第四讲：软件测试覆盖分析

LIAN YU
THE SCHOOL OF SOFTWARE AND MICROELECTRONICS
PEKING UNIVERSITY
NO.24 JINYUAN RD, BEIJING 102600

提　纲

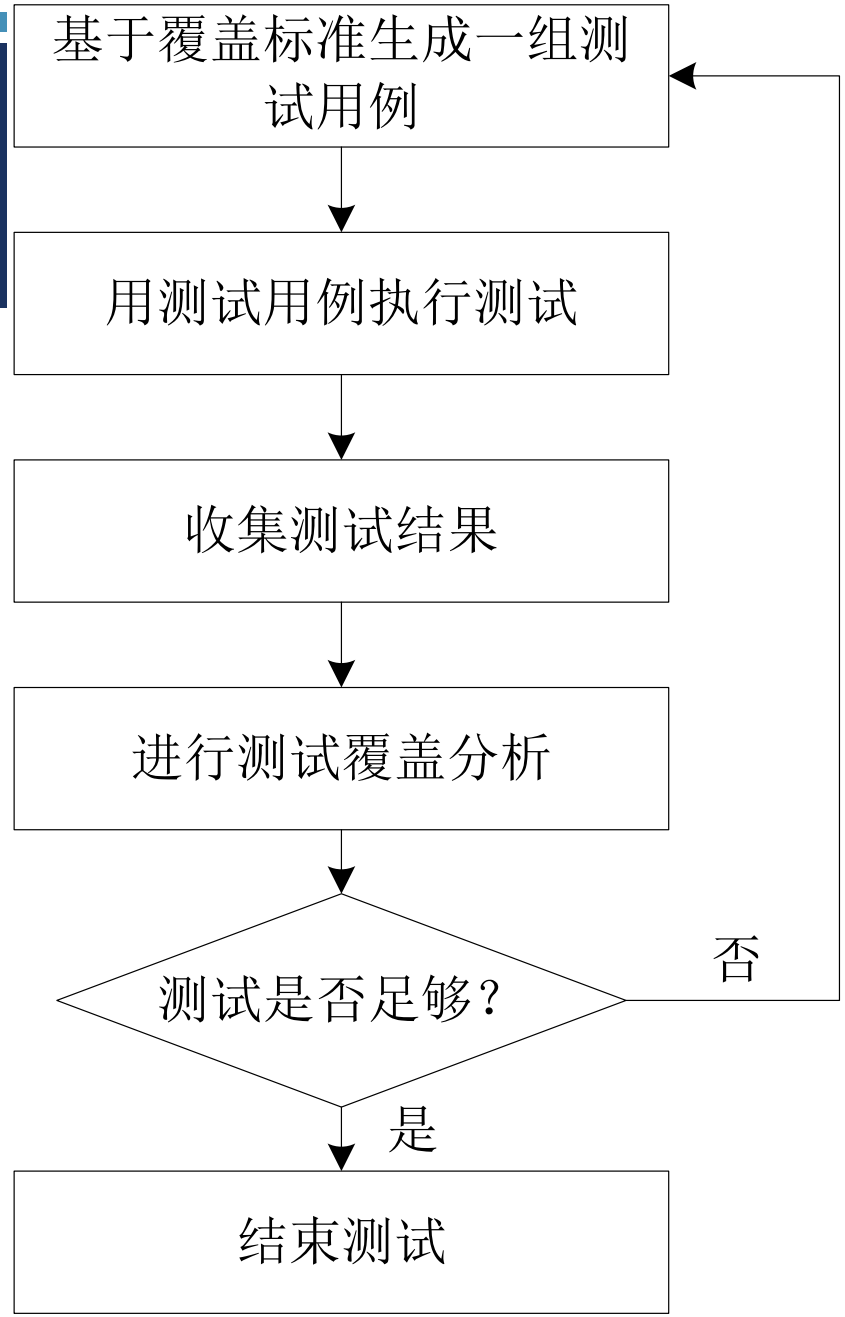
1. 软件测试覆盖分析
2. 代码覆盖分析
3. 控制流覆盖
4. 数据流覆盖
5. 其它覆盖标准
6. 总结

软件测试覆盖分析

- 我们分别学习了白盒测试与黑盒测试技术。我们可能会提出一个问题，就是“测试执行到何时是足够的？”我们需要一种方式来知道测试已经执行的程度。
- 测试覆盖是一种可以凭经验确定软件质量的方法。
  - 每种测试覆盖意味着一种针对特定种类的程序缺陷的测试技术。
- 测试覆盖分析可以在测试计划阶段与测试执行阶段进行。
  - 在测试计划阶段，我们须确定用何种测试覆盖分析及相应的覆盖率。覆盖率通常表示为百分比，但是百分比的意义取决于使用了什么测试覆盖分析。
  - 在测试执行阶段，我们将根据既定的覆盖率来检查是否进行了足够的测试。

基于测试覆盖分析的测试过程

本章将主要地介绍面向白盒测试技术的代码覆盖分析，并简要地介绍几种面向黑盒测试技术的覆盖分析。



提　纲

代码覆盖分析

- 代码覆盖是测试软件的一种量度标准。它描述程序的源代码被测试了的程度。
- 代码覆盖是一种直接观测代码而进行的测试，因而归于白盒测试。
- 代码覆盖技术是最早的系统性软件测试技术中的成员。
- 第一篇参考文献是由Miller and Maloney于1963年发表于“*Communications of the ACM*”杂志上。
- 基于代码覆盖测试工具（方法）的输入是一个程序和一个覆盖标准；输出是一组满足该覆盖标准路径有限集，称作测试组（Suite）。
- 基于代码覆盖测试的两个主要步骤是
  - 识别满足覆盖标准的一组实体，
  - 然后选择一组覆盖该组实体的有限路径。

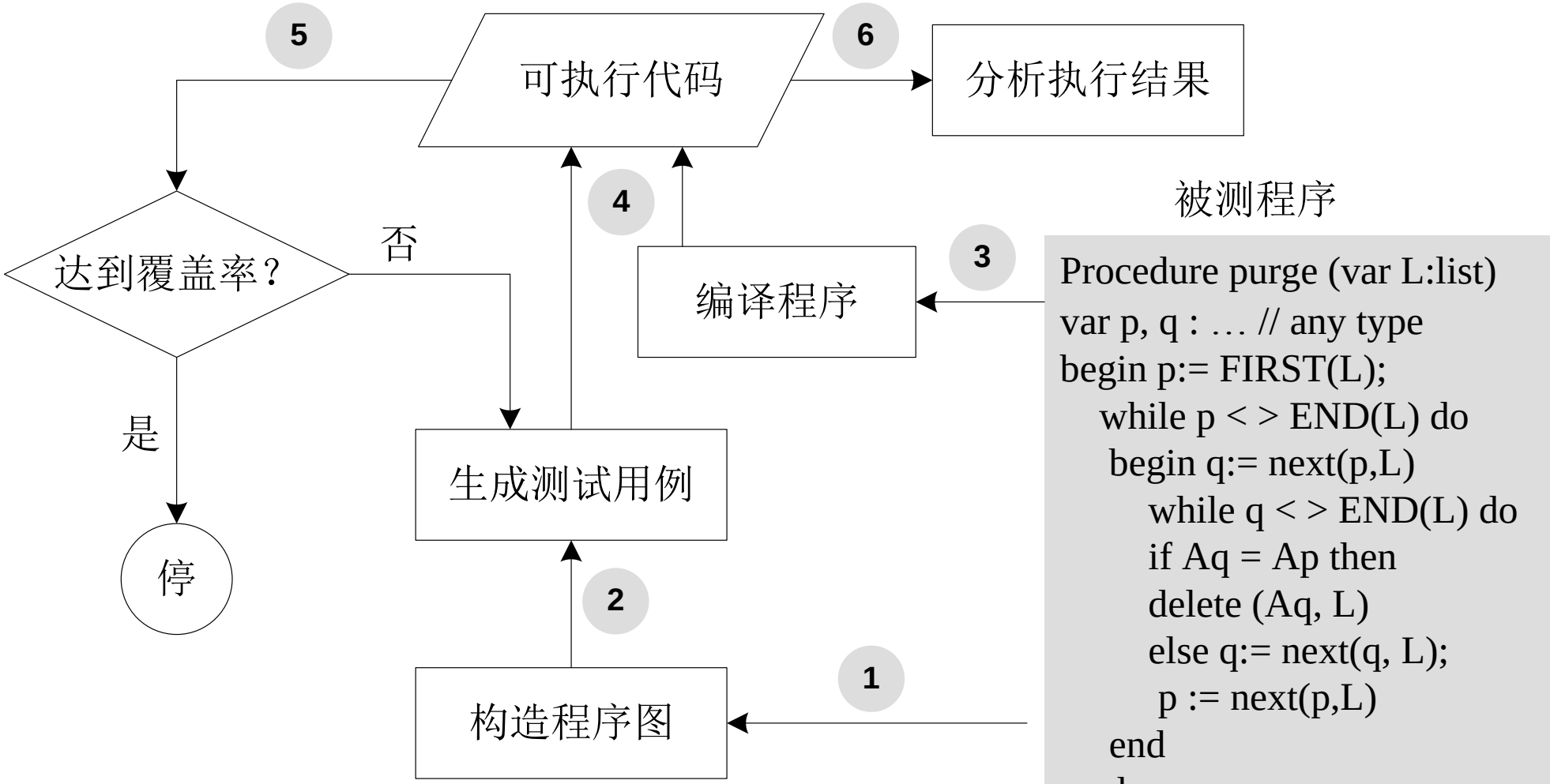
代码覆盖分析(续)

- 有很多种不同的代码覆盖标准及量度代码覆盖方法，介绍两种代码覆盖类型：控制流覆盖与数据流覆盖。
  - 控制流覆盖是选择一组实体以满足覆盖标准：语句覆盖、判定覆盖、条件覆盖、多条件覆盖、条件判定组合覆盖、修正条件/判定覆盖及路径覆盖；然后选择一组覆盖该组实体的有限路径。
  - 数据流覆盖是选择一组满足变量的定义与引用间的某种关联关系实体；然后选择一组覆盖该组实体的有限路径。

代码覆盖分析(续)

- 无论是哪种覆盖类型，它们都遵循如下的测试过程：
  - 由被测程序的源代码，构造程序图。如基本路径法的流图，数据流法的定义使用关联图等。
  - 根据程序图，生成测试用例。如基本路径法中，先算出环形复杂度，再据此找出基本路径集，生成测试用例。
  - 编译被测源程序，生成可执行代码（假设源程序无语法错误）。
  - 生成的可执行代码，用测试用例的输入条件驱动，以执行程序测试。
  - 计算测试结果的实际覆盖率，如果达不到既定覆盖率，则返回第2步，否则结束测试。
  - 对于测试结果，除了进行代码覆盖分析外，还可以进行其他方面的分析，如测试通过率，失败率，可靠性等。

基于覆盖标准的白盒测试流程



被测程序

```
Procedure purge (var L:list)
var p, q : … // any type
begin p:= FIRST(L);
    while p < > END(L) do
     begin q:= next(p,L)
         while q < > END(L) do
         if Aq = Ap then
         delete (Aq, L)
         else q:= next(q, L);
          p := next(p,L)
    end
end
```

提　纲

控制流覆盖

- 控制流覆盖是选择一组实体以满足覆盖标准：
  - 语句覆盖、判定覆盖、条件覆盖、多条件覆盖、条件判定组合覆盖、修正条件/判定覆盖及路径覆盖；
  - 然后,选择一组覆盖该组实体的有限路径。

语句覆盖

- 语句覆盖（Statement Coverage ）度量报告每个可执行语句是否被执行，即每行源代码是否都被执行了并且被测试了。
  - 其含义是选择足够多的测试数据，使被测程序中每条语句至少执行一次。
- 语句覆盖亦称为行覆盖面（line coverage）或段覆盖面（segment coverage）。
- 要达到100%覆盖可能相当难的。
  - 也许有的代码片断被设计用来处理错误条件，如果这种错误不出现，这段代码无法执行也就无法测试；
  - 或很少发生的事件，例如在代码的一个片断接受某一信号，这种情况也给测试此段代码造成困难。

语句覆盖（续）

- 也有可能有的代码永远都不会被执行到，例如以下代码段：

```
if ( x > y && y > z && z > x) {
        die "This should never happen!"; //print out the message
}
```

- 这种类型的覆盖是比较弱的，因为即使通过100%语句覆盖的程序也许仍然有严重的问题，而这些问题通过其他的测试方法可能被发现。看下面的一段代码：

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

语句覆盖（续）

- 这是语句覆盖的一个严重的缺点，因为If语句在程序中普遍存在。
- 即使如此，对于一个任何合理大小软件开发，首先使用语句覆盖是可以发现错误的。

- 语句覆盖的优势是：
  - 它可以直接地被应用于目标代码，并且不需要处理源代码；
- 劣势是：
  - 它对一些控制结构是不敏感的，对程序执行逻辑的覆盖很低，往往发现不了判断中逻辑运算符出现的错误。

语句覆盖（续）

- 语句覆盖计算代码的每一行作为一个可能的语句。这是个好的近似值，但是可能会有各种各样的潜在的曲解和滥用，因为理论上每个记号(例如“If”)都可以占一行。
  - 每一行安置一个记号(这一定不是好的编程样式)，代码行数（LOC）可以是无谓地加大。同样，程序员可以将多条语句放在同一行中(很明显，这也不是一个好的编程实践)，但这样做法使得所需测试用例数量大大地减少，因为一行中的任一语句被执行，都认为此行被执行了。

判定覆盖

- 判定覆盖（Decision Coverage ）度量报告在控制结构中(例如if语句和while语句)是否测试了布尔表达式取值分别为真和假的情况。
- 整个布尔型的表达式取值是true 和false，而不考虑内部是否包含了逻辑与（logical-and）或逻辑或（logical-or）操作符。

- 判定覆盖保证只要程序能跳转，它能跳转到所有可能的目的语句。
  - 其含义是：设计足够的测试用例，使得每个判定至少都获得一次“真”和“假”，或使得每一个取“真”分支和取“假”分支至少经历一次。
- 判定覆盖亦称为分支覆盖、 所有边覆盖。

判定覆盖（续）

- 判定覆盖有语句覆盖的优点简单性，但是没有语句覆盖的问题。
- 缺点是判定覆盖忽略了在布尔表达式内的分支，这是由短路操作(short-circuit)符引起的。
- 考虑以下代码段：

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

- 我们已提到判定覆盖的优点是简单的但没有语句覆盖的问题。它是语句覆盖的超集，比语句句覆盖要强。
- 判定覆盖缺点是它忽略有短路操作符的布尔表达式的分支。当程序中分支的判定是由几个条件组合构成时，它未必能发现每个条件的错误。

条件覆盖

- 条件覆盖（Condition Coverage ）报告每个布尔型**子表达式**的结果是真或假是否都被执行和测试了。
- 子表达式是用辑与（logical-and）运算符和逻辑或（logical-or）运算符分离开。条件覆盖检查每个评估点(例如真/假的判定)是否被执行和测试了。
- 其含义是：构造一组测试用例，使得每一判定语句中每个子逻辑条件的可能值至少满足一次。

条件覆盖（续）

- 考虑以下C++/Java代码段：

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

- 这种情况下判定覆盖率只能达到50%。然而，如果您用b和c所有可能的取值组合来运行这段代码，条件覆盖报告全部覆盖，即条件覆盖率却能达到100%。

条件覆盖（续）

- 条件覆盖与判定覆盖是相似的，但关于控制流有更好的敏感性。
- 但是完全的条件覆盖并不能保证完全的判定覆盖。

条件判定组合覆盖

- 条件判定组合覆盖（Condition/Decision Coverage ）是条件覆盖（condition coverage）和判定覆盖（decision coverage）的一个混合。
- 它有两者的简单性但是没有两者的缺点。条件判定组合覆盖的含义是设计足够的测试用例，使得判定中每个布尔型子表达式条件的所有可能（真/假）至少出现一次，并且每个判定本身的判定结果（真/假）也至少出现一次。

条件判定组合覆盖－示例

ID	布尔型子表达式			判定条件
	a	b	c	a && (b || c)
1	T	T	T	T
2	F	F	F	F

2组测试用例的判定组合覆盖率为100%。

条件判定组合覆盖（续）

- 但是判定组合覆盖也存在一定的缺陷。例如，判定条件a &&(b||c)中的第一个运算符“&&"错写成运算符“||"或第二个运算符“||" "错写成运算符“&&"时，使用表中的2组测试用例无法发现这类错误。

条件判定组合覆盖的缺陷－示例

ID	布尔型子表达式			判定条件	
	a	b	c	a || (b || c)	a && (b && c)
1	T	T	T	T	T
2	F	F	F	F	F

多条件覆盖

- 多条件覆盖（Multiple Condition Coverage ）报告每一种可能的布尔型子表达式的**组合**是否发生了。和条件覆盖一样，子表达式出现时是用辑与（logical-and）运算符和逻辑或（logical-or）运算符分离开。
- 它的含义是：生成足够的测试用例，使得每个判定中的条件的**各种可能组合**都至少出现一次。
- 和条件覆盖一样，多条件覆盖不包括判定覆盖。

多条件覆盖 （续）

- 对于Visual Basic 和Pascal等不含有短路操作符（short circuit operators）的语言，多条件覆盖实际上是对于逻辑表达式的路径覆盖，和路径覆盖具有相同的优缺点。考虑下列的Visual Basic 代码段：

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

- 多条件覆盖需要4个测试用例，a 和b分别取值真（T）和假（F）每一个组合。和路径覆盖相同，每增加一个的逻辑操作符就需要加倍测试用例的数量。

多条件覆盖－示例

ID	布尔型子表达式组合			判定条件
	a	b	c	a && (b||c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

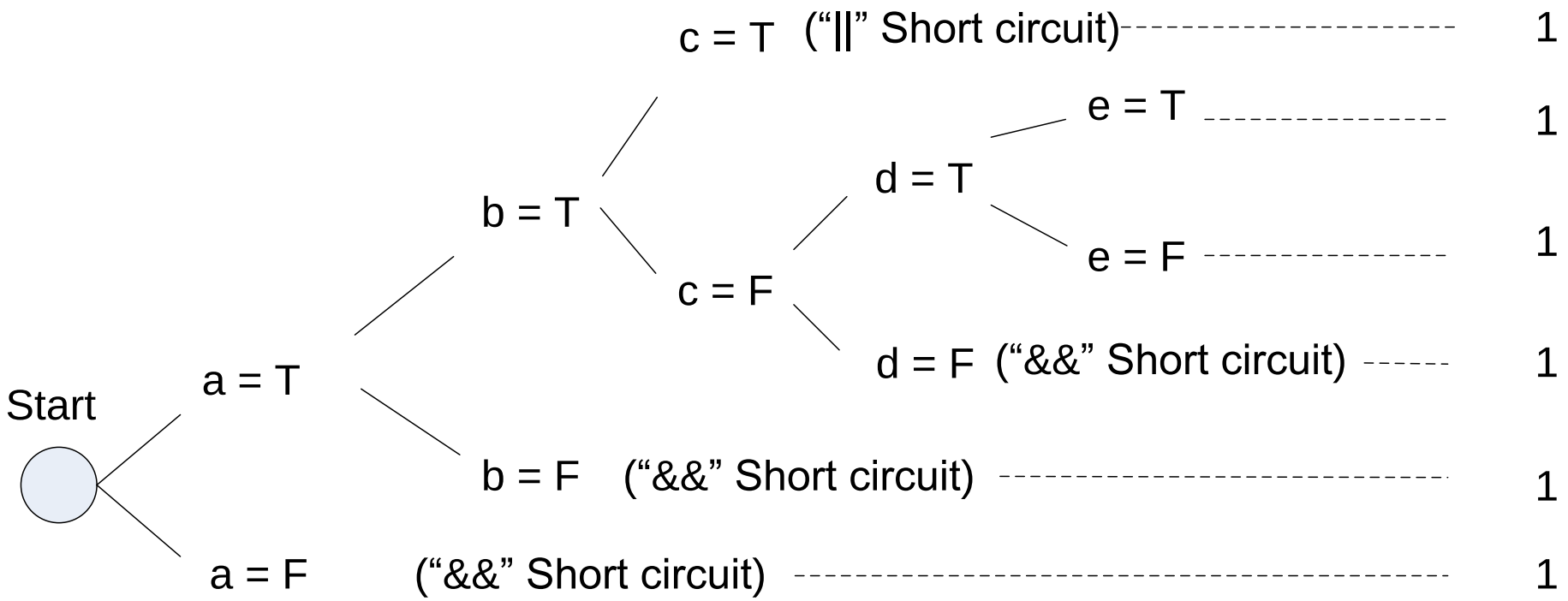
多条件覆盖 （续）

- 对于C, C++和Java等具有短路操作符（short circuit operators）的语言，多条件覆盖的所要求的一些组合测试执行时无法达到。
- 寻求这样一个最小测试用例集可能是非常冗长乏味工作，尤其是对于非常复杂的布尔型表达式。

- 多条件覆盖所需要的测试用例的数目对于具有相似的复杂性的条件却有非常大的不同。
- 例如，考虑如下两个C/C++/Java 的条件

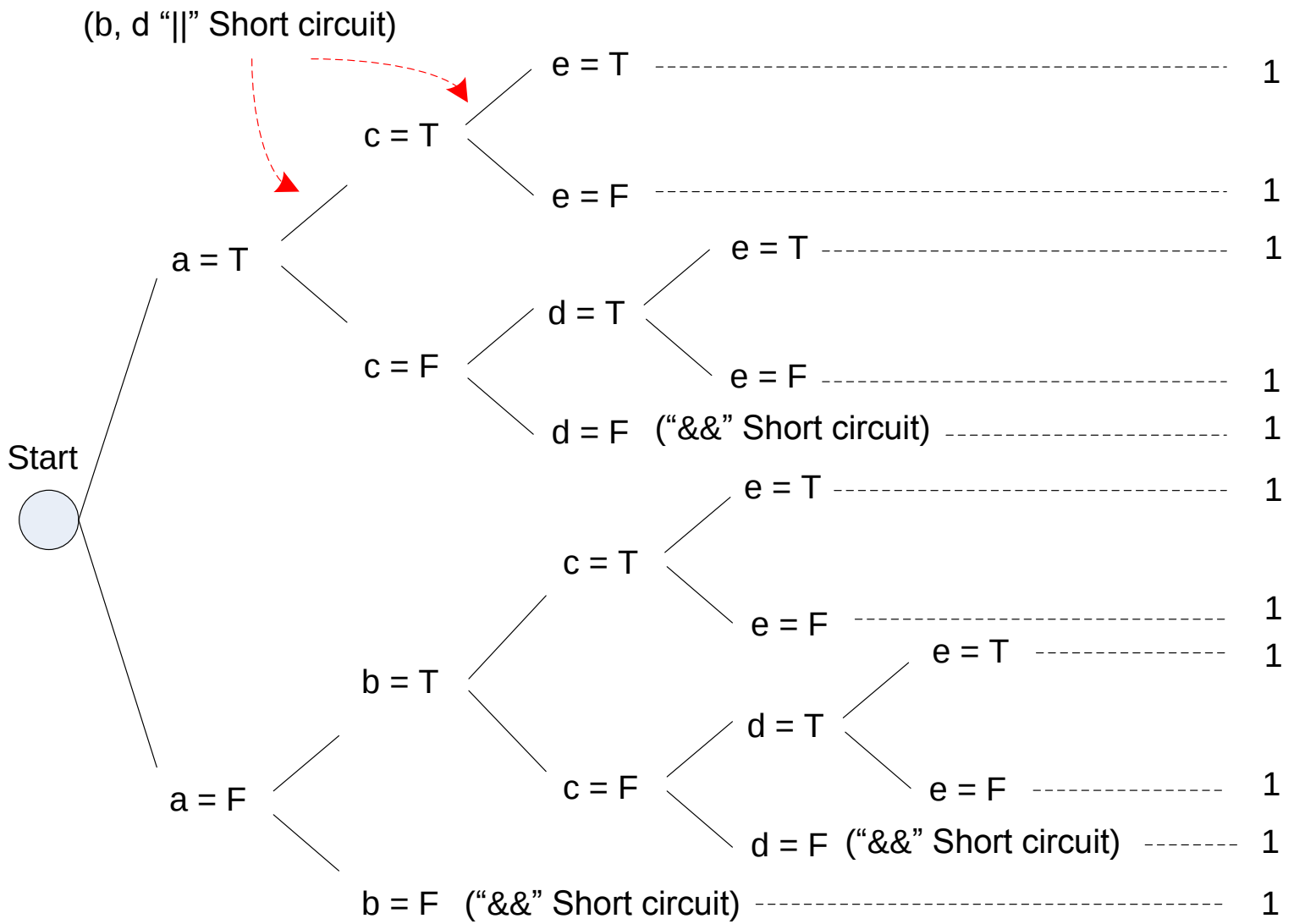
(1) a && b && (c || (d && e))
(2) ((a || b) && (c || d)) && e

多条件覆盖 （续）



多条件覆盖－具有短路操作例（1）条件

多条件覆盖 （续）



多条件覆盖－具有短路操作例（2）条件

多条件覆盖 （续）

- 注意:
  - 上述两个条件有相同的操作数和操作符，由于短路操作符起作用，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。

修正条件/判定覆盖

- 修正条件/判定覆盖（Modified Condition/Decision Coverage）也被称为MC/DC 和MCDC，最早是由波音公司创建。
- 当前在欧美，MC/DC 是“空运系统与设备认证软件考虑事项”（RCTA/DO-178B）中作为必要的测试方法。

修正条件/判定覆盖 （续）

- 它要求生成足够测试用例以满足以下四种条件：
  - 程序里的每个入口和出口都要至少被调用一次，即程序里的要从每一模块入口至少进入一次并至少要从每一模块出口退出一次。
  - 程序中每一个判定的所有可能结果至少取一次。
  - 程序中一个判定的每一条件的所有可能结果至少取一次。
  - 程序中一个判定的每一条件呈现出独立地影响该判定的结果，即只变化此条件而保持其它所有可能的条件不变。

修正条件/判定覆盖（续）

ID	布尔型子表达式组合			判定条件	各条件独立影响		
	a	b	c	a && (b‖c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

MC/DC can guarantee to detect the two types of faults
- ENF (Entire negation fault)
- TNF (Term negation fault) (for Disjunctive Normal Form, DNF)

S = a(b +c) S' = !(a(b +c)) S'' = ! (ab) +ac S'''= ab +c

ID	a	b	c	S
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
6	F	T	F	F

Other types of Boolean Expression Related Faults

- Term omission fault (TOF)
- Conjunctive Operator reference fault (CORF)
- Disjunctive Operator reference fault (DORF)
- Literal insertion fault (LIF)
- Literal negation fault (LNF)
- Literal omission fault (LOF)
- Literal reference fault (LRF)
- Stuck-at fault (SAF)

路径覆盖

- 路径覆盖（Path Coverage）报告是否每个函数的每一条可能的路径都被走过。它检查代码中给定部分每条可能的路径是否都被执行了并且被测试了。
- 一条路径是从函数的入口到出口分支里的一个唯一序列。
- 路径覆盖的一个好处是进行非常彻底的测试。它比判定覆盖强。但有两个缺点：一是路径是以分支的数增加而指数级增加，例如：一个函数包含10个IF语句，就有2^{10}=1024个路径要测试。如果加入一个IF语句，路径数就是原来的2倍2^{11}=2048。二是许多路径由于**数据相关**不可能被执行。
- 考虑以下代码段：

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

提　纲

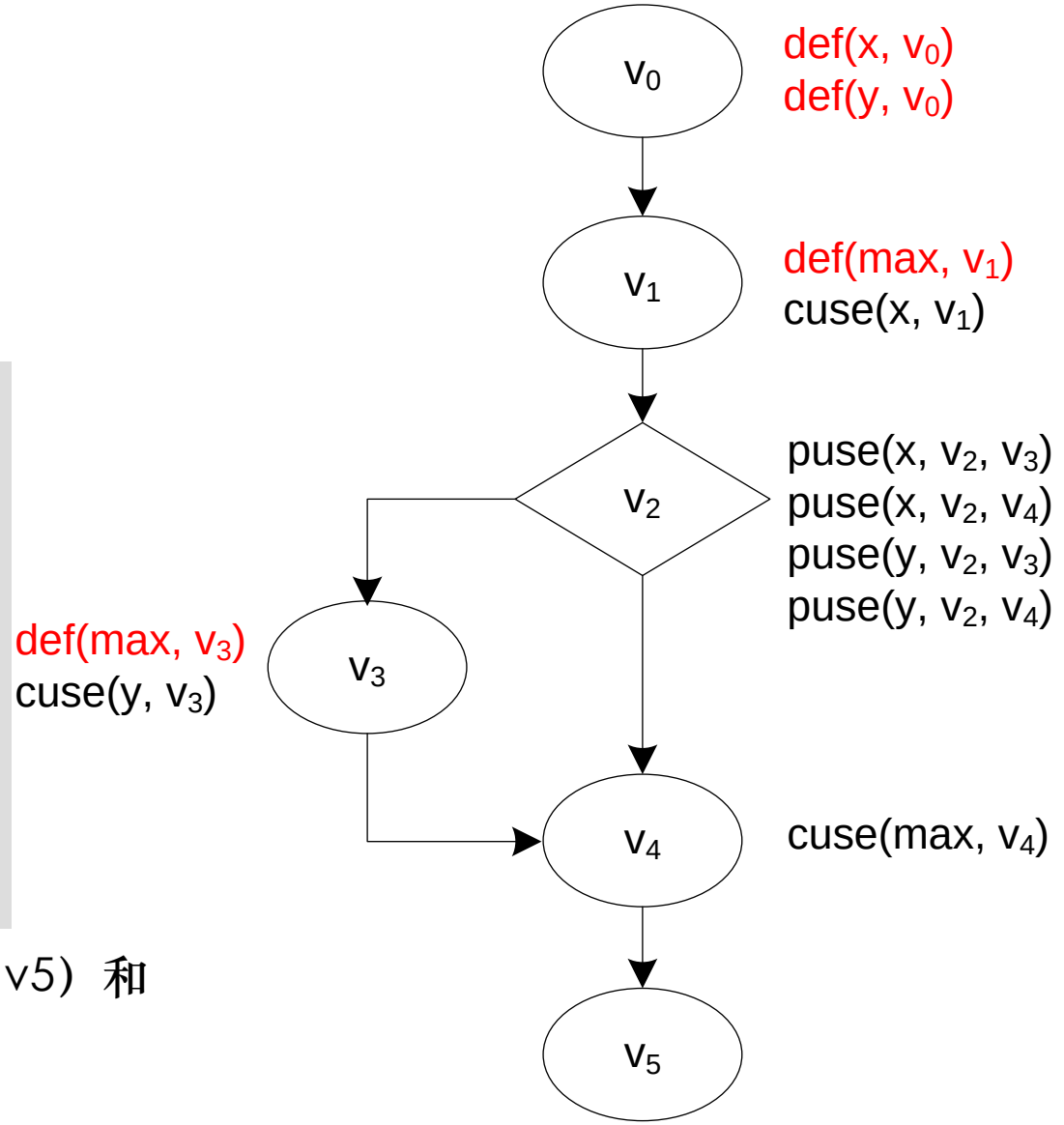
数据流覆盖

- 数据流覆盖是路径覆盖的一个变异。这类路径覆盖的变异仅考虑从变量定义到其后变量的引用之间的子路径。
- 数据流覆盖中的变量“定义”是指变量赋值而不是类型定义。
- 下面介绍四种数据流覆盖选择标准：
  - Rapps和Weyuker的标准
  - Ntafos的标准
  - Ural的标准
  - Laski和Korel的标准

RAPPS和WEYUKER的标准

- Rapps和Weyuker把变量的出现分类为三种类：
  - 定义的(definitional)，用def表示
  - 计算使用（computation use），用c-use或cuse表示
  - 谓语使用（predicate uses），用p-use或puse表示。

```
v0:   int getMaxValue (int x, int y) {
          int max;
v1:       max = x;
v2:       if (y > x)
v3:           max = y;
v4:       return max;
v5:   }
```



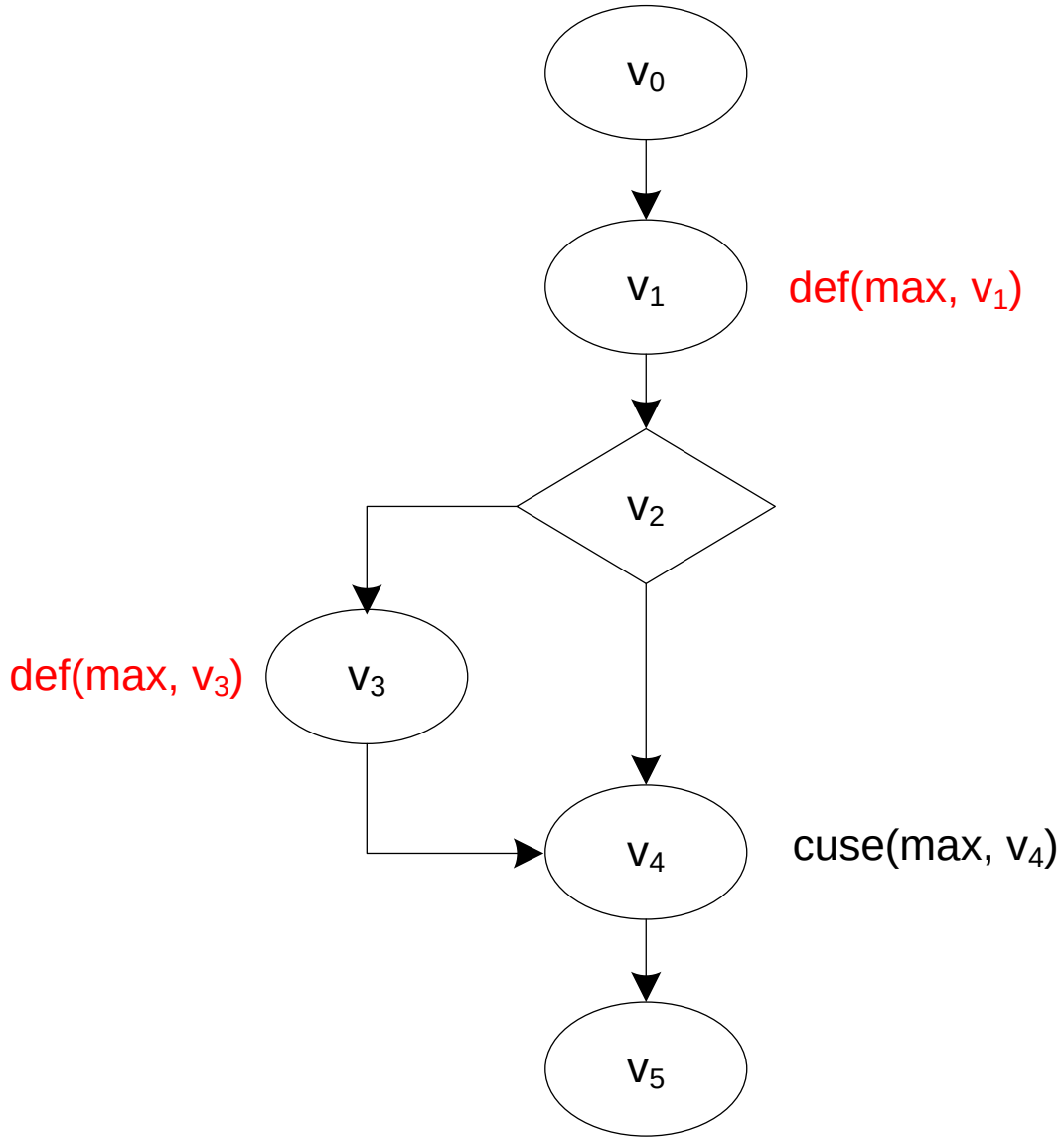
路径P1=（v0，v1，v2，v3，v4，v5）和
路径P2=（v0，v1，v2，v4，v5）

RAPPS和WEYUKER的标准（续）

- 程序图里，(def(max,v1), cuse(max,v4)) 是一个“dcu-对”。
  - （v1，v2，v4）是一个“无重定义”子路径；而（v1，v2，v3，v4）不是，因为在v3语句处变量max被重定义。

```
v0:   int getMaxValue (int x, int y) {
                int max;
v1:             max = x;
v2:             if (y > x)
v3:                     max = y;
v4:             return max;
v5:    } //Correct Program
```

```
v0:   int getMaxValue (int x, int y) {
                int max;
v1:             max = x+1; //Incorrect!!
v2:             if (y > x)
v3:                     max = y;
v4:             return max;
v5:    }//Incorrect Program
```



ALL-USES 和ALL-DEFS

- 一个测试组（Test Suite）被称作是满足“all-uses”覆盖标准：
  - 对于每一个变量x的每一个定义与每一个使用，如果该定义－使用对是du-对，则此du-对被测试组的某一条路径覆盖。
- 一个测试组被称作是满足“all-defs”覆盖标准：
  - 对于每一个变量x的每一个定义与某个使用，如果该定义－使用对是du-对，则此du-对被测试组的某一条路径覆盖。

所有变量每个du-对 / 覆盖标准		all-uses		all-defs
		P_1	P_2	P_1
变量 x	(def(x,v_0), cuse(x,v_1))	♥	♠	♥
	(def(x, v0), puse(x, v2, v3))	♥		♥
	(def(x, v0), puse(x, v2, v4))		♠	
变量 y	(def(y, v0), puse(y, v2, v3))	♥		♥
	(def(y, v0), puse(y, v2, v4))		♠	
	(def(y, v0), cuse(y, v3))	♥		♥
变量 max	(def(max, v1), cuse(max, v4))		♠	
	(def(max, v3), cuse(max, v4))	♥		♥

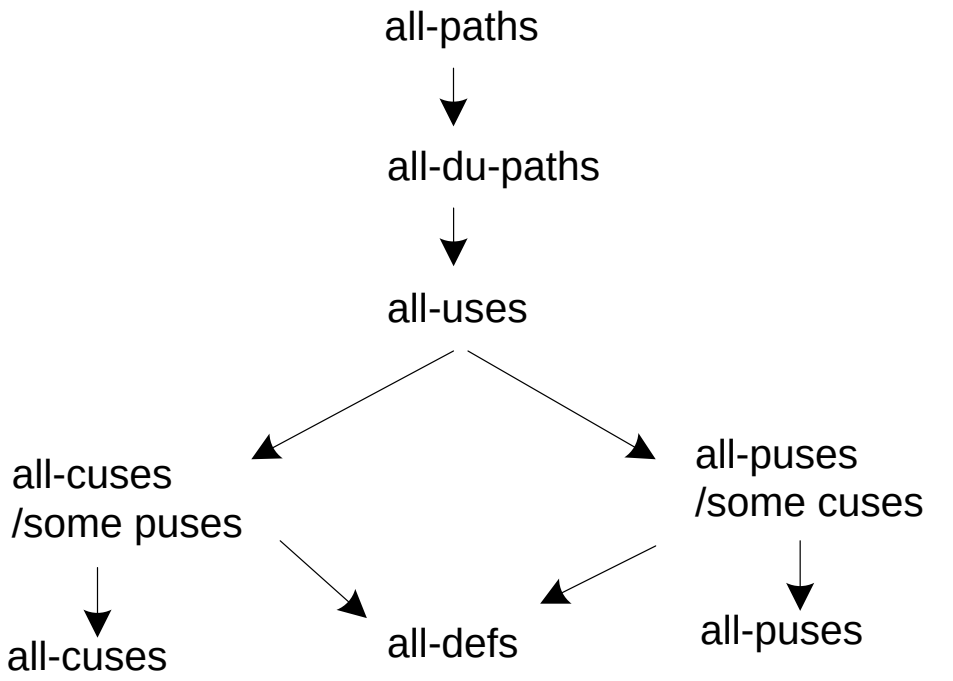
数据流覆盖一族标准

- 下面总结一下数据流覆盖已提出的路径选择的一族标准，包括all-paths, all-edges, all-nodes, all-defs, all-uses, all-puses, all-puses/some-cuses and all-du-paths (du代表定义和使用)标准。
  - all-paths标准要求选择一个路径集合包括贯穿流图的每条路径。它对应于路径覆盖。
  - all-edges（all-branches分支覆盖）和all-nodes（all-statements语句覆盖）标准要求所选择的路径集合包含流图中的每个边和每个结点。
  - all-defs标准要求选择一个路径集合至少要覆盖从每个定义到该定义的某一使用之间的一个定义清纯子路径。
  - all-uses标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个使用之间的一个定义清纯子路径。

数据流覆盖一族标准（续）

- all-puses标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个谓词使用之间的一个定义清纯子路径。
- all-puses/some-cuses标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个谓词使用之间一个定义清纯的子路径。如果定义仅到达计算使用，路径集合必须至少包含从每个定义到一个计算使用之间的一个定义清纯子路径。
- all-cuses/some-puses标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个计算使用之间一个定义清纯的子路径。如果定义仅到达谓词使用，路径集合必须至少包含从每个定义到一个谓词使用之间的一个定义清纯子路径。
- all-du-paths标准比all-uses更进一步。它要求选择一个路径集合要覆盖从每个定义到该定义的每个使用之间的一个定义清纯子路径。并且要求这样的定义清纯子路径是一个简单循环或是无循环的。
- 路径选择的一族标准间的关系可以用层次图表示。处于层次上方的覆盖标准“强于”处于层次下方的覆盖标准。一个测试标准或策略X另外某个标准Y如果由Y产生的测试用例集包含于满足由X产生的测试用例集。

数据流覆盖一族标准（续）

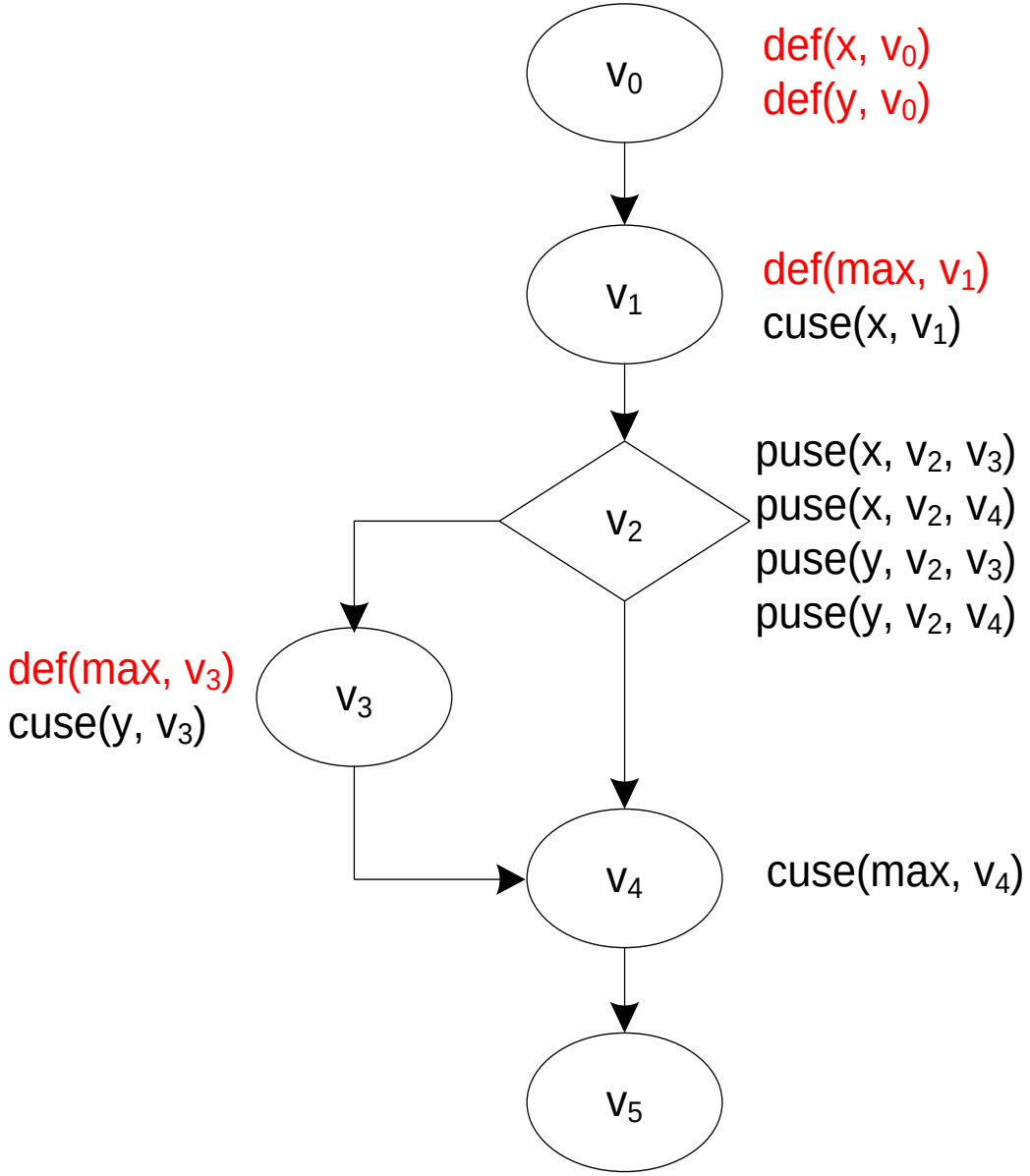


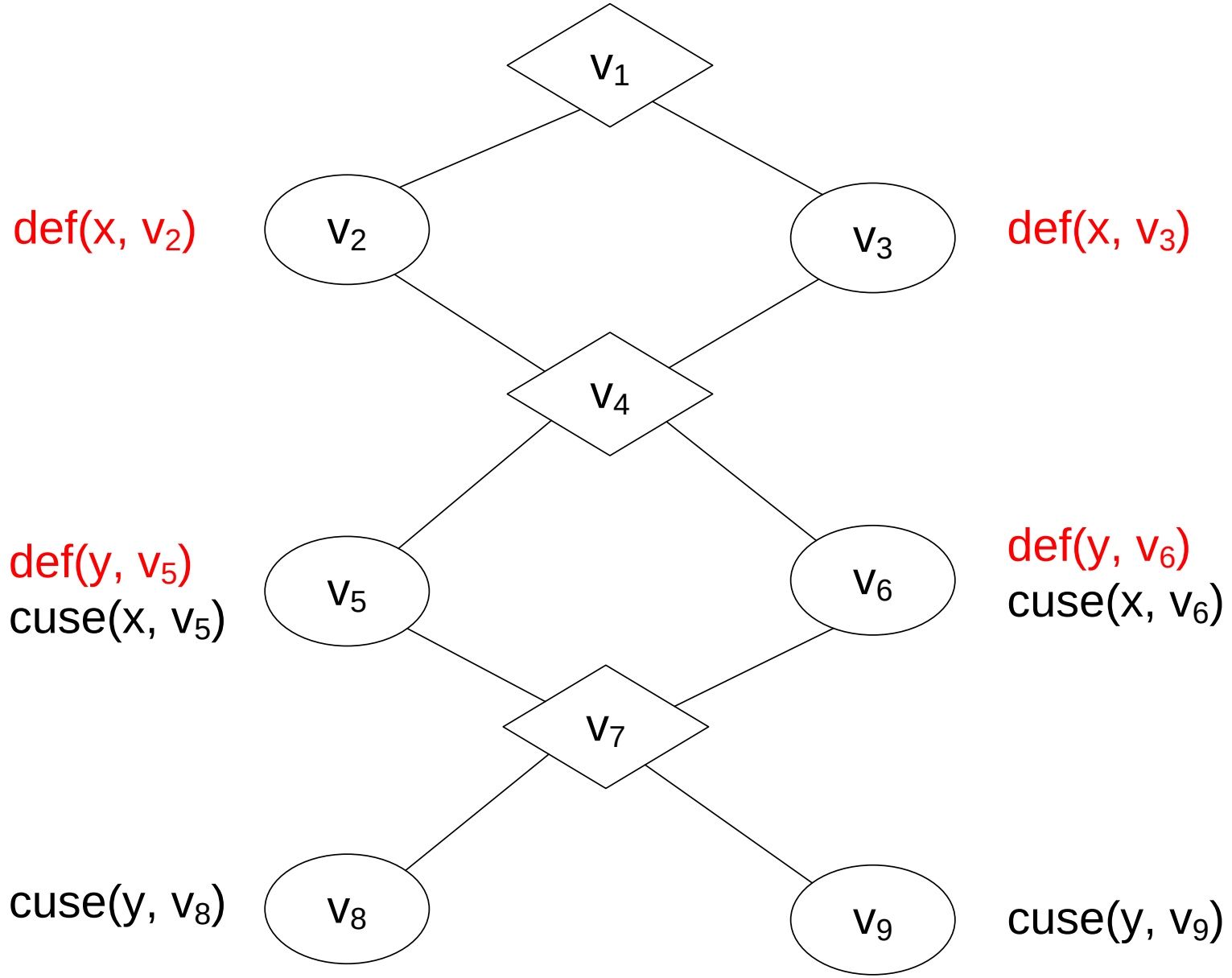
A testing strategy X is stronger than some other strategy Y if all test cases produced under Y are included in those produced under X.

NTAFOS的标准

- 在Ntafos的标准中定义了数据流链（data flow chain, df-chain）：
  - 一个由一组du-对组成的序列 [(d(x1,v1),u(x1,v2)), …(d(xi,vi),u(xi,vi')), (d(xi+1,vi+1),u(xi+1,vi+1')),…, (d(xn+1,vn+1),u(xn+1, vn+1'))]，
  - 其中vi' = vi+1, 即vi'和vi+1是相同节点；除了u(xn+1,vn+1)可能是一个**谓词使用**，其它都是**计算使用**。

- (def(x,v0), cuse(x,v1))是一个du-对，(def(max,v1), cuse(max,v4)) 是另一个du-对。
- 这两个du-对组成一条数据流链（df-链）[(def(x,v0), cuse(x,v1)), (def(max,v1), cuse(max,v4))]。

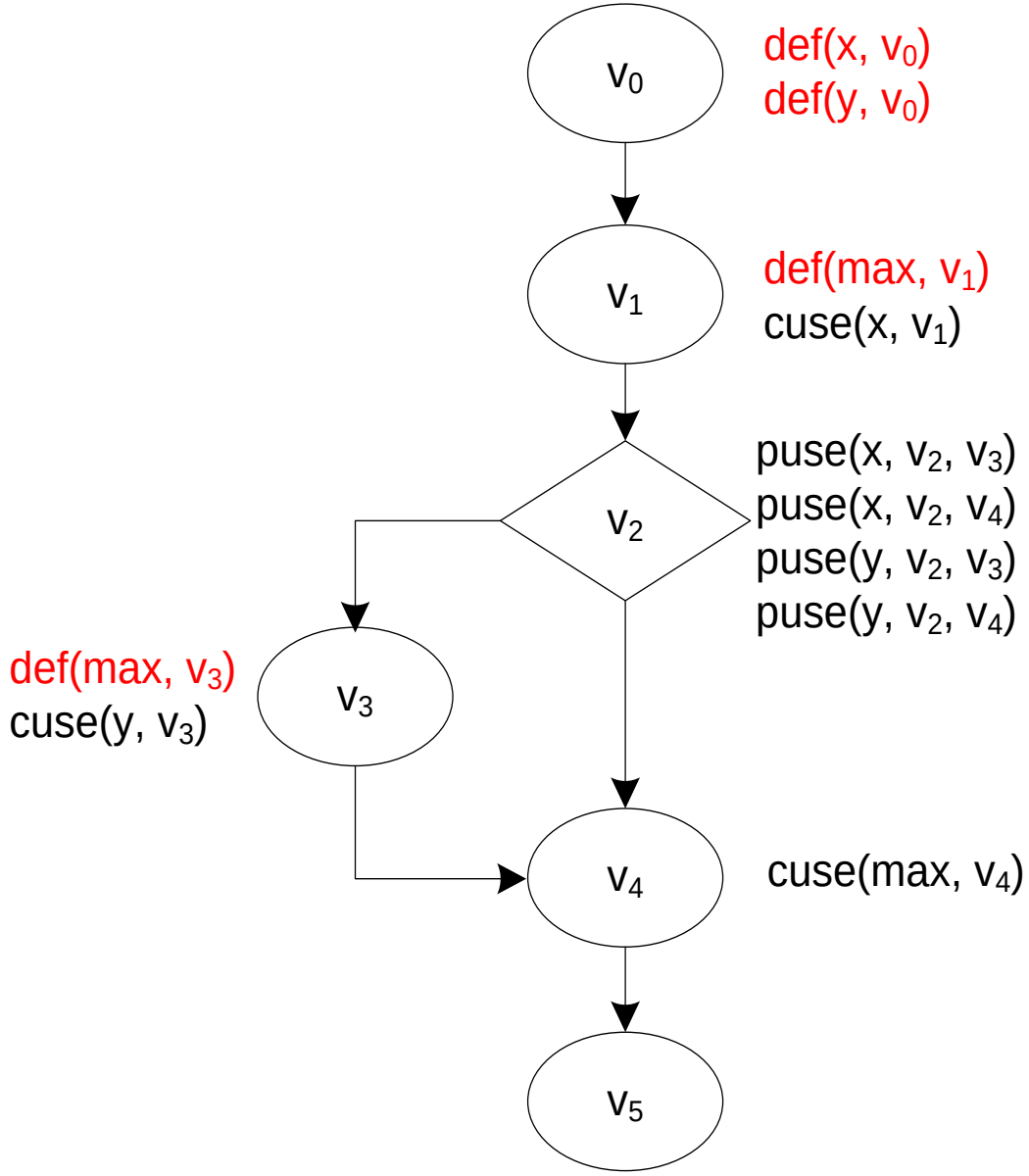



v1
v2
v3
v4
v5
v6
v7
v8
v9
def(x, v2)
def(x, v3)
def(y, v5)
cuse(x, v5)
def(y, v6)
cuse(x, v6)
cuse(y, v8)
cuse(y, v9)

NTAFOS的标准（续）

- 如果一条路径是串接每一个“无重定义”的du-对的df-链某路径，称此路径覆盖该df-链。
- 如果含有k-1个du-对的每一条df-链都被一个测试组的某一条路径覆盖，则该测试组（Test Suite）被称作是满足“必需的k-元组（required k-tuples）”覆盖标准。

图 中的两条df-链[(def(y,v0), cuse(y, v3)) , (def(max,v3), cuse(max,v4))]和[(def(x,v0), cuse(x,v1)), (def(max,v1), cuse(max,v4))]分别被路径P1= (v0，v1，v2，v3，v4，v5) 和P2= (v0，v1，v2，v4，v5) 覆盖，由路径P1和P2组成的测试组满足“必需的3-元组 (required 3-tuples) ”覆盖标准。



NTAFOS的标准（续）

- 注意，Ntafos的标准 “必需的k-元组”可能不包含在Rapps and Weyuker的标准中定义的all-defs标准。

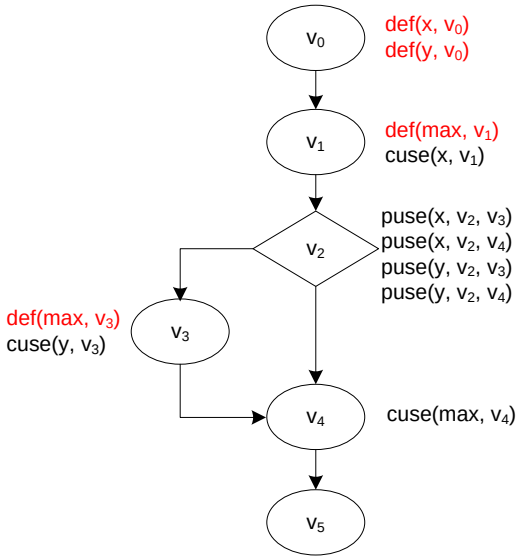
URAL的标准

- 和其它标准不同，基于Ural的覆盖标准的测试输入是一个输入参数或全局变量的一个定义；输出是一个输出参数或全局变量的一个使用或是在返回语句里的一个使用。
- IO-df-链（IO-df-chain）是从输入到输出的一条df-链，它通过描述来自环境的输入是如何影响对环境的输出，来捕获程序的行为。

URAL的标准 （续）

- Ural的覆盖标准明确要求从程序传入参数或定义一个全局变量开始到程序输出使用或返回使用。
- 一个测试组（Test Suite）被称作是满足IO-df-链覆盖标准：如果对于每一个输入和输出，从该输入到该输出之间所有du-链被测试组的某一条路径覆盖。

URAL的标准 （续）

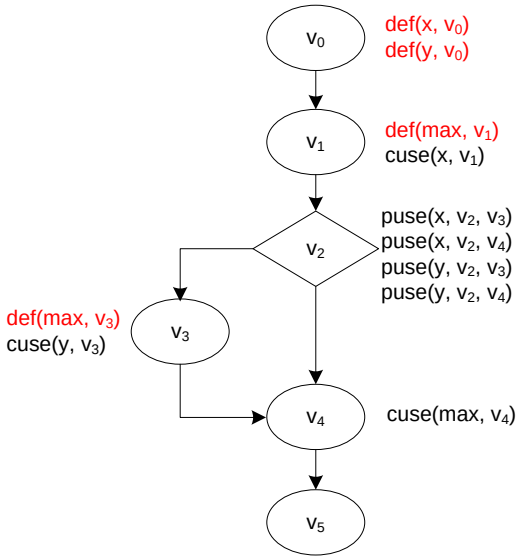


- 输入参数是x和y的定义，输出时是在返回语句里返回max的值。路径P1=（v0，v1，v2，v3，v4，v5）和P2=（v0，v1，v2，v4，v5）覆盖了从每一个输入（x和y）到输出（max）之间所有df-链[(def(y,v0), cuse(y, v3)) , (def(max,v3), cuse(max,v4))]和[(def(x,v0), cuse(x,v1)), (def(max,v1), cuse(max,v4))]。

LASKI和KOREL的标准

- 设cuse(x1,v)，…，cuse(xn,v)是一个在节点v的使用集合。
- 对于节点v的一个有序定义上下文（ordered definition context, ODC）是一个定义序列 [def(x1,v1), …, def(xn, vn)]，使得存在一条路径 v1 p1 … vi pi … vn pn v，满足对于每一个 1≦i≦n, vi pi …v是关于变量xi的一条“无重定义”路径。
- 用同样方法，可以定义对于弧（v, v’） 的一个有序定义上下文。

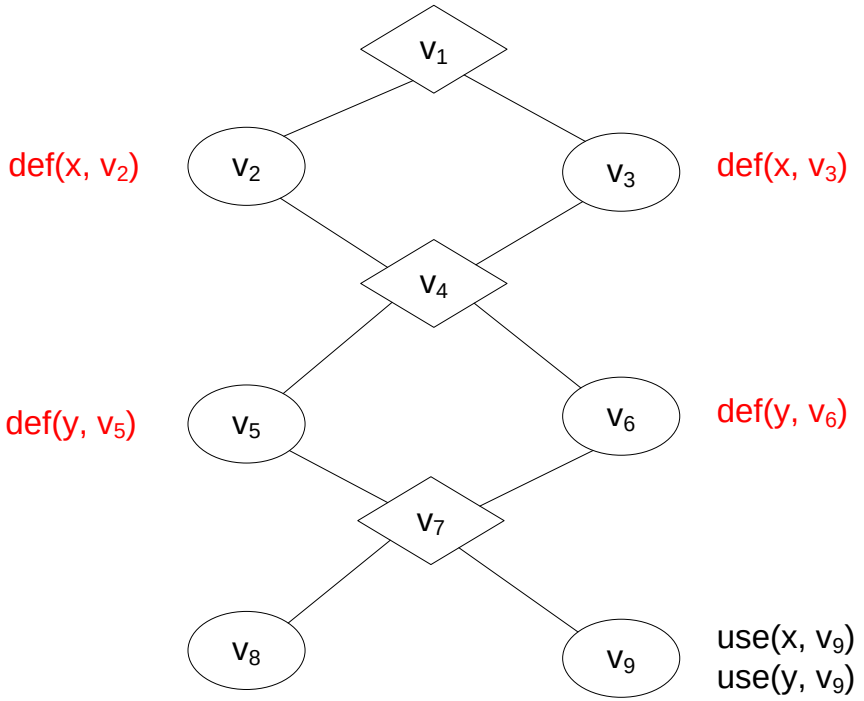
LASKI和KOREL的标准 （续）



- 图中的 [def(x,v0), def(y,v0)] 是弧（v2,v3）的一个有序定义上下文(ODC)。

LASKI和KOREL的标准 （续）

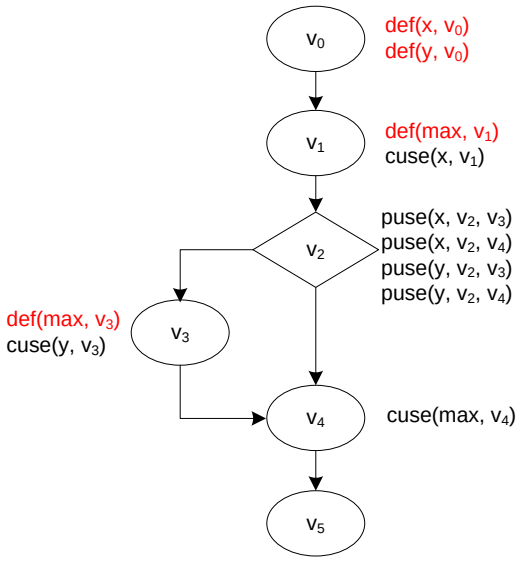
- 为什么使用ODC而不是du-对？如图所示，变量x，y在节点v9的使用受它们各自定义的影响；对于不同定义序列，这种影响可能是不一样的。
- 如[def(x,v2), def(y,v5)]和[def(x,v3), def(y,v6)] 是不同的ODC，对应不同路径，可能产生不同测试效果。



LASKI和KOREL的标准 （续）

- 一个测试组（Test Suite）被称作是满足有序定义上下文（ODC）覆盖标准：如果对于每一个节点和弧，该节点或弧的有序定义上下文被测试组的某一条路径覆盖。

LASKI和KOREL的标准 （续）



- 图 中为{ [def(x,v0)], [def(y,v0)]}是弧（ v2，v3 ）的有序定义上下文；{[def(x,v0)], [def(y,v0)], [def(max,v1)]},是弧（ v2，v4 ）的有序定义上下文；分别被路径P1=（v0，v1，v2，v3，v4，v5）和P2=（v0，v1，v2，v4，v5）覆盖，由路径P1和P2组成的测试组满足有序定义上下文（ODC）覆盖标准.

提　纲

其它覆盖标准

- 我们分别介绍了控制流覆盖与数据流覆盖，它们都属于代码覆盖，也就是基于程序的覆盖分析。
- 然而，覆盖可以延伸到黑盒测试。基于规格说明书，而不是基于代码确定覆盖是可能的。
  - 这当然要依靠所用规格说明的语言。对于较为规范的规格说明语言，是比较容易提出一套覆盖标准。例如，为状态模型提出覆盖标准要比为用英语写的说明书提出覆盖标准更容易。

数据域覆盖

- 数据域覆盖(Data Domain Coverage)是基于等价类划分测试和边界测试。程序里的数据是无穷的。
  - 等价类划分测试能让我们把那些数据分离为几个部分。
  - 这么做，使得测试用例的数量测试运行时切实可行。

数据域覆盖（续）

- 我们如何可以知道我们所关心的用例是否覆盖了所有部分？数据域覆盖的主要步骤如下：
  - 根据程序的逻辑判定把输入范围划分成次级范围。
  - 检查那些子域在当前测试用例集里被覆盖的百分比。

数据域覆盖（续）

- 数据域覆盖的优点是可以容易地和迅速地管理无限数据；
- 其缺点是我们不能肯定我们的划分是否足够完备。

风险覆盖

- 风险覆盖（Risk Coverage）是基于风险分配（risk assignment）。
- 风险是指一个潜在的问题，它是由事件、危险、威胁等情况发生可能程度，及发生后不良后果来衡量的。风险部分与系统的可靠性密切相关。
- 风险覆盖分析给出了有关可能导致危险的问题的一个系统的可靠性；而ABT给出了对于执行的测试所达到的确保水平。

安全覆盖

- 安全对于防御、医疗设备、航空航天、核和空间站应用是至关重要的。在取得测试结果之后，我们如何从结果中可以知道系统的安全程度？安全覆盖（Safety Coverage）能给我们这样的度量。
- 安全覆盖的主要步骤如下：
  - 做FTA (Fault tree analysis)，ETA或FMEA (Failure mode and effects analysis)分析以得到与安全有关的组件。
  - 比较测试用例与那些组件。
- 对于因为安全覆盖很重要的，我们需要高安全性。当检查安全覆盖，我们需要确定它是100%。

状态模型的覆盖标准

- 这种覆盖标准是基于状态模型（State model）、状态机（State machine）或UML的Statechart，
- 有以下几种度量：
  - 状态覆盖：测试用例集中被覆盖的状态数与给定状态模型里所有状态的比率。
  - 事件覆盖：测试用例集中被覆盖的事件数与给定状态模型里所有状态的比率。

状态模型的覆盖标准（续）

- 转换覆盖：测试用例集中被执行的转换数与给定状态模型里所有转换的比率。
- 状态-事件覆盖：测试用例集中被执行的状态-事件数与给定状态模型里所有状态数与所有事件数的乘积的比率。
- 路径覆盖：测试用例集中从开始状态到结束状态之间的路径数与给定状态模型里从开始状态到结束状态之间的所有路径数的比率。
- 高风险的路径、转换和状态覆盖：测试用例集中与风险有关的状态数、路径数以及转换数与给定状态模型里所有被识别的与风险有关的状态数、路径数以及转换数的比率。

状态模型的覆盖标准（续）

- 基于状态模型覆盖标准的优点是，
  - 一旦一个系统的状态被正确地和完备地确定，我们几乎可以确信，测试用例覆盖了整个系统。
  - 即使我们需要增加测试用例，应该增加的测试用例是容易计算出来；
- 缺点是
  - 我们不能肯定的是状态是完备的和正确的。
  - 我们如何能检查状态是完备的和正确的？这项工作是费时的。

覆盖标准有关问题、局限性

- 覆盖标准的选择是重要的。需要执行效益/成本分析。标准的数量是重要的，包括黑盒和白盒覆盖；白盒覆盖里控制流、数据流还有数据域覆盖。
- 在测试上花费的时间是有限的。软件经常频繁地改变，因而我们早一些时候所保证的某些覆盖，但是软件改变了，我们也许需要重开发测试用例或重写脚本。

覆盖标准有关问题、局限性（续）

- 即使选择一系列的测试覆盖，有时很难确定所需的确切的测试标准或很难执行测试覆盖。
  - 例如，假设我们想用等价类划分测试方法，并且我们在每个划分里要有至少3个测试用例，并且在边界值附近有4个测试用例。但是，这套标准是不足够强的，因为一位测试工程师也许会有2个划分，但是一位好测试工程师会有10个划分。
  - 他们两个都满足测试覆盖，但因为第二位测试工程师有更多的划分，她有更多的测试用例!
- 提供覆盖但不能保证软件是高质量的。测试覆盖的评估可能是一项繁琐的任务，需要有采样策略进行检查。

覆盖标准有关问题、局限性（续）

- 如果规格没有改变，同样的黑盒测试用例可以用于回归测试以保证相同的覆盖。不幸地是，如果软件改变了，无论是规格或代码变动而引起的变化，经常需要更新相应的白盒测试用例或是重新开发。改变管理如果软件改变了，经常需要更新相应的覆盖。
- Poston描述了这样常见的情况，即当接近最后期限时，工程师倾向于说有些错误是不重要的或很少发生因而没有必要修改它。这的确是一个危险做法，但是它却是会发生，特别是当最后期限是接近时。当最后期限接近时，变动率增加。我们所观察到是变动率在最后期限达到高峰。

实际应用的建议

- 强调黑盒测试覆盖，例如黑盒路径。
- 现代软件强调迅速开发，例如基于测试的开发，极限编程和敏捷过程，E2E测试。
  - 在这种情况下，建议覆盖测试应该在可交付使用前中期时执行。当软件迅速改变时，维护一些测试覆盖是简直是不可行的。
- 由于每种覆盖代表一种具体测试技术，及对于软件和关注的问题应用这种测试技术进行应该执行的程度范围，在提出建议之前仔细地分析问题是根本的。

实际应用的建议（续）

- 一般来说，要点是要仔细地审查情形与场合。
  - 例如，实时系统可能经常需要控制流覆盖，数据应用需要数据覆盖，可能需要数据流覆盖。数据库事务系统经常要求处理并发事务，因此它也需要控制流覆盖。
  - 特殊系统也需要相应地特殊覆盖，例如时间分析覆盖、并发分析覆盖。
  - 如果系统在维护或开发中，规定回归测试覆盖和影响分析覆盖是重要的。效益/成本分析也是重要的。
  - 我们选择一个覆盖目标达到100%有时是不可能的；但我们应该避免制定目标会低于80%。

提　纲

总 结

- 软件测试覆盖分析
- 控制流覆盖
- 数据流覆盖
- 其它覆盖标准