修正的条件 / 判定覆盖相关的问题

张卫民　申敬松
（北京航天飞行控制中心）

摘　要　通过讨论进行修正的条件 / 判定覆盖(MC/DC)分析时可能会遇到的问题，明确所有布尔表达式、按位运算等程序代码结构均需要满足 MC/DC，并证明如果将一个判定分解为多个等价的判定，对各个部分满足 MC/DC 的测试集，对整个判定并不一定满足 MC/DC。同时指出汇编语言程序同样需要考虑 MC/DC 问题，提出 MC/DC 分析发现问题的方式，以及对问题的处理建议。

关键词　修正的条件 / 判定覆盖　条件　判定　软件测试　MC/DC 分析

1 MC/DC 定义

一个测试用例集满足 MC/DC 要求，是指：

(1) 程序中的每个判定的所有可能结果至少取值一次；

(2) 程序中每个判定中的每个条件的所有可能结果至少取值一次；

(3) 一个判定中的每个条件曾经独立地对判定的结果产生影响；所谓一个条件独立地影响判定的结果，是指固定所有其它可能的条件，仅改变该条件的值，就能使判定的结果改变；

(4) 程序中的每个入口和出口至少执行一次。

条件是指不包含布尔运算符的布尔表达式。

判定是指由条件和零个或多个布尔运算符组成的布尔表达式。一个不含布尔运算符的判定是一个条件。如果一个条件在一个判定中出现多次，则其每次出现都是一个不同的条件。

由于对入口和出口的测试要求在除语句覆盖外的其它覆盖中都有要求，再加上现在的编程规则一般都要求对模块进行单入口、单出口设计，所以对于 MC/DC 分析来说，入口和出口覆盖并不关键，所以本文仅讨论 MC/DC 的前三项要求。

2 条件的定义问题

从 MC/DC 的定义来看，要确定一个条件是否独立地影响判定似乎非常简单。即，如果一个条件能够单独决定判断的结果，那么就可以认为它具有独立的影响。但是，当我们实际分析一个测试用例集是否满足 MC/DC 时，就会发现一些使我们困惑的问题。这其中的一个问题涉及到了条件一词的含义问题。

在判定的定义中，如果没有“如果一个条件在一个判定中出现多次，则其每次出现都是一个不同的条件。”这句话，我们可能都会认为判定(A and B) or (A and C)包含 A、B 和 C 三个条件。然而，根据判定的定义，这个判定却是包含四个条件，即 A、B、A 和 C。对于两次出现的条件 A，改变其中任一个条件就会影响到另一个条件，我们称这两个条件是耦合的。根据上面 MC/DC 的定义，要显示各条件的独立影响，就必须分析在固定第一个条件 A 的值，而使第二个条件 A 的值在 false 和 true 之间变换时，判定结果的变化情况。而这一点在任何实际的环境下都是无法实现的。

根据上面的讨论，在进行 MC/DC 分析时，对于耦合条件的独立影响，不能考虑判定定义中第(3)条最后一句话的限制。

3 判定的范围界定问题

很多人都认为，所谓判定是指用作程序分支条件的布尔表达式，也就是，选择语句的条件、循环语句的循环条件或终止条件。实际上，MC/DC 中的判定除了包含这些条件外，还应包括程序中所有其它的布尔表达式。

例如，假设包含选择语句的一个程序段为：

```
do_any_work;
if ((A and B) or C) then
  do_some_work;
end if
```

在对该程序的测试用例进行 MC/DC 分析时，大家肯定都会对其中的表达式((A and B) or C)进行覆盖分析。但是，如果该程序写成了如下的结构：

```
D:=A and B;
do_any_work;
if (D or C) then
  do_some_work;
end if
```

这时，有的软件测试分析人员会仅对选择语句中的(D or C)进行覆盖分析，也就是认为只要测试用例对(D or C)满足了 MC/DC，也就满足了对该程序段的 MC/DC，而不会再对语句 D:=A and B 右侧的表达式进行覆盖分析。这样做实际上是违背了 MC/DC 要求的真实含义。

我们可以换个角度来分析这个问题。假设仅需对选择语句的条件、循环语句的循环条件或终止条件进行覆盖分析，只要对这些分支条件满足了 MC/DC 就可以认为对整个程序满足了 MC/DC，我们来看一下会出现什么情况。

我们用<expression>表示用作分支条件的布尔表达式。不管该表达式是用于选择结构还是循环结构，也不管该表达式多么复杂，都可以在分支结构之前将该表达式赋值给一个布尔变量，然后在分支结构中使用该布尔变量作为分支条件。如：

```
X:=<expression>;
if (X) then
  ...;
end if
```

如果测试用例使得 X 分别取值 false 和 true，则对 if 语句来说满足了 MC/DC 的要求。这时，对于 if 语句，MC/DC 等价于分支覆盖。如果这时认为对整个程序段满足 MC/DC，那么对整个程序段来说，MC/DC 等价于分支覆盖。也就是说，对任意的分支结构，我们总能将其改写为使得 MC/DC 与分支覆盖等价的形式。这显然违背了 MC/DC 的本意。

所以说，MC/DC 所针对的不仅仅是作为分支结构条件的布尔表达式，而是应该包括程序中的所有布尔表达式。

4 判定的分解和合并问题

有时为了使逻辑概念或表示形式更加清晰，可能需要将一个表达式分解为多个等价的表达式，或者反过来，将多个表达式合成为一个等价的表达式。例如，考虑下面的三个语句：

```
E:= B and C;
A:= E or D;
A:= (B and C) or D;
```

这三个语句都包含有一个判定，且前两个语句在逻辑上等价于第三个语句。这里我们需要讨论的一个问题是，如果一个测试集对前两个判定满足 MC/DC，它对第三个判定是否也满足 MC/DC。在本例中，B、C 和 D 取值为(TTF)、(FTT)和(TFF)的三个测试用例组成的测试集对前两个语句满足 MC/DC，但是对于第三个语句却不满足 MC/DC。这是因为，对于有三个条件的判定，要满足 MC/DC 至少需要四个测试用例。所以说，如果将一个复杂的判定分解为等价的多个简单的判定，对各个简单判定满足 MC/DC 的测试集，对于原有整体的复杂判定不一定满足 MC/DC。

反过来，如果一个测试集对整体满足 MC/DC，则它的各个部分也满足 MC/DC。这里仅以上例说明这一结论。如果一个测试集对第三个语句满足 MC/DC，我们来分析该测试集对第一个语句的覆盖情况。根据对第三个语句的 MC/DC，首先可知条件 B 和 C 的各种可能取值都取到了，且由于 B 和 C 都分别独立影响到了 A，则它们也必然独立影响到了(B and C)，且使得(B and C)的取值发生了变化，否则，它们无法影响到 A。也就是说，B 和 C 的各种可能结果至少取值了一次，E 的各种可能结果至少取值了一次，且 B 和 C 分别独立影响了 E，所以该测试集对第一个语句也满足 MC/DC。同样分析可知，该测试集对第二个语句也满足 MC/DC。

5 用算术运算代替逻辑运算的问题

从 MC/DC 的定义可以看出，MC/DC 关注的对象是布尔表达式。一些富有创造性的程序员可能已经想到了采用灵活的编程技巧来避免 MC/DC 测试的

方法。如果我们将布尔条件看作是取值 0 和 1 的整数，进而用算术代数代替布尔代数，由于对代数表达式可以用少得多的用例进行测试，这样就避免了对测试要求很高的 MC/DC 要求。

例如，有一个包含布尔表达式的程序段：

```
A:= (B and C) or D;
if (A) then
Do_somework( );
end if
```

我们可以将其改写为等价的程序段：

```
A:= (B * C) + D;
if (not(A=0)) then
Do_somework( );
end if
```

从形式上看，满足第二个程序段的 MC/DC 要简单得多，仅需 A 取值 0 和 1 两个测试用例即可。而且这样还能骗过多数的测试覆盖分析工具，轻而易举地达到 MC/DC 要求。

然而，这样做违背了对高关键等级软件要求满足 MC/DC 的本意，这对于发现编码中存在的问题没有任何益处。而且这种构造方法使得代码的含义变得难以理解，使得日后的维护工作变得更加困难。因此，我们要尽力杜绝这种看似聪明的编程方法。如果在 MC/DC 分析时发现有类似的代码，同样需要分析 B、C 和 D 是否独立地影响了 A 的结果。

6 汇编语言程序是否需要 MC/DC 的问题

因为在汇编语言代码中不存在布尔表达式，各个分支条件均为单一条件，所以有不少人认为，对汇编语言代码不存在 MC/DC 的问题，或者说，对汇编程序来说，MC/DC 等价于分支覆盖。其实，这种看法也是错误的。汇编程序同样存在 MC/DC 问题。

例如，我们需要一段如下的处理过程：

```
A:= (B and C) or D;
if (A) then
...;
end if
```

由于汇编语言的限制，我们无法直接在程序中书写上面的复杂逻辑表达式，每个逻辑运算都需要用一条独立的汇编指令表示。上述处理过程的汇编语言编码为：

```
START: GET  B;
AND C;
OR  D;
PUT A;
JUMPNZ ENDIF;
...;
ENDIF: ...;
```

这个程序结构与第三节和第五节中经过改写的程序结构是一样的，都是先计算出布尔表达式的值，然后将该值作为分支的条件。其差别仅在于，在汇编语言中布尔表达式是用多条指令实现，而在高级语言中布尔表达式是用一个语句实现。为了发现计算布尔表达式的程序代码中可能存在的编码错误，对该汇编程序进行测试时同样需要 B、C 和 D 独立地影响 A。

也就是说，我们不要错误地认为判定条件就是一个 A。在该例中，条件应该是 B、C 和 D，一个测试集要满足 MC/DC，必须使条件 B、C 和 D 独立地影响 A。

所以说，汇编语言程序的 MC/DC 并不等价于分支覆盖，汇编程序与高级语言程序一样，都存在 MC/DC 问题。对汇编语言实现的高关键等级软件，同样需要进行 MC/DC 分析，同样需要满足一定的 MC/DC 覆盖率。

7 位运算是否考虑 MC/DC 的问题

在一些实时嵌入式应用中，由于受到存储和传输等限制，以及为了支持硬件接口操作，经常需要进行位操作。在这些系统中，布尔表达式可以用一个字或字节中的不同的位来表示。对于位运算，要满足 MC/DC，必须对表示条件的每一位进行测试。也就是说，如果一个字的 16 位分别表示不同的条件，那么就必须将其当作 16 个条件来处理。

例如，假设 x 和 y 是两个压缩的 16 位字，运算“&”表示“按位与”。对于代码：z = x & y，下表所示的测试集对每一位的“与”都进行了 TT、FT 和 TF 测试。所以，该测试集满足 MC/DC。注意，这里将 z 看作是一个整数值，而不是一个布尔量。

随着位运算语句变得复杂，说明测试集满足 MC/DC 也会变得更加复杂。例如，变量 status 的低两位表示

表 1 两个字"按位与"的测试集

测试用例编号	1	2	3
输入 x	2#1111111111111111#	2#0000000000000000#	2#1111111111111111#
输入 y	2#0000000000000000#	2#1111111111111111#	2#1111111111111111#
输入 z	2#0000000000000000#	2#0000000000000000#	2#1111111111111111#

布尔条件，位 0 表示 power1_on，位 1 表示 power2_on。当 power1_on 或 power2_on 取 true 值时，布尔条件 power_on 取 true 值。该运算的 C 语言代码可写为：

power_on = (status & (3))! = 0;

由于在这个语句中按位与运算还伴随着一个比较运算，作为整数的按位与运算的结果是不可观测的。我们要每次改变 status 的一位，来观察输出状态数否变化。所以，该例满足 MC/DC 的测试集如表 2 所示。

需要注意的是，在上面第 1 个例子中，与运算发生在两个字的对应位之间，因此仅需要三个测试用例。而对于第 2 个例子，尽管形式上也是按位与，而在本质上是一个字不同位之间相或，所以如果有 n 位参加运算，则需要 n+1 个测试用例。

有时在一个字中既包含数据又包含状态信息，在使用数据之前通常需要将状态信息去掉。比如，一个字的高四位是状态位，低 12 位是数据位，则可以用一个屏蔽字 0FFF 与输入字按位与来去掉高四位：

output_word = input_word & 0FFF;

要测试这一功能就需要显示出高四位中的每一位均被置为了 0，而低 12 位保持不变。由于屏蔽字为固定值，无法改变，所以在按位与运算满足 MC/DC 所需的三个测试用例中，有一个是无法实现的，只需要两个测试用例即可。

有些人认为，位运算只是算术运算，因此，对于位运算来说不需要考虑 MC/DC。如果真是这样做的话，我们可以将所有的布尔变量用位来表示，这样就可以简单地避免了 MC/DC 的要求。这显然违背了 MC/DC 目标的本意。

8 测试用例设计依据问题

为了使测试更好地达到 MC/DC 的要求，很多程序员在设计测试用例时很自然的一种做法是，首先分析代码中的判定和条件，然后根据对代码的分析来设计测试用例，即确定测试输入和期望输出。但是，这种基于代码的测试存在很大的缺陷，很难达到测试的目的。

首先，基于代码的测试无法发现部分需求被遗漏这种严重错误。如果某一需求未实现，也就是说程序中根本没有实现这一需求的代码，则根据代码设计的测试用例就不可能发现这一错误。

其次，基于代码的测试可能发现不了很多的编码错误。例如，根据软件需求，应该有如下的功能：

if (A and B and C) then

...; end if

在编码时，错误地将其实现为：

if (A or B or C) then

...; end if

如果根据代码设计测试用例，则我们会设计出表 3 所示三个条件相"或"的最小测试集，根据这个测试集对代码进行测试，四个测试用例都会成功通过，不会发现问题。如果根据需求设计测试用例，就会得到表 4 所示的测试集，根据这个测试集进行测试则很容易发现代码中的编码错误。

所以，我们设计测试用例的依据应该是需求，而不是程序代码结构。当然，这里所指的需求可以是高层需求，也可以是低层需求，还可以是导出需求。

表 2 代码 power_on = (status & (3))! = 0 的最小测试集

测试用例编号	1	2	3
status	2#xxxxxxxxxxxxxx00#	2# xxxxxxxxxxxxx 01#	2# xxxxxxxxxxxxx 10#
power_on	false	true	true

表 3 三个条件相"或"的最小测试用例集

测试用例编号	1	2	3	4
输入 A	F	T	F	F
输入 B	F	F	T	F
输入 C	F	F	F	T
输出 A or B or C	F	T	T	T

表 4 三个条件相"与"的最小测试用例集

测试用例编号	1	2	3	4
输入 A	T	F	T	T
输入 B	T	T	F	T
输入 C	T	T	T	F
输出 D	T	F	F	F

9 问题分析

利用 MC/DC 方法进行结构覆盖分析能够以多种方式发现错误或不足。

首先，通过分析有可能发现代码结构未被基于需求的测试充分执行来满足 MC/DC 标准。未执行代码的产生原因可能是基于需求的测试用例或过程存在缺陷、软件需求不充分、死代码或不活动代码。对于不同的产生原因，应采取不同的处理措施。

（1）如果是基于需求的测试用例或过程存在缺陷，应当补充测试用例或修改测试过程来覆盖未被覆盖的部分，对执行基于需求的覆盖分析方法也应进行检查。

（2）如果是软件需求不充分，应该修改需求，并增加测试用例，重新执行测试过程。

（3）如果是存在死代码，应该将死代码删除，并且评估删除的影响以及是否需要进行重新验证。

（4）对于不活动代码，则存在两种情况。一种情况是，在任一配置环境下都不打算执行的代码，比如一些安装信息，在程序安装完成后就不需要再执行，对于这类代码应进行专门分析或测试，确保避免、隔离或消除无意中执行这些代码的所有的途径。还有一种情况是，这些不活动代码只在某些特定的目标计算机环境配置下执行，对于这类代码，应该建立正常执行这些代码所需的运行配置环境，增加测试用例和测试过程，来满足所需的覆盖率目标。

其次，如果在进行覆盖分析时发现测试输出与期望输出不一致，也可以识别出错误。这时发现的错误可能是需求错误，也可能是代码错误。经过分析，如果确定是需求错误，则应该改正需求，补充测试用例，重新执行测试过程；如果是代码错误，则需要根据项目的代码更动过程对代码进行修改，并重新执行测试过程。

还有，进行 MC/DC 分析还可能以其它的方式发现错误。如下面的例子所示。

假设软件需求是要计算表达式 D:=A and (B xor C)，基于需求的测试用例如表 5 所示。不难证明，表 5 的测试集对于要计算的表达式满足 MC/DC 标准。

假设编码人员将需求表达式错误地编写成了 D:=B and (B xor C)。执行表 5 的测试用例，我们会发现实际测试结果与期望结果完全相同。也就是说，尽管

表 5 基于需求的测试用例集

测试用例编号	1	2	3	4
输入 A	F	T	T	T
输入 B	F	F	T	T
输入 C	T	F	F	T
输出 D	F	F	T	F

表 5 的测试集对需求表达式满足 MC/DC，仅通过测试我们还是无法识别这种编码错误。

现在来分析表 5 的测试集是否对程序代码满足 MC/DC。由于 B 取值为 F 的测试用例屏蔽了 (B xor C)的影响，所以，对于 or 运算来说，测试用例 1 和 2 是无效测试用例，只剩下 3 和 4 两个有效测试用例。而我们知道，对于 or 运算，要满足 MC/DC 至少需要三个测试用例，所以表 5 的测试集对于程序代码不满足 MC/DC 标准。

当发现测试集对程序代码不满足 MC/DC 时，就需要补充测试用例。而当 MC/DC 分析人员试图补充测试用例而分析软件需求时，就会发现编码中的错误。对于错误代码，应根据项目的代码更动过程对代码进行修改，并重新执行测试过程。

10 结束语

本文介绍和讨论了几个有关 MC/DC 的容易混淆的问题和进行 MC/DC 分析时会遇到的问题。通过这些讨论可以看出，对软件增加 MC/DC 要求，并不仅仅是对程序分支点增加了覆盖要求，对程序中的很多代码成分都需要满足 MC/DC，软件测试和覆盖分析的工作量大大增加了。因此，需要我们投入更多的时间和精力来从事这些工作。在进行具体项目测试和覆盖分析时，可能还会遇到很多其它问题。对于遇到的问题，应本着遵循 MC/DC 本意的原则，加以认真研究和处理。 ◇

参考文献

[1] Kelly J H, Dan S V, John J C, et al. A Practical Tutorial on Modified Condition / Decision Coverage [R]. NASA/TM-2001-210876.

[2] RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification [S]. RTCA, Inc. Washington, D. C.. December 1992.

[3] RTCA/DO -248B. Final Report for Clarification of DO -178B Software Considerations in Airborne Systems and Equipment Certification [S]. RTCA, Inc. Washington, D. C.. October 12, 2001