accu 2023

MODIFIED CONDITION/DECISION COVERAGE IN GCC

JØRGEN KVALSVIK

Topics

- Code coverage - overview and taxonomy
- Introduction to modified condition/decision coverage (MC/DC)
- Some example programs
- A (thorough) description of the algorithm in gcc (notation warning)

By the end you will ...

- Be familiar with {line, branch, condition} coverage
- Know about different kinds of MC/DC
- Understand masked conditions
- Be able to measure MC/DC with gcc and gcov
- Have seen some cool maths

There will also be some nuggets, words of wisdom, from smart people

Code coverage

Code coverage is a **collection of metrics** for different properties of your **test suite**. Programs are **instrumented** to record **run-time** information. This is sensitive to inputs and results are usually aggregated over **multiple runs**.

Nugget

Any coverage metric should not be a **goal**, but a measurement of how well the **requirement tests** exercise the structure of the program.

Hayhurst (2001): A Practical Tutorial on Modified Condition/ Decision Coverage.

Silly example:

```
bool maybe_record(double a) {
    if (round(a) >= a) {
        update_counter();
        return true;
    } else {
        return false;
    }
}
maybe_record(0.6); // should round up
maybe_record(6.0); // should not round up
```

```
    2:    9:bool maybe_round(double x) {
    2:   10:    if (round(x) >= x) {
    2:   11:        update_counter();
    2:   12:        return true;
    -:   13:    } else {
#####:   14:        return false;
    -:   15:    }
    -:   16:}
    -:   17:
    1:   18:int main() {
    1:   19:    maybe_round(0.6);
    1:   20:    maybe_round(6.0);
    1:   21:}
```

Oops, else-block not exercised.

A taxonomy of coverage metrics

Line/Statement	Has every **line** of the source been executed?
Branch/Decision	Has every **control flow** structure been evaluated to *both* true and false?
Condition	Has every **boolean sub-expression** been evaluated to *both* true and false?

A taxonomy of coverage metrics

Line/Statement	Has every **line** of the source been executed?
Branch/Decision	Has every **control flow** structure been evaluated to *both* true and false?
Condition	Has every **boolean sub-expression** been evaluated to *both* true and false?

Even line coverage can require a lot of effort

Line/Statement coverage

Has every **line** of the source been executed?

```
int badadder(int x, int y) {
    int tmp = x;
    tmp = tmp + (y - 5);
    return tmp;
    tmp += 5;  // dead as a do-do
}
```

Obviously cannot achieve 100% line coverage.

Line/Statement coverage

Has every **line** of the source been executed?

```
int fn(T* x) {
    if (precondition1(x))
        return -1;
    if (precondition2(x))
        return -1;
    work(x);
    return 0;
}
```

Branch/Decision coverage

Has every **control flow** structure been evaluated to *both* true and false?

```
if (x) {
    // at least once
    first();
} else {
    // at least once
    second();
}
```

Branch/Decision coverage

Has every **control flow** structure been evaluated to *both* true and false?

```
if (x) {
    // at least once
    first();
} else {
    // at least once
    second();
}
```

How is this different from statement coverage?

```
if (x) {
    first();
}
next();
```

When `x` is `true` this has 100% statement coverage and 50% decision coverage.

```
if (always_true()) {
    first();
}
next();
```

every **control flow** [...]

`for` and `while` are `if`s with fake beards

```
while (cond) {
    f(); g();
}
reset();
loop:
    if (!cond) goto endloop;
    f(); g();
    goto loop;
endloop: ;
```

every **control flow** [...]

`for` and `while` are `if`s with fake beards

```
while (cond) {
    f(); g();
}
reset();
loop:
    if (!cond) goto endloop;
    f(); g();
    goto loop;
endloop: ;
```

```
1:     6:int main() {
2:     7:    while (cond) {
branch  0 taken 50%
branch  1 taken 50% (fallthrough)
1:     8:        f(); g();
-:     9:    }
1:    10:    reset();
2:    11:loop:
2:    12:    if (!cond) goto endloop;
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
1:    13:    f(); g();
1:    14:    goto loop;
1:    15:endloop: ;
-:    16:}
```

Condition coverage

Has every **boolean sub-expression** been evaluated to *both* true and false?

```
if (x && y) {
     both();
}
```

x	y	%
0	0	25
1	0	75
1	1	100

More statement vs decision coverage

```
while (accidently_always_true()) {
    f();
    if (g()) break;
    h();
}
```

More statement vs decision coverage

```
while (accidently_always_true()) {
    f();
    if (g()) break;
    h();
}
```

The **loop** always terminates, but only because of the **break**. Could have 100% statement coverage, but not decision coverage.

More condition coverage

```
if (x && accidently_always_true(y)) {
    both();
} else {
    htob();
}
```

```
fn(0, 0); // 25%
fn(1, 0); // 75%
fn(1, 1); // 100%
```

Condition coverage is clearly insufficient.

```
$ gcov -b program
3:      5:void fn(int x, int y) {
3:      6:    if (x && accidently_always_true(y)) {
branch  0 taken 67% (fallthrough)
branch  1 taken 33%
branch  3 taken 100% (fallthrough)
branch  4 taken 0%
2:      7:        both();
-:      8:    } else {
1:      9:        htob();
-:     10:    }
3:     11:}
-:     12:
1:     13:int main() {
1:     14:    fn(0, 0);
1:     15:    fn(1, 0);
1:     16:    fn(1, 1);
-:     17:}
```

```
struct C {
    C() { ... }
    C(const C&) { ... }
    C(C&&)      { ... }
    C& operator = (const C&) { ... }
    C& operator = (C&&)      { ... }
};
```

```
struct C {
    C() { ... }
    C(const C&) { ... }
    C(C&&)      { ... }
    C& operator = (const C&) { ... }
    C& operator = (C&&)      { ... }
};
```

Does your test suite *actually* call the move constructor?

Nugget

C++ is the only language I know that lets you specify a custom copy function and then do its best to not call it.

Tony van Eerd

gcc

gcc emits **.gcno** files (source annotation) and runs create or update **.gcda** files (counters)

Quick start

- Build with `gcc --coverage`
- Run program, test suite
- Generate report with `gcov <program>` or `gcov <source>`
- Read the manual

General advice

- lcov is very useful
- Results (particularly source mapping) only reliable without optimizations
- Apply common sense and good engineering

Modified condition/decision coverage

- How is it different from condition coverage?
- Why even care?

Why even care?

- DO-178B/C (Level A)
- ISO26262 (ASIL D)

Why even care?

- It is a good metric
- Can detect unintended data dependence
- Can detect classes of bad expressions
- Requires testing more interactions in your program
- Drives robustness

The problem with decision coverage:

```
if ((a && b) || c) {
    //
} else {
    //
}
```

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

Metric not very sensitive to the conditions and their interaction, only need two tests for three parameters.

Modified condition/decision coverage satisfied if:

- every entry and exit point has been invoked
- every basic condition has taken on all possible outcomes
- each basic condition has been shown to independently affect the decision's outcome

```
if ((a && b) || c) {
    //
} else {
    //
}
```

- every entry and exit point has been invoked

Branch coverage

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

```
if ((a && b) || c) {
    //
} else {
    //
}
```

- every basic condition has taken on all possible outcomes

Condition coverage

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

```
if ((a && b) || c) {
    //
} else {
    //
}
```

- each basic condition has been shown to *independently* affect the decision's outcome

Modified condition/decision coverage

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

- Testing all 2^N inputs would not reliably catch more defects
- $N + 1$ test cases is sufficient (for coverage)

The problem with MC/DC

- Only tests implementation, not the spec
- Possible to cheat
- Needs many test cases in maybe uninteresting places
- *Awful* to determine without tooling
- Can be expensive and at odds with fuzzing
- Can lead to *Metric Driven Development* (MDD)

Nugget

This is because MC/DC testing discourages defensive code with unreachable branches, but without defensive code, a fuzzer is more likely to find a path that causes problems.

Nugget

MC/DC testing seems to work well for building code that is robust during normal use, whereas fuzz testing is good for building code that is robust against malicious attack.

https://www.sqlite.org/testing.html

A taxonomy of coverage metrics

Line/Statement	Has every **line** of the source been executed?
Branch/Decision	Has every **control flow** structure been evaluated to *both* true and false?
Condition	Has every **boolean sub-expression** been evaluated to *both* true and false?
Modified Condition/Decision coverage	Has every **control flow** structure been evaluated to *both* true and false **and** every condition been shown to affect the decision outcome **independently**?

Unique-cause MC/DC

Only **one** condition may change between a test vector pair, and the resulting decision must be different for the two test vectors.

Unique-cause MC/DC

Only **one** condition may change between a test vector pair, and the resulting decision must be different for the two test vectors.

```
if ((a && b) || (c && d))
```

a	b	c	d	
0	1	0	1	0
1	1	0	1	1
1	0	0	1	0
1	0	1	1	1
1	0	1	0	0

- Need $N + 1$ *specific* test cases to achieve coverage.
- No coverage set if **strongly coupled conditions**.

Masking MC/DC

Only **one** condition having an **influence** on the outcome may change between a test vector pair.

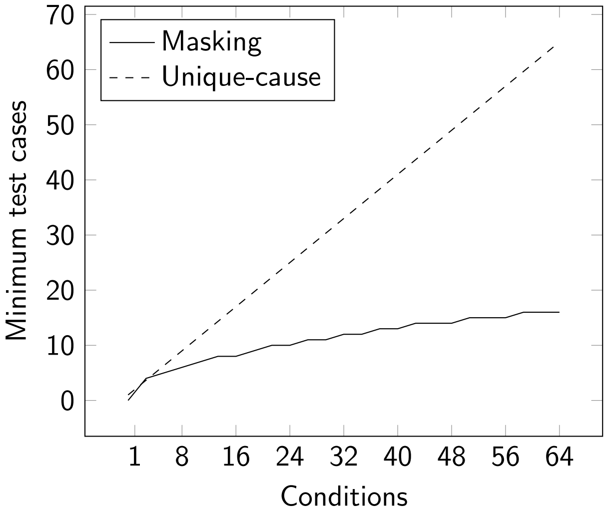
Masking MC/DC

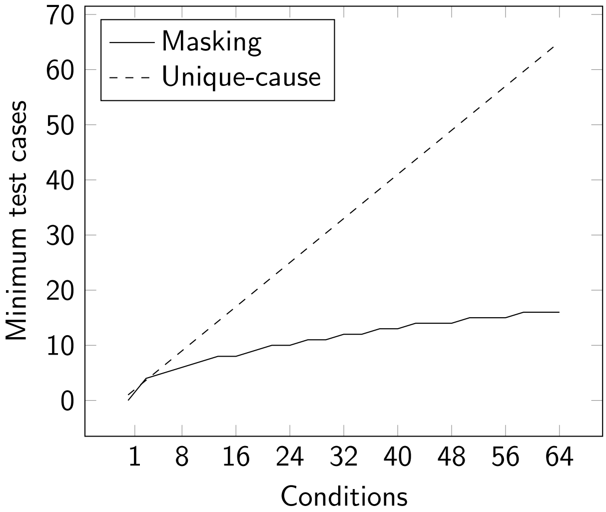
Only **one** condition having an **influence** on the outcome may change between a test vector pair.

```
if ((a && b) || (c && d))
```

a	b	c	d	
0	1	0	1	0
1	1	0	1	1
1	0	1	1	1
1	0	1	0	0

- Need $\lceil 2\sqrt{N} \rceil$ test cases to achieve coverage.
- Multiple test vector sets to choose from, some tests may map better to the requirements.


Masking
Unique-cause
Minimum test cases
0
10
20
30
40
50
60
70
1
8
16
24
32
40
48
56
64
Conditions



$N + 1 \approx \left\lceil 2\sqrt{N} \right\rceil$ for small N

Nugget

Masking MC/DC generally require fewer test cases than unique-cause MC/DC, but is as good at detecting errors.

Chilenski (2001): An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion.

I wrote a patch for gcc

```
$ git log -n 1 --format=short --shortstat
Author: Jørgen Kvalsvik <jorgen.kvalsvik@woven-planet.global>

    Add condition coverage profiling

21 files changed, 2952 insertions(+), 27 deletions(-)
```

```
 gcc/tree-profile.cc |  +978
```

Quick start

```
gcc --coverage -fprofile-conditions
```

Demo

```
$ gcc --coverage -fprofile-conditions
    demo.c -o demo
$ ./demo 0 0 0
$ ./demo 0 0 1
$ ./demo 1 0 0
$ gcov --conditions demo
$ cat demo.c.gcov

    if ((a && b) || c) {
condition outcomes covered 4/6
condition  0 not covered (true)
condition  1 not covered (true)
```

Question

Why is `a = 1` not covered?

Note

This section covers **masking** MC/DC

Requirement

Each basic condition has been shown to **independently** affect the decision's outcome.

Definition

A condition **independently** affects the outcome if changing it while keeping the **other values constant** changes the outcome.

```
if ((a && b) || c) {
    //
} else {
    //
}
```

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

Observation

Changing a does not change the decision

This effect is called *masking*

- `* || true`
- `* && false`

Commutation

Reversing a boolean expression does not change its truth table

$$(P \wedge Q) \equiv (Q \wedge P)$$
$$(P \vee Q) \equiv (Q \vee P)$$

Observation

Masked conditions are *short circuited* in the reversed expression

`(a && b) || c`

a	b	c	
F	*	F	F
F	*	T	T
F	*	F	F
F	*	T	T
T	F	F	F
T	F	T	T
T	T	*	T
T	T	*	T

`c || (b && a)`

c	b	a	
F	F	-	F
F	F	-	F
F	T	F	F
F	T	T	T
T	-	-	T
T	-	-	T
T	-	-	T
T	-	-	T

Short circuiting for the expression * and the reverse -

`(a && b) || c`

a	b	c	
F	*	F	F
F	*	T	T
F	*	F	F
F	*	T	T
T	F	F	F
T	F	T	T
T	T	*	T
T	T	*	T

`c || (b && a)`

c	b	a	
F	F	-	F
T	-	-	T
F	T	F	F
T	-	-	T
F	F	-	F
T	-	-	T
F	T	T	T
T	-	-	T

`(a && b) || c`

a	b	c	
F	*	F	F
-	-	T	T
F	*	F	F
-	-	T	T
T	F	F	F
-	-	T	T
T	T	*	T
T	T	*	T

Note

Row order in `c || (b && a)` changed

```
(a && b) || c
```

	a	b	c	
1	F	*	F	F
2	-	-	T	T
3	F	*	F	F
4	-	-	T	T
5	T	F	F	F
6	-	-	T	T
7	T	T	*	T
8	T	T	*	T

$$a = \{5, 7, 8\}$$
$$\neg a = \{1, 3\}$$
$$b = \{7, 8\}$$
$$\neg b = \{5\}$$
$$c = \{2, 4, 6\}$$
$$\neg c = \{1, 3, 5\}$$

Test sets for cases for masking MC/DC.

`(a && b) || c`

	a	b	c	
1	F	*	F	F
2	-	-	T	T
3	F	*	F	F
4	-	-	T	T
5	T	F	F	F
6	-	-	T	T
7	T	T	*	T
8	T	T	*	T

-
-

$$a = \{5, 7, 8\}$$
$$\neg a = \{1, 3\}$$
$$b = \{7, 8\}$$
$$\neg b = \{5\}$$
$$c = \{2, 4, 6\}$$
$$\neg c = \{1, 3, 5\}$$

Test sets for cases for masking MC/DC.

`(a && b) || c`

	a	b	c	
1	F	*	F	F
2	-	-	T	T
3	F	*	F	F
4	-	-	T	T
5	T	F	F	F
6	-	-	T	T
7	T	T	*	T
8	T	T	*	T

-
-

$$a = \{5, 7, 8\}$$
$$\neg a = \{1, 3\}$$
$$b = \{7, 8\}$$
$$\neg b = \{5\}$$
$$c = \{2, 4, 6\}$$
$$\neg c = \{1, 3, 5\}$$

Test sets for cases for masking MC/DC.

`(a && b) || c`

	a	b	c	
1	F	*	F	F
2	-	-	T	T
3	F	*	F	F
4	-	-	T	T
5	T	F	F	F
6	-	-	T	T
7	T	T	*	T
8	T	T	*	T

$$a = \{5, 7, 8\}$$
$$\neg a = \{1, 3\}$$
$$b = \{7, 8\}$$
$$\neg b = \{5\}$$

- $c = \{2, 4, 6\}$
- $\neg c = \{1, 3, 5\}$

Test sets for cases for masking MC/DC.

`(a && b) || c`

	a	b	c	
1	F	*	F	F
2	-	-	T	T
3	F	*	F	F
4	-	-	T	T
5	T	F	F	F
6	-	-	T	T
7	T	T	*	T
8	T	T	*	T

$$a = \{5, 7, 8\}$$
$$\neg a = \{1, 3\}$$
$$b = \{7, 8\}$$
$$\neg b = \{5\}$$
$$c = \{2, 4, 6\}$$
$$\neg c = \{1, 3, 5\}$$

Test sets for cases for masking MC/DC.

Detecting errors

Specification	`(a && b)  || c`
Implementation	`(a && !c) || c`

Masking table

a	b	c	
F	*	F	F
-	-	T	T
F	*	F	F
-	-	T	T
T	F	F	F
-	-	T	T
T	T	*	T
T	T	*	T

`(a && !c) || c`

a	!c	c	
F	*	F	F
-	-	T	T
F	*	F	F
-	-	T	T
T	T	*	T
-	-	T	T
T	T	*	T
T	T	*	T

Masking table

a	b	c	
F	*	F	F
-	-	T	T
F	*	F	F
-	-	T	T
T	F	F	F
-	-	T	T
T	T	*	T
T	T	*	T

`(a && !c) || c`

a	!c	c	
F	*	F	F
-	-	T	T
F	*	F	F
-	-	T	T
T	T	*	T
-	-	T	T
T	T	*	T
T	T	*	T

Note

Some strong coupled conditions cannot be detected by masking MC/DC

```
(a && b) || (a && c)
```

a	b	a	c	
0	*	0	*	0
0	*	0	*	0
-	0	-	0	0
-	0	1	1	1
0	*	-	0	0
0	*	0	*	0
1	1	*	*	1
1	1	*	*	1

Full unique-cause coverage is not possible (a repeated)

Cheating MC/DC

```
if ((a && b) || c) {
    //
} else {
    //
}
```

```
int ab = a && b;
if (ab || c) {
    //
} else {
    //
}
```

a	b	c	
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

ab	c
F	F
F	T
T	F
T	T

Programs

If computers had blood, we would be considered butchers

gcc does **control flow graph** analysis for coverage

gcc does **control flow graph** analysis for coverage

```
if (a && b && c)
    x = 1;
```

```
if (a)
    if (b)
        if (c)
            x = 1;
```

gcc does **control flow graph** analysis for coverage

```
if (a && b && c)
    x = 1;
```

```
if (a)
    if (b)
        if (c)
            x = 1;
```

```
    #####:    2:    if (a && b && c)
condition outcomes covered 6/6
    #####:    3:        x = 1;

    #####:    2:    if (a)
    #####:    3:        if (b)
    #####:    4:            if (c)
    #####:    5:                x = 1;
condition outcomes covered 6/6
```

Rust #1

```
fn f(a: bool, b: bool, c: bool) -> bool {
    a || (b && c)
}

fn main() {
    f(true,  true, false);
    f(false, true, false);
}
```

Rust #1

```
$ gccrs --coverage -fprofile-conditions prog.rs -o prog \
    -frust-incomplete-and-experimental-compiler-do-not-use
$ ./prog
$ gcov --conditions prog
File 'prog.rs'
Lines executed:100.00% of 5
Condition outcomes covered:50.00% of 6
Creating 'prog.rs.gcov'
```

Rust #1

```
        2:    1:fn f(a: bool, b: bool, c: bool) -> bool {
        2:    2:    a || (b && c)
condition outcomes covered 3/6
condition  1 not covered (true false)
condition  2 not covered (true)
        -:    3:}
        -:    4:
        1:    5:fn main() {
        1:    6:    f(true,  true, false);
        1:    7:    f(false, true, false);
        -:    8:}
```

Rust #1

```
        2:    1:fn f(a: bool, b: bool, c: bool) -> bool {
        2:    2:    a || (b && c)
condition outcomes covered 3/6
condition  1 not covered (true false)
condition  2 not covered (true)
        -:    3:}
        -:    4:
        1:    5:fn main() {
        1:    6:    f(true,  true, false);
        1:    7:    f(false, true, false);
        -:    8:}
```

Summary

Rust #1

```
        2:    1:fn f(a: bool, b: bool, c: bool) -> bool {
        2:    2:    a || (b && c)
condition outcomes covered 3/6
condition  1 not covered (true false)
condition  2 not covered (true)
        -:    3:}
        -:    4:
        1:    5:fn main() {
        1:    6:    f(true,  true, false);
        1:    7:    f(false, true, false);
        -:    8:}
```

Condition index

Rust #1

```
        2:    1:fn f(a: bool, b: bool, c: bool) -> bool {
        2:    2:    a || (b && c)
condition outcomes covered 3/6
condition  1 not covered (true false)
condition  2 not covered (true)
        -:    3:}
        -:    4:
        1:    5:fn main() {
        1:    6:    f(true,  true, false);
        1:    7:    f(false, true, false);
        -:    8:}
```

Quiet if fully covered

Rust #1

```
        2:    1:fn f(a: bool, b: bool, c: bool) -> bool {
        2:    2:    a || (b && c)
condition outcomes covered 3/6
condition  1 not covered (true false)
condition  2 not covered (true)
        -:    3:}
        -:    4:
        1:    5:fn main() {
        1:    6:    f(true,  true, false);
        1:    7:    f(false, true, false);
        -:    8:}
```

Conditions **not** shown to be independent

Rust #2

```
fn loops(init: i32) -> i32 {
    let mut i = init;
    let mut x = 0;
    while true {
        x *= i;
        i += 1;
        if i > 5 { break }
    }
    while i < 20 {
        x -= i;
        i *= 2;
    }
    x
}

fn main() {
    loops(0);
    loops(5);
}
```

Rust #2

```
        2:    1:fn loops(init: i32) -> i32 {
        2:    2:    let mut i = init;
        2:    3:    let mut x = 0;
        5:    4:    while true {
       7*:    5:        x *= i;
condition outcomes covered 1/2
condition  0 not covered (true)
       7*:    6:        i += 1;
condition outcomes covered 1/2
condition  0 not covered (true)
        7:    7:        if i > 5 { break }
condition outcomes covered 2/2
        -:    8:    }
        6:    9:    while i < 20 {
condition outcomes covered 2/2
       4*:   10:        x -= i;
condition outcomes covered 1/2
condition  0 not covered (true)
       4*:   11:        i *= 2;
condition outcomes covered 1/2
condition  0 not covered (true)
        -:   12:    }
        2:   13:    x
        -:   14:}
```

C++ #1

```
class C {
public:
    explicit C(int c) noexcept (true) : v(c) {}
    bool operator < (const C& o) const noexcept (true) {
        return this->v < o.v;
    }

private:
    int v;
};

int main() {
    C one(1), two(2);
    int three = 3, four = 4;
    int x = 0;
    if (one < two && four < three)
        x = 1;
}
```

C++ #1

```
        1:    9:int main() {
        1:   10:    C one(1), two(2);
        1:   11:    int three(3), four(4);
        1:   12:    int x = 0;
       1*:   13:    if (one < two && four < three)
condition outcomes covered 1/4
condition  0 not covered (true false)
condition  1 not covered (true)
condition outcomes covered 1/2
condition  0 not covered (true)
    #####:   14:        x = 1;
        1:   15:}
```

C++ #1

```
        1:    9:int main() {
        1:   10:    C one(1), two(2);
        1:   11:    int three(3), four(4);
        1:   12:    int x = 0;
       1*:   13:    if (one < two && four < three)
condition outcomes covered 1/4
condition  0 not covered (true false)
condition  1 not covered (true)
condition outcomes covered 1/2
condition  0 not covered (true)
    #####:   14:        x = 1;
        1:   15:}
```

gcc uses a temporary for the `if`

D #1

```
       1:    3:void main()
       -:    4:{
       1:    5:    stdin
       -:    6:        .byLineCopy
       -:    7:        .array
       3:    8:        .sort!((a, b) => a > b)
       1:    9:        .each!writeln;
       -:   10:}
```

D #1

```
        1:    3:void main()
        -:    4:{
        1:    5:    stdin
        -:    6:        .byLineCopy
        -:    7:        .array
        3:    8:        .sort!((a, b) => a > b)
        1:    9:        .each!writeln;
        -:   10:}
```

string.d.gcov:

```
        -:  251:    {
        -:  252:        import core.stdc.string : memcmp;
        -:  253:
        5:  254:        const ret = memcmp( s1.ptr, s2.ptr, len );
        5:  255:        if ( ret )
condition outcomes covered 1/2
condition  0 not covered (true)
    #####:  256:            return ret;
        -:  257:    }
        5:  258:    return (s1.length > s2.length) - (s1.length < s2.length);
```

C #1

```
        2:    1:int lt(int x, int y) {
        2:    2:    return x < y;
        -:    3:}
        -:    4:
        1:    5:int main() {
        1:    6:    int one = 1, two = 2;
        1:    7:    int three = 3, four = 4;
        1:    8:    int x = 0;
        1:    9:    if (lt(one, two) && lt(four, three))
condition outcomes covered 1/4
condition  0 not covered (true false)
condition  1 not covered (true)
    #####:   10:        x = 1;
        -:   11:}
```

C #2

```
        1:    1:int main() {
        1:    2:    int one = 1, two = 2;
        1:    3:    int three = 3, four = 4;
        1:    4:    int x = 0;
       1*:    5:    int v = one < two && three < four;
condition outcomes covered 2/4
condition  0 not covered (false)
condition  1 not covered (false)
        1:    6:    if (v)
condition outcomes covered 1/2
condition  0 not covered (false)
        1:    7:        x = 1;
        -:    8:    else
    #####:    9:        x = -1;
        -:   10:}
```

C #3

```
        1:    1:int ternary(int a, int b) {
       1*:    2:    int x = (a || b) ? f() : g();
condition outcomes covered 1/4
condition  0 not covered (false)
condition  1 not covered (true false)
        -:    3:}
```

C #4

```
        1:    1:int main() {
        1:    2:    int a = 0, b = 3, c = 2;
        1:    3:    int x = 0;
       1*:    4:    if ((a && b) || (c && a))
condition outcomes covered 2/8
condition  0 not covered (true)
condition  1 not covered (true false)
condition  2 not covered (true false)
condition  3 not covered (true)
    #####:    5:        x = 1;
```

Current status

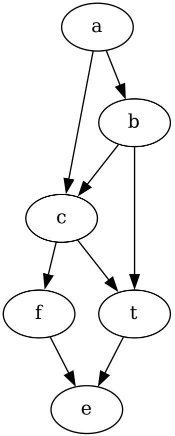
- Condition profiling is currently **pending review**
- Inferring conditionals from the CFG is **accurate**, but sometimes surprising
- Approach is sensitive to **frontend decisions**
- Reports can be unwieldy; see **lcov**
- No integration with build systems and testing frameworks

Algorithm

```
if ((a && b) || c) {
    // t
} else {
    // f
}
// e
```

```
_a:
  if (a) goto _b
  else   goto _c
_b:
  if (b) goto _t
  else   goto _c
_c:
  if (c) goto _t
  else   goto _f
_t:
  goto _e
_f:
  goto _e
_e:
```

```
if ((a && b) || c) {
    // t
} else {
    // f
}
// e
```

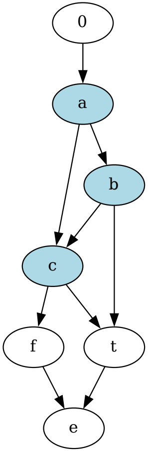


Control flow graph

- Directed graph
- Nodes are an uninterruptible sequence of instructions
- Edges are next possible paths of execution
- Edges are labelled fallthrough, true/false (conditional), complex
- Fallthrough and conditional are mutually exclusive

Act I: Inferring decisions

```
if ((a && b) || c) {
    // t
} else {
    // f
}
// e
```



Observation

$$\bigcup \{ \, Succ(v) \mid v \in B \, \} = N[B]$$

$$N[B] = B \cup O_B$$

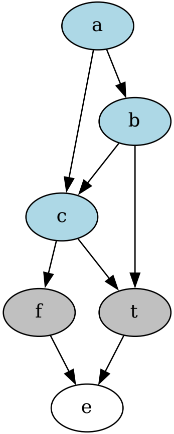
B	is a decision (boolean expression)
O_B	is the *outcome* of B
$N(B)$	is the *open neighborhood* of B
$N[B]$	is the *closed neighborhood* of B

uninterruptible

```
if ((a && b) || c) {

} else {

}
```

outcome

$$\bigcup \{\, Succ(v) \mid v \in B \,\}$$



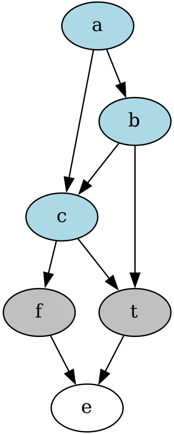
uninterruptible

```
if ((a && b) || c) {

} else {

}
```

outcome

$$E(B) = \{\, (u, v) \in E \mid u \in B, v \in N[B] \,\}$$

All edges in $E(B)$ are conditional

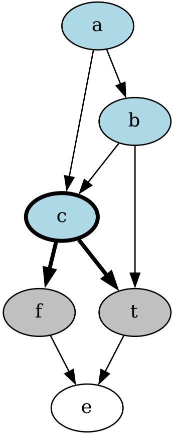


uninterruptible

```
if ((a && b) || c) {

} else {

}
```

outcome

$$Succ(B_{\Omega}) = O_B$$



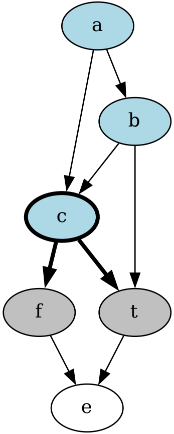
uninterruptible

```
if ((a && b) || c) {

} else {

}
```

outcome

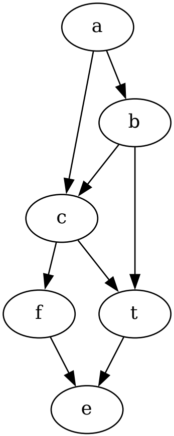
no goto

Can not goto to/from the middle of an *expression*

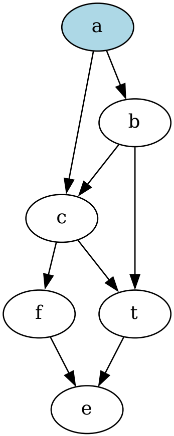


Reachable-by-condition-edge (BFS)

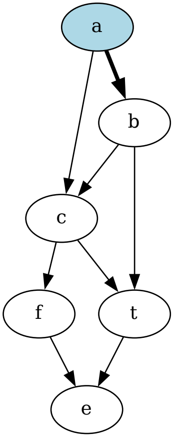
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



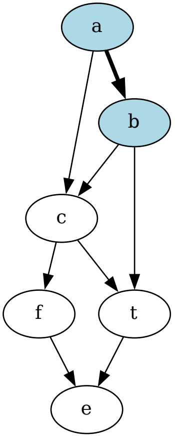
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



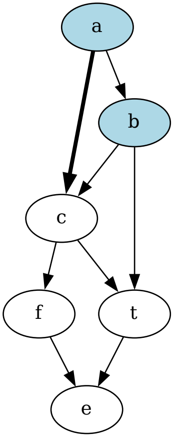
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



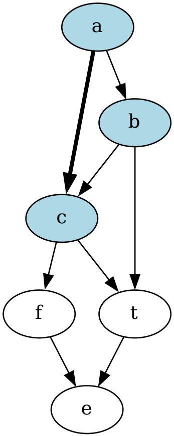
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



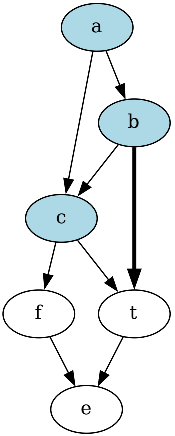
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



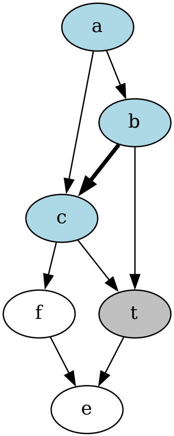
```
if ((a && b) || c) {
     // t
} else {
     // f
}
```



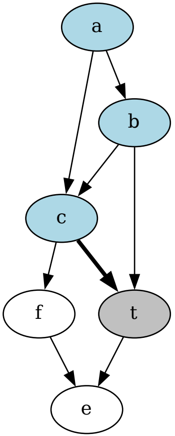
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



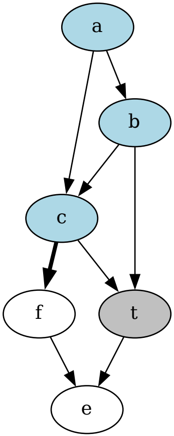
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



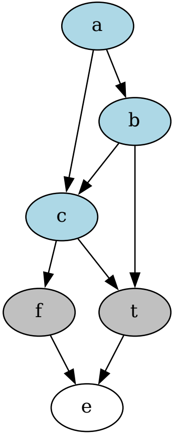
```
if ((a && b) || c) {
    // t
} else {
    // f
}
```



```
if ((a && b) || c) {
    // t
} else {
    // f
}
```

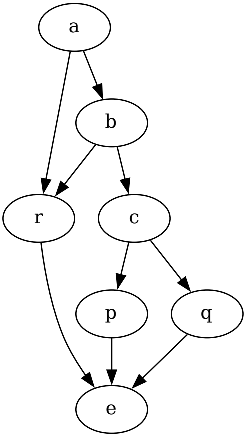


```
if ((a && b) || c) {
    // t
} else {
    // f
}
```

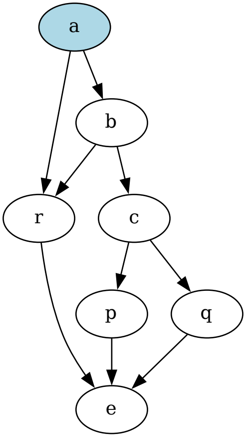


```
function REACH(v0, vp)
    R ← { }
    Q ← QUEUE(v0)
    repeat
        v ← POP(Q)
        for s in Succs(v) do
            skip if s ∈ R
            skip if IS-SAME(s, vp)
            skip if IS-BACK-EDGE(v, s)
            skip if ¬ DOMINATED-BY(s, v0)
            skip if ¬ IS-CONDITIONAL(s)
            ENQUEUE(Q, s)
            ADD(R, s)
    until EMPTY(Q)
    return R
```

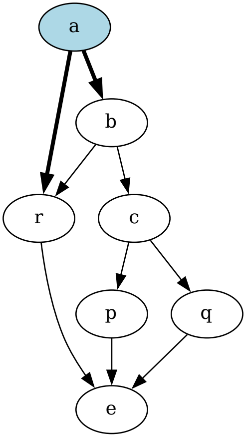
```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



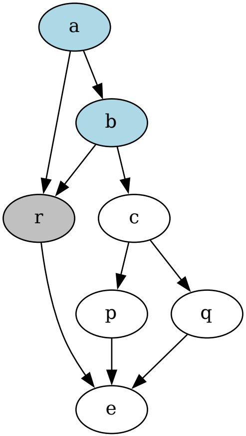
```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



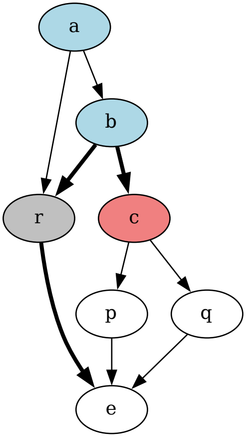
```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



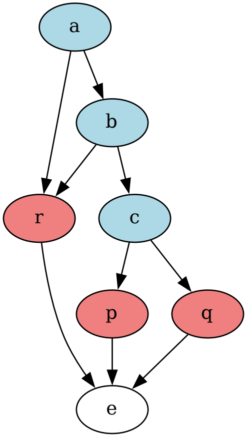
```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



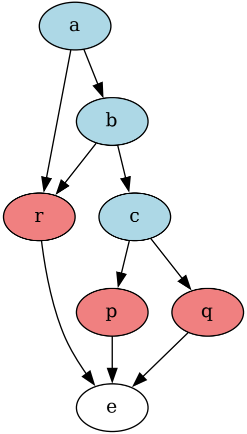
```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```

$$C = (G, G')$$

$$\forall_e \in E(G) \bullet cond(e)$$
$$\Rightarrow O_B \subset N[G]$$
$$\Rightarrow B \subseteq G$$

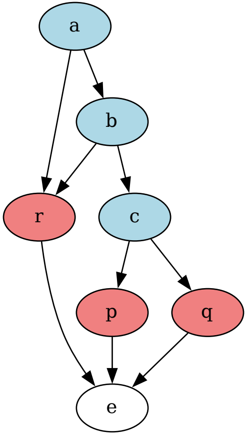


```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```



No path from **then** to **else**

```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```

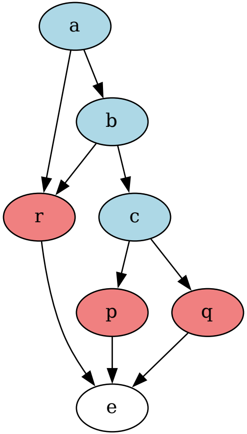


$$\forall_v \in then(B) \bullet B \subset A(v)$$
$$\forall_v \in else(B) \bullet B \subset A(v)$$

where $A(v)$ are the ancestors of v

```
if (a && b) {
    if (c) {
        // p
    } else {
        // q
    }
} else {
    // r
}
```

outcome

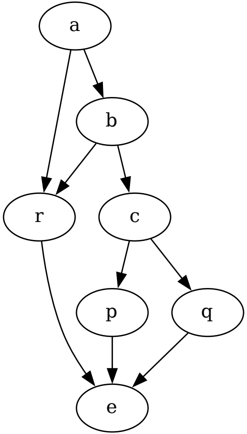


$$\forall_v \in \mathit{then}(B) \bullet B \subset A(v)$$

$$\forall_v \in \mathit{else}(B) \bullet B \subset A(v)$$

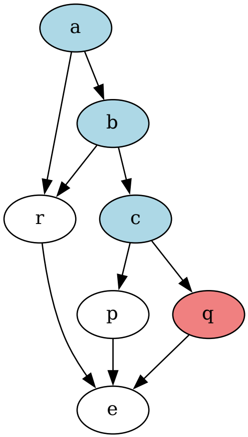
where $A(v)$ are the ancestors of v

```
if (a && b) {
    if (c) // p
    else   // q
} else {
    // r
}
```



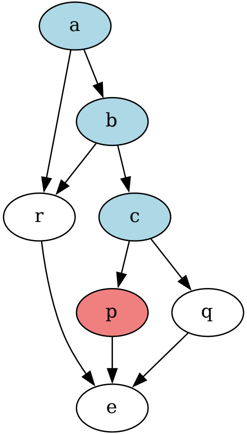
```
if (a && b) {
    if (c)  // p
    else    // q
} else {
    // r
}
```

$$A(q) = \{ c, b, a \}$$



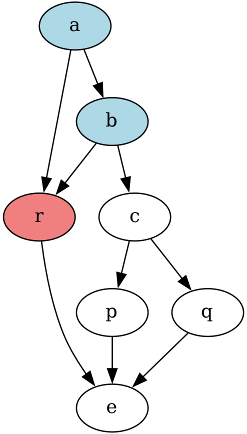
```
if (a && b) {
    if (c) // p
    else   // q
} else {
    // r
}
```

$$A(q) = \{ c, b, a \}$$
$$A(p) = \{ c, b, a \}$$



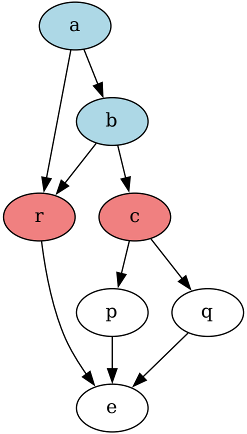
```
if (a && b) {
     if (c) // p
     else   // q
} else {
     // r
}
```

$$A(q) = \{\, c, b, a \,\}$$
$$A(p) = \{\, c, b, a \,\}$$
$$A(r) = \{\, b, a \,\}$$



```
if (a && b) {
    if (c) // p
    else   // q
} else {
    // r
}
```

$$A(q) = \{\, c, b, a \,\}$$
$$A(p) = \{\, c, b, a \,\}$$
$$A(r) = \{\, b, a \,\}$$
$$B = \bigcap \{\, A(q), A(p), A(r) \,\}$$
$$= \{\, a, b \,\}$$
$$O_B = \{\, r, c \,\}$$



Problem

BFS needs to start at left-most term B_0

Solution

Process program depth-first, mark when processed

- If v is fallthrough $\Rightarrow$ mark and continue
- If v is conditional $\Rightarrow$ is B_0 and B are marked
- If v is marked $\Rightarrow$ continue

Note

May lead to expressions being processed "out of order"

```
function FIND-DECISION(v_0, v_p)
    G ← REACH(v_0, v_p)
    if |G| = 1 then
        return G
    B ← G
    for n in N(G) do
        P ← { }
        for v in Preds(n) do
            P ← P ∪ A_G(v)
        B ← B ∩ P
    return B
```

```
cond_reachable_from (p, post, reachable, G);
if (G.length () == 1) {
    out.safe_push (p);
    return;
}

neighborhood (G, reachable, NG);
bitmap_copy (expr, reachable);

for (const basic_block neighbor : NG) {
    bitmap_clear (ancestors);
    for (edge e : neighbor->preds)
        ancestors_of (e->src, p, reachable, ancestors);
    bitmap_and (expr, expr, ancestors);
}

for (const basic_block b : G)
    if (bitmap_bit_p (expr, b->index))
        out.safe_push (b);
out.sort (cmp_index_map, &ctx.index_map);
```

```
function FIND-ALL-DECISIONS(G)
    R ← { }
    for v_0 ← DEPTH-FIRST(G) do
        skip if MARKED(v_0)
        if IS-CONDITIONAL(v_0) then
            v_p ← GET-POST-DOMINATOR(v_0)
            B ← FIND-DECISION(v_0, v_p)
            ADD(R, B)
            MARK(B)
        else
            MARK(v_0)
    return R
```

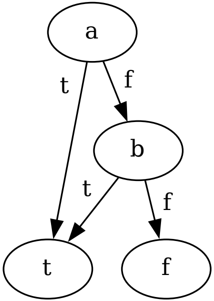
Act II: The masking vector

When masking happens

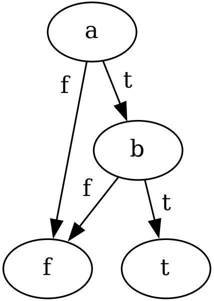
```
* || true
* && false
```

Observation

Boolean expression are are isomorphic under the operator



`a || b`



`a && b`

Proposition

Boolean expression are are isomorphic under the operator

Proof

De Morgan's Laws

$$\neg(P \land Q) \equiv \neg P \lor \neg Q$$
$$\neg(P \lor Q) \equiv \neg P \land \neg Q$$

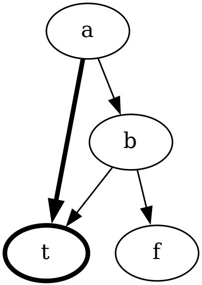
Implication

We don't need to know the operator, only the graph shape

When masking happens (in CFG)

When a value c is changed (taking a different edge at v_c) and we still end in the same outcome node

```
f = a || b
```



`f 1 0` `f 1 1` `f 0 1`

Observation

Masking happens at nodes with *multiple* predecessors

Implication

Multiple predecessors means short circuiting edge

Implication

We know where to start searching

Association

$$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$$
$$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$$

Implication

We can re-write expressions to an *alternating* form

$$(A \vee B) \vee ((C \wedge D) \vee E)$$
$$A \vee B \vee (C \wedge D) \vee E$$

Observation

Masking propagates until the operator changes

$$A \wedge (B \vee C \vee D)$$

$D = t$ masks B, C, but not A

Subexpressions can mask

$$A \wedge (B \vee C)$$

$C = f$ masks A, but not B

$$F = B \vee C$$

$$A \wedge F$$

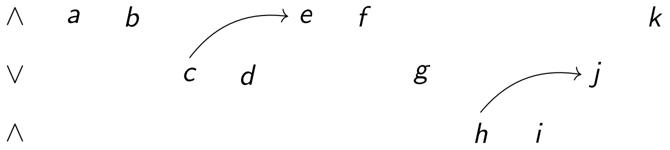
Observation

The *last term* S_Ω in a subexpression can short circuit the superexpression

$$a \wedge b \wedge (c \vee d) \wedge e \wedge f \wedge (g \vee (h \wedge i) \vee j) \wedge k$$

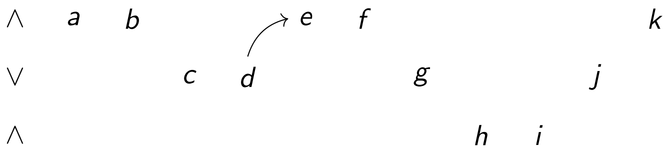
$\wedge$	a	b			e	f					k
$\vee$			c	d			g			j	
$\wedge$								h	i		

$$a \wedge b \wedge (c \vee d) \wedge e \wedge f \wedge (g \vee (h \wedge i) \vee j) \wedge k$$



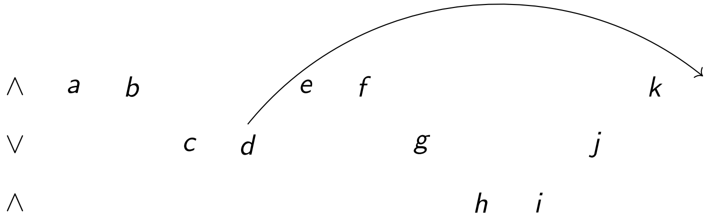
$$c = true, h = true$$

$$a \wedge b \wedge (c \vee d) \wedge e \wedge f \wedge (g \vee (h \wedge i) \vee j) \wedge k$$



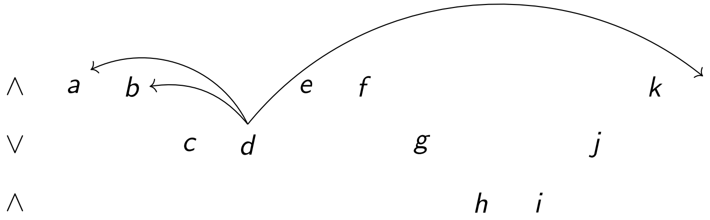
$d = true$

$$a \wedge b \wedge (c \vee d) \wedge e \wedge f \wedge (g \vee (h \wedge i) \vee j) \wedge k$$



$d = \mathit{false}$

$$a \wedge b \wedge (c \vee d) \wedge e \wedge f \wedge (g \vee (h \wedge i) \vee j) \wedge k$$



$d = false$

$$Succ(B_\Omega) = O_B$$

Observation

On evaluating a condition; either

- Short-circuit right operands
- Evaluate next operand

Implication

If one edge is a short circuiting edge, the other must be a masking edge

Problem

Given a pair of incoming edges, which is masking and which is short circuting?

Proposition

An ordering $v_n < v_m$ if v_n is a left operand and v_m is a right operand in the same expression

Problem

Given a pair of incoming edges, which is masking and which is short circuting?

Proposition

An ordering $v_n < v_m$ if v_n is a left operand and v_m is a right operand in the same expression

Solution

Topological sort

Given $\{ v_n, v_m \} = Preds(v), v_n < v_m$ then

$$
\begin{aligned}
v_n &= S_\Omega \\
O_S &= Succ(v_n)
\end{aligned}
$$

where S is a subexpression of B ($S \subset B$)

Implication

When (v_m, v) is taken, S are masked

$$
\begin{aligned}
v_n, v_m &= Preds(v) \\
v_n &= S_\Omega \\
O_S &= Succ(v_n)
\end{aligned}
$$

Remember

$$
\begin{aligned}
N[B] &= \bigcup \{\, Succ(v) \mid v \in B \,\} \\
N(B) &= O_B
\end{aligned}
$$

Everything that applies to the superexpression B applies to the subexpression S

Problem

Given a node v with $|Preds(v)| \geq 2$, find the nodes masked when taking an edge to v

Intermediate problem

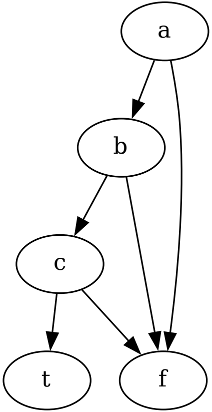
There can be more than one masking edge

Solution

$$\{\, (v_n, v_m) \in Preds(v) \times Preds(v) \mid v_n < v_m \,\}$$

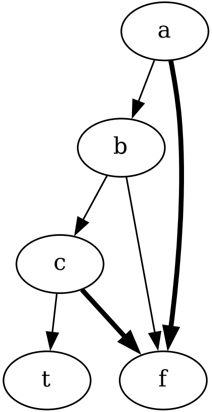
$$\{\, (v_n, v_m) \in \mathit{Preds}(v)^2 \mid v_n < v_m \,\}$$

```
a && b && c
```



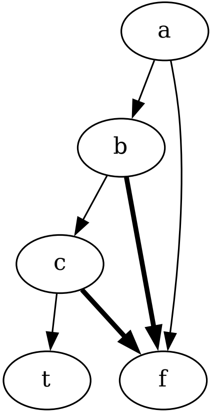
$$\{ (v_n, v_m) \in \mathit{Preds}(v)^2 \mid v_n < v_m \}$$

`a && b && c`



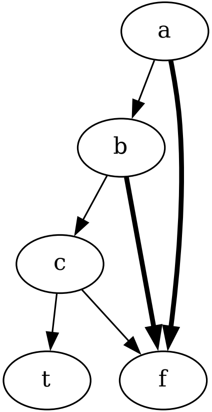
$$\{ (v_n, v_m) \in Preds(v)^2 \mid v_n < v_m \}$$

`a && b && c`



$$\{\, (v_n, v_m) \in \mathit{Preds}(v)^2 \mid v_n < v_m \,\}$$

```
a && b && c
```



Remember

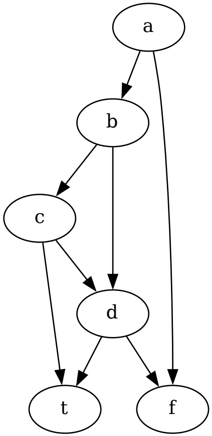
$$N[S] = \bigcup \{ Succ(v) \mid v \in S \}$$
$$N(S) = O_S$$

Implication

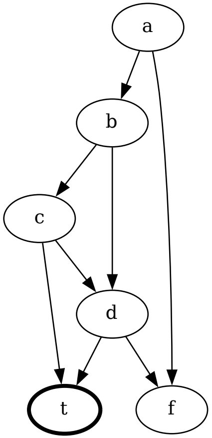
$$Succs(v_k) \subset S_n \Rightarrow S_{n+1} = S_n \cup \{ v_k \}$$
$$S_0 = O_S$$
$$S = S^f - S_0$$

where S^f is the fixed point $S_{n+1} = S_n$

```
A && ((B && C) || D)
```



```
A && ((B && C) || D)
```

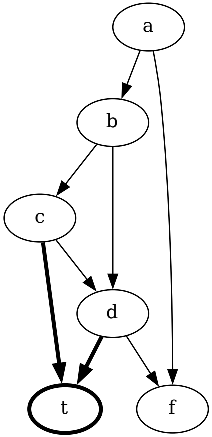


```
A && ((B && C) || D)
```

(c, t) short circuits

(d, t) masks

$c < d$



`A && ((B && C) || D)`

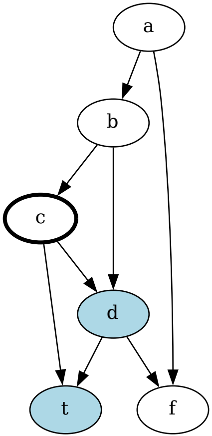
(c, t) short circuits

(d, t) masks

$c < d$

$$S_\Omega = c$$

$$O_S = Succ(c) = \{ d, t \}$$

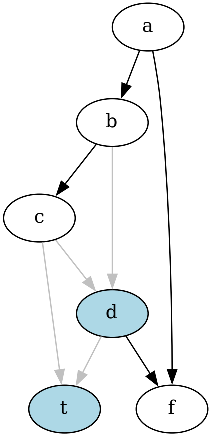


	(c, t)	short circuits
`A && ((B && C) || D)`	(d, t)	masks
	$c < d$	

$$S_\Omega = c$$

$$O_S = Succ(c) = \{\, d, t \,\}$$

$$S_0 = d, t$$



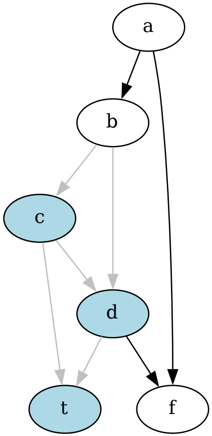
`A && ((B && C) || D)`

(c, t) short circuits

(d, t) masks

$c < d$

$$S_\Omega = c$$
$$O_S = Succ(c) = \{\, d, t \,\}$$
$$S_0 = d, t$$
$$S_1 = S_0 + c = \{\, d, t, c \,\}$$



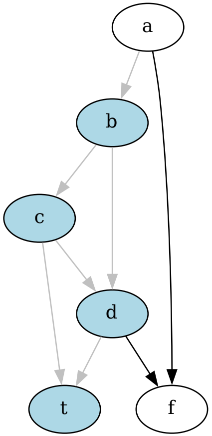
`A && ((B && C) || D)`

(c, t) short circuits

(d, t) masks

$c < d$

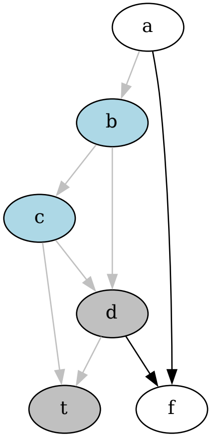
$$S_\Omega = c$$
$$O_S = Succ(c) = \{ d, t \}$$
$$S_0 = d, t$$
$$S_1 = S_0 + c = \{ d, t, c \}$$
$$S_2 = S_1 + b = \{ d, t, c, b \}$$
$$S^f = S_2$$



`A && ((B && C) || D)`

(c, t)	short circuits
(d, t)	masks
$c < d$	

$$S_\Omega = c$$
$$O_S = Succ(c) = \{ d, t \}$$
$$S_0 = d, t$$
$$S_1 = S_0 + c = \{ d, t, c \}$$
$$S_2 = S_1 + b = \{ d, t, c, b \}$$
$$S^f = S_2$$
$$S = S^f - O_S$$
$$S = \{ b, c \}$$



```
function MASKING-VECTOR(B)
    M ← { }
    for b in B ∪ O_B do
        for (u, v) in { (u, v) | Pred(b)^2, u < v } do
            Q ← QUEUE(u)
            MARK(Succ(u))
            repeat
                q ← POP(Q)
                skip if MARKED(Succ(q))
                MARK(q)
                ADD(M(v, b), q)
                for p in Pred(q) do
                    skip if ¬ IS-CONDITIONAL(p)
                    skip if IS-BACK-EDGE(q, p)
                    skip if MARKED(p)
                    skip if p ∉ B
                    ENQUEUE(Q, p)
            until EMPTY(Q)
    return M
```

Act III: Instrumentation

Instrumentation must be fast

Remember

There is an ordering $v_n < v_m$ if v_n is a left operand and v_m is a right operand in the same expression

Implication

We can sort B

Implication

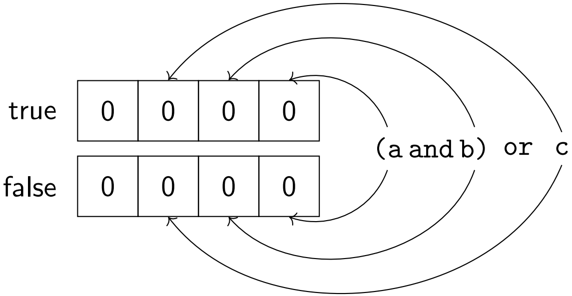
There is a bijection $f : B \to \mathbb{N}$

Global accumulators

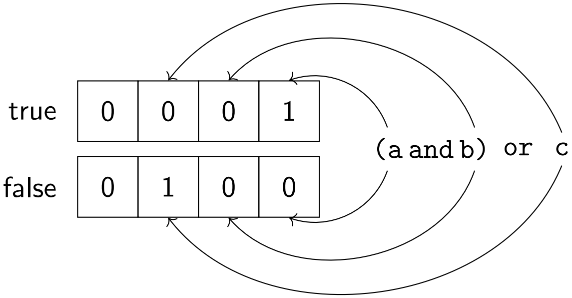
true	0	0	0	0
false	0	0	0	0

`(a and b) or c`

Global accumulators

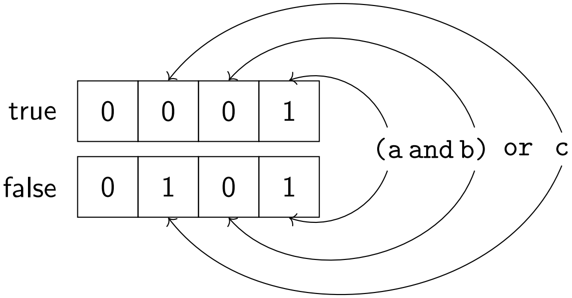


Global accumulators



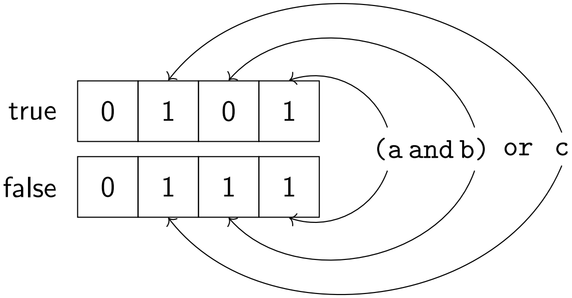
`f 0 0 1`

Global accumulators




```
f 0 0 1
f 0 0 0
```

Global accumulators




```
f 0 0 1
f 0 0 0
f 1 0 0
```

Local accumulators

$$acc \leftarrow acc \cup B(E(u, v))$$
$$acc \leftarrow acc \cap M(E(u, v))$$

where $E(u, v)$ is the edge taken and $M(E)$ are nodes masked for E

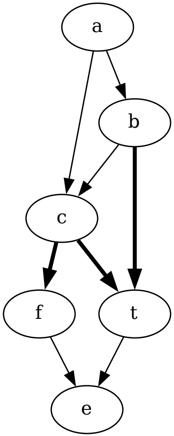
Remember

There is bijection $f : B \to \mathbb{N}$

a || b

```
_prelude_fn:
  _t = {0}
  _f = {0}
_a:
  if (a)
    _t |= 0x01
    goto _T
  else
    _f |= 0x01
    goto _b

_b:
  if (b)
    _t &= 0x01
    _f &= 0x01
    _t |= 0x02
    goto _T
  else
    _f |= 0x02
    goto _F

_T:
  goto _E
_F:
  goto _E
_E:
  _fn_t |= _t
  _fn_f |= _f
```

Local accumulators are flushed (bitwise-or) on edge-to-outcome



Thank you

Me

Who Jørgen Kvalsvik

How `<j@lambda.is>`

Where Woven by Toyota in Tokyo, Japan

Resources

Hayhurst (2001) A Practical Tutorial on Modified Condition/ Decision Coverage.

Chilenski (2001) An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion.