不完整。这个现象是实际存在的，当产品开发的时间线就要到的时候，当需求定义很少更新，产品自身代替了需求定义的作用的时候。

1.3 假定

一些基本原理的假定如下所列：

- Faults －－－和控制流相关的缺陷，你可以发现这些缺陷通过变更控制流[Beizer1990 p.60]。例如，一个程序写为"if (c)" 比"if (!c)"好。
- 你可以寻找缺陷而不必知道这个缺陷可能引起的后果和所有测试的可靠性。
- 其它的假定包括可完成需求的定义、没有疏忽的缺陷和没有不可以达到的代码等。

2 基本的度量

有许多覆盖率度量存在，这里介绍一些基本的度量的益处和弱点。

2.1 语句覆盖（Statement Coverage ）

这个度量报告每一个可执行语句是否被执行。也称为： 行覆盖（ line coverage ） ，段覆盖（segment coverage ） [Ntafos1988], C1 [Beizer1990 p.75] 和基本块覆盖（basic block coverage)。基本块覆盖当每一个序列的语句是无分支的语句时和语句覆盖相同。这个覆盖度量的主要好处是它可以直接应用在目标码上，不需要对源代码进行处理。执行轮廓就完成了这个度量。这个覆盖度量的主要缺点是对一些控制结构很迟钝。例如，考虑下列的C/C++ 代码：

```
int* p = NULL;
if (condition)
p = &variable;
*p = 123;
```

如果当condition 取假的情况下，语句覆盖率显示这四句都覆盖到了，但是代码执行是失败的。这是一个语句覆盖率的严重的缺陷，IF语句是很普通的一种情况。语句覆盖不能报告循环是否到达它们的终止条件－－－只能显示循环是否被执行了。既然do-while 循环通常要至少执行一次，语句覆盖认为它们和无分支语句是一样的。

语句覆盖率对逻辑运算符反映是迟钝的(|| and &&)。

语句覆盖不能区分连续的switch 语句。

测试用例通常和判定有关而不是和语句有关。你可能不必用10 个单独的测试用例来测试一个有10 个无分支语句的语句，你可能只用一个测试用例就够了；考虑一个IF－else 语句，包含一条语句在then 子句，99 条语句在else 子句，当执行两个可能路径的其中之一时，语句覆盖率得到如下的结果： 1 或99 的覆盖率。基本块覆盖率就忽略了这个问题。

2.2 判定覆盖（Decision Coverage ）

这个度量报告是否BOOL 型的表达式取值true 和false 在控制结构中被测试到了(例如if-statement 和while-statement)。整个的BOOL型的表达式被认为是取值一个true 和false，而不考虑是否内部包含了logical-and 或logical-or 操作符。另外，这个度量包括switch-statement cases, exception handlers, and interrupt handlers的覆盖.也被称为： 分支覆盖（ branch coverage ） ，所有边界覆盖（all-edges coverage） [Roper1994 p.58], 基本路径覆盖（basis path coverage ）[Roper1994 p.48], C2 覆盖[Beizer1990 p.75], 判定到判定路径覆盖（decision-decision-path 或DDP testing [Roper1994 p.39]. ）"Basis path" 测试就是选择路径来达到所有的判定覆盖.这个度量有语句覆盖的简单性，但是没有语句覆盖

的问题。

缺点是这个度量忽略了在BOOL 型表达式内部的BOOL 取值。例如考虑如下的C/C++/Java 代码：

```
if (condition1 && (condition2 || function1()))
statement1;
else
statement2;
```

这个度量可以完全可以不用调用function1. 测试表达是为真时可以取condition1 为true 和condition2 为true，测试表达是为假时可以取condition1 为false。

2.3 条件覆盖（Condition Coverage ）

条件覆盖报告每一个子表达式的结果的true 或false 。logical-and 和logical-or 独立起来。条件覆盖独立的度量每一个子表达式.这个度量和decision coverage 相似，但是对控制流更敏感。但是，完全的条件覆盖并不能保证完全的判定覆盖。例如，考虑下列的C++/Java 代码。

```
bool f(bool e) { return false; }
bool a[2] = { false, false };
if (f(a && b)) ...
if (a[int(a && b)]) ...
if ((a && b) ? false : false) ...
```

所有三个IF 语句不管a 和b 取值是什么，判定覆盖率只能达到50％。但是条件覆盖率却能达到100％。

2.4 多条件覆盖（Multiple Condition Coverage ）

多条件覆盖报告每一个可能的BOOL 型子表达式的组合发生了。相对于条件覆盖，即通过logical-and 和logical-or 把子表达式独立起来相比， 多条件覆盖需要的测试用例是用一个条件的逻辑操作符的真值表来确定的。

对于C, C++和Java 等具有short circuit operators 的语言，多条件覆盖的益处是它需要一个彻底的测试。

缺点是它可能是非常冗长乏味的来决定一个需要的测试用例的最小设置，尤其是对于非常复杂的BOOL型表达式。另一个缺点是，这个度量需要的测试用例对于相似的复杂性的条件却需要非常大的变化。例如，考虑如下的C/C++/Java 条件：

要达到完全的多条件覆盖，第一个需要6个测试用例，

而第二个需要11 个测试用例。

但是两个条件却有相同的操作数和操作符。

对于Visual Basic 和Pascal 等不具有short circuit operators 的语言, 多条件覆盖对于逻辑表达式是非常有效的路径覆盖，具有相同的优缺点。考虑下列的

Visual Basic 代码：

```
If a And b Then
...
```

多条件覆盖需要四个测试用例，a 和b分别取值true 和false.

2.5 分支条件组合覆盖（Condition/Decision Coverage ）

分支条件组合覆盖是条件覆盖（condition coverage）和分支覆盖（decision coverage）的一个混血。它有两者的简单性但是没有两者的缺点。

2.6 修正条件/判定覆盖（Modified Condition/Decision Coverage）

也被称为MC/DC 和MCDC.这个度量需要足够的测试用例来确定每个条件能够影响到包含的判定[Chilenski1994]的结果。这个度量是最早是被波音公司创建被用于航空软件中RCTA/DO-178B。这个度量起初是是设计来对于无short circuit operators 的语言。short circuit logical operators 在C, C++和Java 语言中的作用仅仅是当它们的结果能够影响到被包含的判定的评估条件。

当以下的需求遇到时，需要考虑用MCDC覆盖：

需求1： 每一个程序模块的入口和出口点都要考虑要至少被调用一次，每个程序的判定到所有可能的结果值要至少转换一次。

需求2：程序的判定被分解为通过逻辑操作符(AND, OR, etc.)连接为BOOL 条件。每一个条件对于判定的结果值是独立的，或者说单条件的变化将导致判决的变化。

所以，据以上的定义，以下三组数据是必须的。(单条件的变化将导致判决的变化)其中T1 & T2, 或T1&T3已经满足要求1，但要求满足条件2，就必须存在同时存在T1&T2&T3。

T4：(F，F)，是多余的，因为一个条件改变并不能导致判决的改变。

T1（T，T）与T2（T，F）表明Y独立影响了判决，Y的独立对；

T1（T，T）与T3（F，T）表明X独立影响了判决，X的独立对。

再如：

测试用例1 (T,T,T) 和测试用例5 (F,T,T) ,对于X独立。

测试用例2 (T,T,T) 和测试用例6 (F,T,T) ,对于X独立。

测试用例3 (T,T,T) 和测试用例7 (F,T,T) ,对于X独立。

测试用例2 (T,T,T) 和测试用例4 (F,T,T) ,对于Y独立。

测试用例3 (T,T,T) 和测试用例4 (F,T,T) ,对于Z独立。

结果，测试用例组{1,5,2,4,3} 满足MCDC覆盖对表达式X, Y和Z的要求。

明显，这不是唯一的组合，例如还有 {6,2,4,3}。

2.6.1 覆盖率的计算公式：

或者： number of tests

number of tests + minimal nuber of test required

2.7 路径覆盖（Path Coverage ）

这个度量报告是否函数的每一个可能的分支都被执行了。一个路径就是一个从函数的入口到函数的出口的唯一的系列分支。

也称呼为断言覆盖（predicate coverage）。断言覆盖可以看出可以组合的逻辑条件的路径[Beizer1990 p.98]。

因为循环引入了一个很大的路径数，这个度量只考虑一个有限数量的循环可能性。有很多的变种来处理循环，内部边界路径测试（Boundary-interior path testing）只考虑循环的两个可能性。重复零次和多于零次的重复[Ntafos1988]。对于do-while 循环，两种，零次反复和多于零次反复。

路径覆盖的一个好处是：需要彻底的测试。

但有两个缺点：

一是，路径是以分支的指数级别增加的，例如：一个函数包含10个IF语句，就有1024 个路径要测试。如果加入一个IF语句，路径数就达到2048。

二是，许多路径不可能与执行的数据无关。例如

```
if (success)
statement1;
statement2;
if (success)
statement3;
```

路径覆盖认为以上语句包含四个路径，实际上只有两个是可行的：

success=false 和success=true.

研究者发明了很多种的路径覆盖技术来避免大数量的路径。如，n-length sub-path coverage 报告你是否exercised each path of length n branches.其它变种包括linear code sequence and jump (LCSAJ) coverage 和data flow coverage.

3 其它度量

这里介绍一些其它的基本的很少使用的度量的益处和弱点。

3.1 函数覆盖（Function Coverage ）

这个度量报告是否你调用了每个函数或过程。对于初步的测试来保证至少在所有的软件没有总的不足非常有用。大多数覆盖率工具都支持。

3.2 函数出入口覆盖（Function Exits Coverage）

报告对函数的入口、出口和终止指令.覆盖情况统计。据我所知，TestRT 支持此覆盖。

3.3 调用覆盖（Call Coverage ）

这个度量报告是否你执行每个函数调用。前提是缺陷一般发生在模块的接口处。也称呼为调用对覆盖（call pair coverage)。据我所知，TestRT 支持此覆盖。

3.4 线性代码顺序及跳转覆盖（ Linear Code Sequence andJump (LCSAJ) Coverage ）

这个是路径覆盖（path coverage ）的一个变更。考虑到在源代码中只有子路径可以被容易的替，不需要一个流程图。一个LCSAJ 是一系列源代码线执行的序列。优点是这个度量比判定覆盖测试的更彻底，而且避免了路径覆盖的指数级的难度。缺点是它不能避免不可实行的路径。据我所知，LDRA TestBed 支持此覆盖。

3.4.1 覆盖率的计算公式：

如下图所示：

一个LCSAJ是由以下四个特征的数量决定的。

A Start Point：可以是程序的开始或任何控制流跳转的目标的线。

A Linear Code Sequence：通过可以系列处理的控制流的代码体。可以由几个连续的基本块组成。

An End Point ：

The first line encountered from which a jump is made which has been reached from the start point by the unbroken linear sequence of code.

A Target Point： The point to which the End Points″ control flow jump is made. This will be the Start Point of the next LCSAJ. Therefore, since the start point of the linear code sequence is a line which is the target of another jump, these fragments are also called jump-to-jump paths.

这个例子的计算此LCSAJ覆盖的分母就是11。

3.5 数据流覆盖（Data Flow Coverage ）

这是path coverage的一个变种，只是考虑到从变量的分配到后来的变量的引用的子分支。好处是直接适当的报告程序处理的数据。一个缺点是不包含判定覆盖，另一个缺点是太复杂。很

多的变量，用于计算的、用于判定的、局部的、全局的，如果有指针就更复杂。

3.6 目标代码分支覆盖（Object Code Branch Coverage ）

这个度量报告是否机器语言条件分支通过了分支。

这个度量给出的报告更多的是依赖编译器而不是程序结构。因为目标代码的结构和源程序的结构很不相似。而且，分支破坏了指令通道，编译起有时候避免产生一个分支，而替换为一个系列的无分支指令。

3.7 循环覆盖（Loop Coverage ）

这个度量报告你是否执行了每个循环体零次、只有一次还是多余一次（连续地）。对于do-while循环，循环覆盖报告是否执行了每个循环体只有一次还是多余一次（连续地）。这个度量的有价值的方面是确定是否对于while循环和for循环执行了多于一次，这个信息在其它的覆盖率报告中是没有的。据我所知，GCT和rational TestRT 可以执行这个度量。

3.8 竞争覆盖（Race Coverage）

这个度量报告是否多个线程在同一时间执行了同一块代码。它帮助检测资源的同步失败问题。对于多线程的程序例如操作系统程序非常有用。据我所知，GCT可以执行这个度量。

3.9 比较操作符覆盖（Relational Operator Coverage）

这个度量报告是否对于比较操作符(<, <=, >, >=)是否发生了边界条件。假设边界条件测试用例发现了off-by-one errors ，并且错误使用了比较操作符，例如把< 用作了<=。例如，考虑如下的C/C++ 代码：

```
if (a < b)
statement;
```

比较操作符报告是否发生了条件a==b。如果If a==b 发生了而且程序执行正常，那么。。。。。。据我所知，GCT可以执行这个度量。

3.10 弱变化覆盖（Weak Mutation Coverage）

这个度量和relational operator coverage 相似，但是更普遍[Howden1982]。它报告是否测试用例发生了错误的引用了操作符和操作数。据我所知，GCT可以执行这个度量。

3.11 表覆盖（Table Coverage）

这个度量显示是否一个特殊的数组的每个入口都被引用。这个度量对于一个通过有限状态机来控制的程序非常有用。

4 比较各种覆盖

- Decision coverage 包含statement coverage，

- Condition/decision coverage 包含decision coverage 和condition coverage
- Path coverage 包含decision coverage.
- Predicate coverage 包含path coverage 和multiple condition coverage

Academia 说强的覆盖率度量包含了弱的覆盖率度量。各种覆盖率度量结果不能从数量上比较。

4.1 对Release版本的覆盖目标

每个项目都要选择一个最低的覆盖率目标，对安全标准严格要求的软件应该制定一个高的目标。单元测试的目标应该比系统测试的高，因为一个在低水平代码的failure 可以影响到多个高水平代码的调用者。用statement coverage, decision coverage 或condition/decision coverage ，在发行版本前80 -90 或更多的覆盖率。可能有人觉得制定的目标少于100 的覆盖率不能保证软件质量，但是你耗在覆盖率技术上的时间如果用于其它测试技术，例如代码走读等，将会发现更多的faults，要避免制定的目标少于80 。

4.2 中间版本的覆盖目标

选择一个好的中间的覆盖率目标可以很大的提高测试的生产力。

当你发现最多的failures 用了最少的努力， 那你就有最高的测试生产力。努力怎么测量呢？创建测试用例，把它们加入到你的测试套并运行它们的总时间就是。

5 总结

覆盖分析是一种结构化测试技术，帮助弥补测试套中的缝隙。对缺少具体的、没有更新需求定义项目帮助很多。分支条件组合覆盖(Condition/Decision Coverage)技术我觉得是对于C, C++和Java语言的最好的。设置一个目标100 的覆盖率(对于任一种覆盖率) 都将阻止测试的生产力，在发放版本之前，定一个80 -90 或更多的目标对于语句、分支、条件覆盖率较好。

6 参考

Beizer1990 Beizer, Boris, "Software Testing Techniques", 2nd edition, New York: VanNostrand Reinhold, 1990

Chilenski1994 John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.

RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", RCTA, December 1992, pp.31, 74.

Howden1982 "Weak Mutation Testing and Completeness of Test Sets", IEEE Trans. Software Eng., Vol.SE-8, No.4, July 1982, pp.371-379.

McCabe1976 McCabe, Tom, "A Software Complexity Measure", IEEE Trans. Software Eng., Vol.2, No.6, December 1976, pp.308-320.

Morell1990 Morell, Larry, "A Theory of Fault-Based Testing", IEEE Trans. SoftwareEng., Vol.16, No.8, August 1990, pp.844-857.

Ntafos1988 Ntafos, Simeon,"A Comparison of Some Structural Testing Strategies", IEEE Trans. Software Eng., Vol.14, No.6, June 1988, pp.868-874.

Roper1994 Roper, Marc, "Software Testing", London, McGraw-Hill Book Company,1994

7 术语表

Fault －－－ A bug. A defect。一个臭虫，一个过失。

Error －－－一个人为的错误，结果是一个fault。

Failure －－－fault的一个实时运行时的表现。