The following slicing algorithm, which is heavily tailored for the Clang Static Analyzer, somewhat redefines the goal of slicing: traditionally, we'd gather a set of relevant variables, which would initially be the variables in the slicing criterion, and expand it if we find a variable relevant to any variable within the set. Along this, we'd also calculate a set of relevant statements, which is a subset of the program, either executable on it's own or not.

My approach is vastly different: the reason behind this is the function called trackExpressionValue. In a nutshell, trackExpressionValue tracks a variable, and adds notes at where the tracked variable was written, alongside the path that leads there, essentially gathering the set of relevant statements. Teaching it to discover variables immediate relevant to the tracked variable, we could (in theory) calculate the set of relevant variables transitively: if X is tracked, and A would be added to the set because of X, and B would be added because of A, tracking X would trigger tracking A, and that would trigger tracking B.

Our starting point (slicing criterion) is the ExplodedNode and CFGBlock where report originates from, and the variable that was initially tracked.

```
// Traverses the bugpath to discover all reaching definitions, even
// in nested stackframes. Returns true if any reaching definition is
// found.
function conditionTracker(N : ExplodedNode, X : variable,
                          Report : BugReport, OriginB : OriginB)

  SetOfReachingDefinitions = reaching definitions to X in N

  while N is not null do

    // Look for the function call to F
    N := N->getFirstPred()
    if not N->getLocationAs<CallExit>() then
      continue
    fi

    B := CFGBlock associated with N
    if there is no path from B to OriginB then
      continue
    fi

    CallEnterN := N's matching CallEnter
    if CallEnterN is null then
      continue // and assert that we're in a top frame
    fi
```

```
    // CallEnterN's pred should be in the same CFG as OriginB.
    CalleeE := CFGElement associated with CallEnterN->getFirstPred()

    ParamSet := set of F's parameters to which
                X is passed to as a non-const reference

    for all NewX variables in ParamSet do
      if conditionTracker(NC, NewX, Report, F's exit block) then
        SetOfReachingDefinitions += CalleeE
      fi
    od

    // X is global/static, or a field and F is a method, etc...
    if F would have access to X regardless then
      if conditionTracker(NC, X, Report, F's exit block) then
        SetOfReachingDefinitions += CalleeE
      fi
    fi
  od

  for all RD CFGElementRefs in SetOfReachingDefinitions do
    track control dependencies of RD
  od

  return not whether SetOfReachingDefinitions is empty
```

Example 1:

```
01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin(); // no note
06 }
07
08 int main() {
09   int *x = 0; // x initialized to 0
10   flag = 1;
11   foo();
12   if (flag) // assumed false
13     x = new int;
14   foo();
15
16   if (flag) // assumed true
17     *x = 5; // warn
```

```
18   if (coin())
19     x = new int;
20 }

foo():
  [B2(ENTRY)] -> [B1] -> [B0 (EXIT)]

main():
  [B7 (ENTRY)] -> [B6] -----> [B4] -------> [B2] ---> [B0 (EXIT)]
                    \          /  \          /  \          /
                     -> [B5] ->    -> [B3] ->    -> [B1] ->
```

Our slicing criterion is (B3, {x}).

1. Block 0, 1, 2 leads no path to B3, skip them.
2. B3 doesn't write x (only the value it points to).
3. B4 contains a function call to foo: x isn't passed to any of it's parameters as a non-const reference, nor is it a static, global, or field variable.
4. B5 writes x: B5 doesn't post dominate any block or vice versa in SetOfLastWrites (because it's emptry), add it to the set.
5. B6 writes x: it doesn't post dominate B5 or vice versa, add it to the set.
6. B7 doesn't write x.
7. Observe SetOfLastWrites: B6 is the part of the bug path, skip it. B5 isn't, track it's control dependency, flag in block B6, line 12.

`TrackControlDependencyCondBRVisitor` start working:

1. Tracking of x on line 17:
    a. The statement on line 16 is a control dependency of 17: start tracking flag.
    b. No other control dependency is found.
2. Tracking flag on line 16:
    a. Line 12 doesn't depend on any block.
3. Tracking flag on line 12:
    a. Line 12 doesn't depend on any block.

The calculation restarts with two new slicing criterions:
(B4, flag)

1. Blocks 0, 1, 2, 3 are skipped.
2. B4 contains a function call to foo: flag is global and foo has access to it: call trackConditions with the slicing criterion (B0, flag).
    a. B0 doesn't contain a write to flag, nor a function call.
    b. B1 writes flag, and since SetOfLastWrites is empty, add it to the set.
    c. B3 doesn't contain a write to flag, nor a function call.
    d. B1 is a part of the bugpath, do nothing with it, return true.

    Since the function returned true, add B4 to SetOfLastWrites.
3. B5 has no function calls or writes to flag.
4. B6 is the same as B4, except that B4 post dominates B6, don't add it to the set.
5. B7 is the entry.

6. The only block in the set, B4, is a part of the bugpath, ignore it.

(B6, flag): Almost the same as above.

Example 2.:

```
01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin();
06 }
07
08 int main() {
09   int *x = 0;
10   foo();
11   if (flag)
12     x = new int;
13   foo();
14   x = 0; // Note that anything above this is irrelevant.
15   if (flag)
16     *x = 5; // nullptr dereference
17 }
```

```
01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin();
06 }
07
08 int main() {
09   int *x = 0;
10   foo();
11   if (flag)
12     x = new int;
13   foo();
14   if (coin())
15     x = 0; // Note that anything above this is irrelevant.
16   if (flag)
17    *x = 5; // nullptr dereference
```

```
18 }
```

Slicing criterion: (16, {x})
TODO: explain how the algorithm works here

Example 3.:

```
01 struct S {
02   int x, y;
03   S(int r, int k) {
04     if (r)
05       x = 5;
06     if (k)
07       y = 0;
08   } // warning for both x and y being uninitialized
09 };
10
11 void f(int r, int k) {
12   S s(r, k);
13 }
```

Slicing criterion: (8, {x, y})
TODO: explain how the algorithm works here

Example 4.:

```
01 int flag;
02 bool coin();
03
04 void foo() {
05   flag = coin(); // no note
06 }
07
08 int main() {
09   int *x = 0; // x initialized to 0
10   flag = 1;
11   foo();
12   if (flag) // assumed false
13     x = new int;
     if (flag)
```

```
      x = 0;
14   foo();
15
16   if (flag) // assumed true
17     *x = 5; // warn
18   if (coin())
19     x = new int;
20 }
```