14 August 2026

Change Documentation for

Testwell CTC++

Version 10.4.0

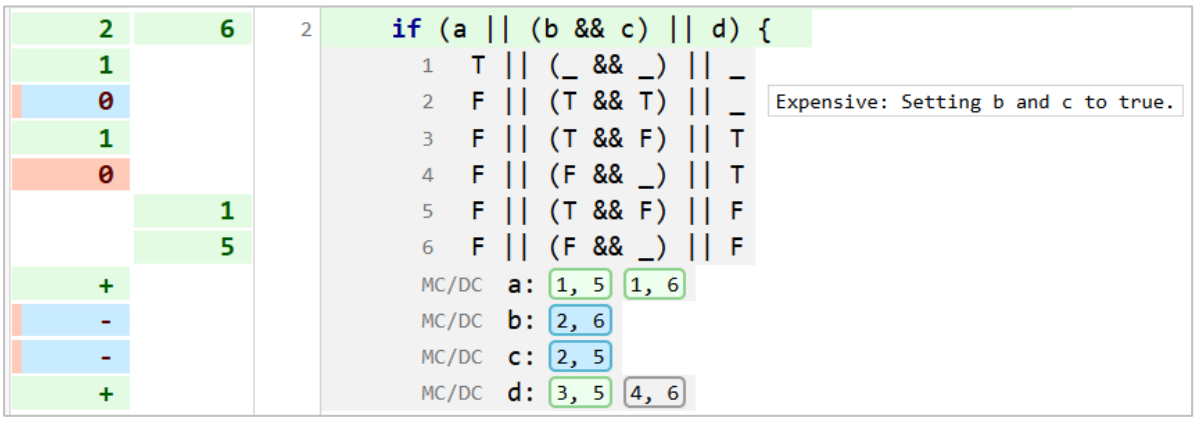
Features and Changes

Justifications for True-False-Combinations

For higher coverage levels (i.e. MC/DC, condition and multicondition coverage), justifications can now be provided on a detailed level: They can be assigned to single True-False-Combinations, using the T, F, _ combination as descriptor. From a justification file,

```
2 FTT_ | Expensive: Setting b and c to true.
```

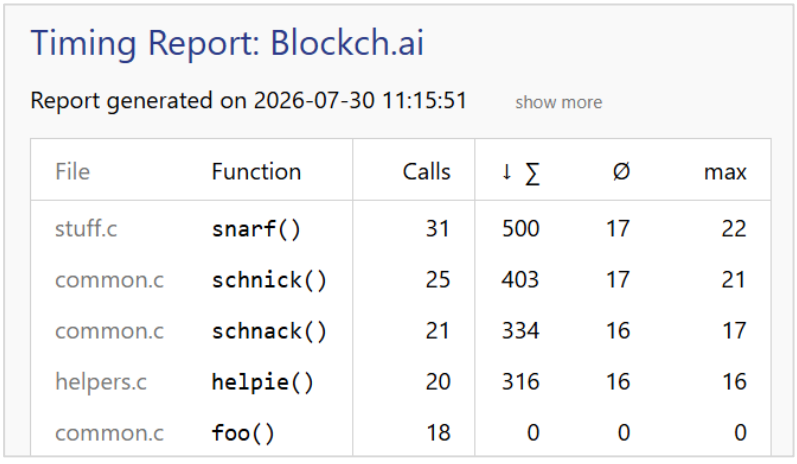
for the second combination in line 2 leads to:



Impact on multicondition is directly calculated from these justifications. For condition coverage and MC/DC, the impact is derived.

Timing in Reports

Timing or cost information (captured with instrumentation mode `ti` or `te`) can now be reported with **ctcreport**. For this purpose, the single-file template **timing.html** is provided.



Timing Report: Blockch.ai

Report generated on 2026-07-30 11:15:51 show more

File	Function	Calls	↓ ∑	Ø	max
stuff.c	snarf()	31	500	17	22
common.c	schnick()	25	403	17	21
common.c	schnack()	21	334	16	17
helpers.c	helpie()	20	316	16	16
common.c	foo()	18	0	0	0

For your own reporting, use the following new variables in a function loop, available in single-file templates or in the source code view of structured templates:

- $TotalCost$
- $AverageCost$
- $MaximumCost$
- $NumberOfCallsWithCost$

The last variable differs from $NumberOfCalls$ only if a source file has been instrumented with and without timing instrumentation.

Bug Fixes

Misplaced Pragmas

In some locations, pragmas were misplaced by **ctc** in the instrumented code. Affected locations are:

- After a loop head or lambda start like

```
for (int i = 0; i < 10; i++)
#pragma AfterForLoopHead
…

my_lambda = 5 + [](int i){
#pragma AfterLambdaBegin
…
```

- After a line directive in preprocessed code like

```
#line 1 "C:\\examples\\header.h"
#pragma pack(push, 1)
```

The misplaced pragmas could lead to uncompilable code.

Not instrumented template specializations

When a template function is generally declared and then implemented only in a specialization, like

```
template<typename T>
T max(T a, T b);

template<>
int max<int>(int a, int b) {
    return (a >= b) ? a : b;
}
```

the specialization was not instrumented, due to a change in internal **ctc** behavior starting with version 10.2.0.

Uncompilable Code with Lambda

A combination of two code elements led to uncompilable instrumented code for multicondition instrumentation:

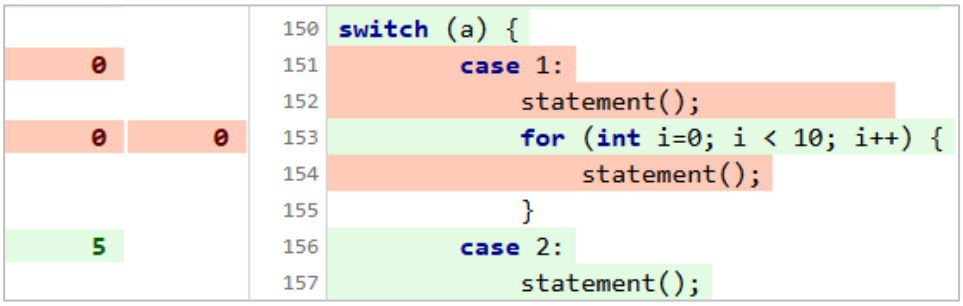
- A lambda function is used in a member initialization of a constructor,
- The constructor contains a complex Boolean decision.

Example:

```
MyClass::MyClass(char a)
      : my_member([](int x) { })
{
      if (a > 0 && a < 10) {
            do_something(a);
      }
}
```

Statement coverage in switch - case

Code elements with a true-false decision like `if`, `for`, `while`, immediately followed by a `case` after their true branch-end or loop-end were counted as executed if the following `case` itself was covered. Only statement coverage and line coverage were affected (line 153 in example below).



Wrong warning

Usually, the right-hand side of definitions with `const` are not instrumented and a warning for this behavior is issued:

```
CTC++ warning 37: Decision not instrumented (because of 'const') in file
foo.c at line 26
```

This warning was accidentally issued when the right-hand side did not contain any logical operator, but was enclosed in parentheses like:

```
const int x = (a + b);
```

Ternaries in template parameter lists

Since version 10.2.1, ternary ?-operator expressions were instrumented in template parameters, leading to not compilable code.

End of namespace not recognized

Due to different reasons, **ctc** did not recognize the end of a namespace:

- When a variable initialization was performed with `{…}`, containing a ternary-? operator with a function call, like

```
namespace MyNameSpace {
    int myInteger {b ? foo(b) : b};
};
```

- Function template calls inside template parameters

```
template <typename T, typename = ab::EnableIf<ab::canConvert<T>()>>
```

Typically, this leads to wrongly named functions, and possibly to wrong code variants.

Attributes for function parameters

An attribute for the first function parameter like in

```
int foo([[maybe_unused]] int a, …
```

prevented **ctc** from instrumenting this function.

Missing justification forwarding

Justifications were not properly forwarded to loop heads immediately after an `if` branch. In the following example, the justification of the false counter in line 4 should lead to a full justification of all the code following.