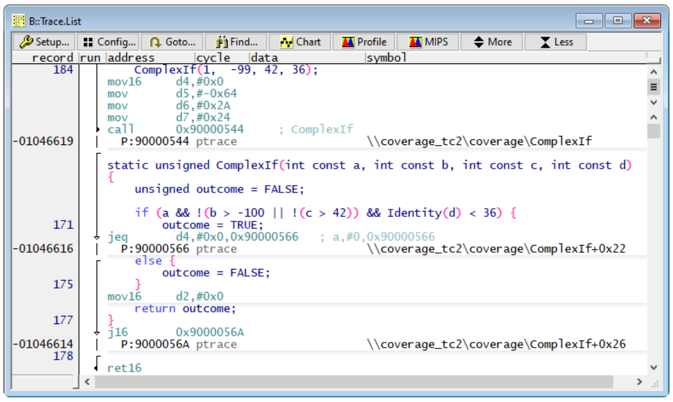
TECHNICAL ARTICLE

TRACE32 Multi-Mode MC/DC Coverage

MC/DC coverage is recommended in most safety standards for adequate testing of software with a high safety level. Lauterbach first introduced its solution for MC/DC coverage in 2018. In our initial solution, we focused on MC/DC coverage solely based on program flow tracing. The goal was to support the widest possible range of core architectures and trace protocols. However, practical experience has shown that we need to extend this approach with some instrumentation to achieve completeness. Lauterbach calls the extended solution TRACE32 Multi-Mode MC/DC Coverage. Multi-mode coverage includes three types of instrumentation: no instrumentation, targeted instrumentation, and full instrumentation only as a fallback. For instrumentation, the goal was to ensure minimal memory footprint and almost no runtime overhead.



```
   record |run |address          |cycle   |data           |symbol
      184 |    |    ComplexIf(1,  -99, 42, 36);
          |    |mov16      d4,#0x0
          |    |mov        d5,#-0x64
          |    |mov        d6,#0x2A
          |    |mov        d7,#0x24
          |    |call       0x90000544     ; ComplexIf
-01046619 |    |  P:90000544 ptrace                           \\coverage_tc2\coverage\ComplexIf
          |    |static unsigned ComplexIf(int const a, int const b, int const c, int const d)
          |    |{
          |    |    unsigned outcome = FALSE;
          |    |
          |    |    if (a && !(b > -100 || !(c > 42)) && Identity(d) < 36) {
      171 |    |        outcome = TRUE;
          |    |jeq        d4,#0x0,0x90000566   ; a,#0,0x90000566
-01046616 |    |  P:90000566 ptrace                           \\coverage_tc2\coverage\ComplexIf+0x22
          |    |    else {
          |    |        outcome = FALSE;
      175 |    |    }
          |    |mov16      d2,#0x0
          |    |    return outcome;
      177 |    |}
          |    |j16        0x9000056A
-01046614 |    |  P:9000056A ptrace                           \\coverage_tc2\coverage\ComplexIf+0x26
      178 |    |
          |    |ret16
```

Figure 1: Listing of recorded program flow trace.

The purpose of this paper is to introduce TRACE32 Multi-Mode MC/DC Coverage. The basis is still the program flow trace. So, we begin with a brief introduction to this technology and summarize the challenges that have motivated our extension.

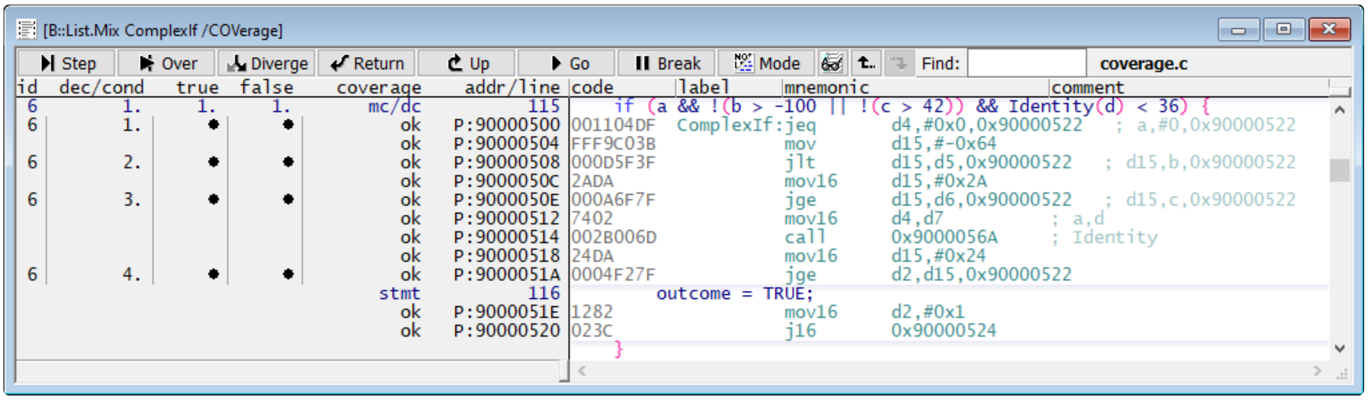
MC/DC Coverage via Program Flow Trace

The basis for MC/DC coverage analysis is the program flow trace recording (see figure 1). A parallel or serial off-chip trace port is certainly best for recording a suitably large amount of trace for analysis. But also, a large onchip trace memory or a trace recording done in a TRACE32 Instruction Set Simulator offer a good basis. For a complete MC/DC coverage analysis via the program flow trace, four criteria must be met:

Criteria #1: TRACE32 has to know the structure and the position of the decisions within the source code. Since the decision details are not included in the debug information generated by the compiler, Lauterbach offers its own Clang-based command line tool named t32cast for this purpose. t32cast analyzes the C/C++ sources and generates an extended code analysis (.eca) file for each source file, that provides the decision details.

Criteria #2: Each decision is composed of one or more (atomic) conditions. And each condition in the source code must be represented by a conditional branch or by a conditional instruction at object code level.

Criteria #3: An exact mapping of the decisions in the source code to the conditional branches/instructions in the object code is required.



```
[B::List.Mix ComplexIf /COVerage]
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:            coverage.c
id  dec/cond  true  false  coverage  addr/line  code      label       mnemonic   comment
6         1.    1.     1.     mc/dc        115      if (a && !(b > -100 || !(c > 42)) && Identity(d) < 36) {
6         1.    ●      ●         ok P:90000500 001104DF  ComplexIf:jeq    d4,#0x0,0x90000522    ; a,#0,0x90000522
                                 ok P:90000504 FFF9C03B            mov    d15,#-0x64
6         2.    ●      ●         ok P:90000508 000D5F3F            jlt    d15,d5,0x90000522   ; d15,b,0x90000522
                                 ok P:9000050C 2ADA                mov16  d15,#0x2A
6         3.    ●      ●         ok P:9000050E 000A6F7F            jge    d15,d6,0x90000522   ; d15,c,0x90000522
                                 ok P:90000512 7402                mov16  d4,d7               ; a,d
                                 ok P:90000514 002B006D            call   0x9000056A          ; Identity
                                 ok P:90000518 24DA                mov16  d15,#0x24
6         4.    ●      ●         ok P:9000051A 0004F27F            jge    d2,d15,0x90000522
                               stmt        116           outcome = TRUE;
                                 ok P:9000051E 1282                mov16  d2,#0x1
                                 ok P:90000520 023C                j16    0x90000524
                                                      }
```

Figure 2: Source code decision and its mapping to conditional branches.

Criteria #4: It must be observable by the conditional branches/instructions in the recorded program flow trace whether a source code condition was evaluated true or false.

So far, the basic concept. The screenshot in figure 2 illustrates what has been described so far.

Observability Gaps and Their Causes

Practice has shown that criteria #2, #3 and #4 are not always fulfilled in every test scenario. When this is the case, Lauterbach speaks of observability gaps. Observability gap means that TRACE32 cannot detect whether a condition has been evaluated as true or false at a certain point in the program flow trace. In this case, no MC/DC coverage result can be displayed for the related decision. Here are the most likely causes of observability gaps and the countermeasures that need to be taken:

1. No dedicated compiler support

First of all, you should consider writing code coverage friendly code. Nested decisions or simple decisions in the assignment context, such as `return a==b`, may cause observability gaps. Here it is not guaranteed for every compiler that all (atomic) conditions are represented by a conditional branch/instruction on the object code level. This can be easily avoided by following some simple coding guidelines summarized by Lauterbach. However, if one cannot or will not modify the source code of a colleague or external provider, gaps are unavoidable. Criterion #2 is violated, but the observability gaps can be closed by targeted instrumentation. The second part of this article will provide details on this instrumentation mode.

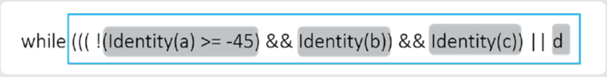
On the other hand, the large number of core architectures and the associated diversity of compilers represents a challenge. An impressive number of cores offer the possibility to generate program flow trace. And there are a big number of compilers, especially for commonly used core architectures. The result is a large amount of possible core architecture/compiler pairings. There is no generic heuristic for mapping source code decisions to conditional branches/instructions at object code level that generates an exact result for every possible pairing. In practice, TRACE32 has to tailor the mapping to the core architecture/compiler combination. Much, especially for common core/compiler combinations is already tailored.

For not yet supported core architecture/compiler pairings, for which the generic heuristic of TRACE32 does not provide an exact result, criterion #3 is not to be met. Lauterbach offers targeted instrumentation or even full instrumentation as a fallback for these cases.

2. Macros

A macro that is used in a decision can in itself contain decisions. The compiler expands all macros before compilation and handles the expanded statement as a single source block. During this step the source code locations of the decisions inside the macro are lost. In this case, criterion #3 is violated. A mapping of the inside-macro-decisions to the conditional branches/instructions is no longer possible. The resulting observability gap can be closed by targeted instrumentation.

GLOSSARY



while (((!(Identity(a) >= -45) && Identity(b)) && Identity(c)) || d

- A CONDITION (grey in the picture above) is a logical indivisible, atomic expression. It can only be "true" or "false".
- A DECISION (framed by turquoise rectangle) is a logical expression which can be composed of several (atomic) conditions separated by logical operators such as "or", "and", "not". It results in true or false.

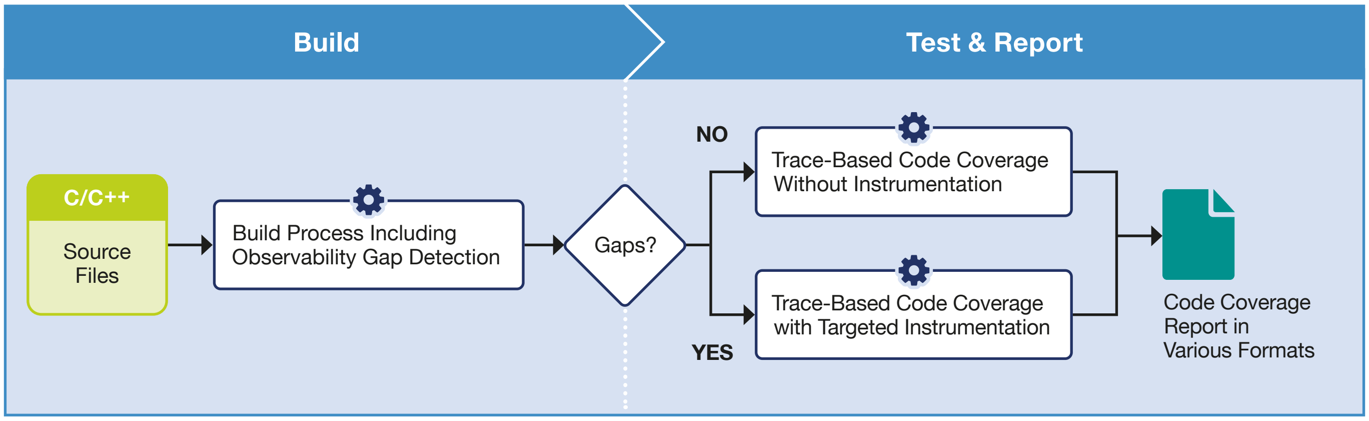

Build
Test & Report
C/C++
Source Files
Build Process Including Observability Gap Detection
Gaps?
NO
YES
Trace-Based Code Coverage Without Instrumentation
Trace-Based Code Coverage with Targeted Instrumentation
Code Coverage Report in Various Formats

Figure 3: Two step workflow for TRACE32 multi-mode MC/DC coverage.

3. Highly-optimized code

Highly-optimized code is not recommended for trace-based code coverage analysis. For one, individual conditions may not be represented by conditional branches/instructions at the object code level. Criterion #2 is violated here. However, this can be remedied easily by targeted instrumentation. Highly optimized code is particularly challenging because it may not possible to map the decisions exactly to the conditional branches/instructions. The violation of criterion #3 cannot be resolved in all cases.

Moderate optimization is recommended here. This is also advantageous because TRACE32 can display the results of the MC/DC coverage analysis clearly and in an intuitively and readable way.

4. Limitations of the trace protocol

The instruction set for a core architecture may contain conditional instructions. The compiler uses these to implement source code conditions at object code level. For trace-based code coverage to work, the trace protocol used must generate details about the execution of these conditional instructions. Unfortunately, this is not always the case. Currently there is no option that advises the compiler not to use conditional instruction. Observability gaps in program tracing are therefore inevitable. Criterion #4 is violated, but targeted instrumentation can be used to close the gaps.

5. Instruction set complexity

The challenges described in 1-4 are essentially the ones faced by cores with general-purpose RISC architecture. However, complex SoCs also contain coprocessors and special-purpose cores for which an instruction trace is generated. Examples are DSPs, configurable cores with user-defined instructions, timer IP and many more. Here, TRACE32 must always be specially adapted to the instruction set for an MC/DC coverage analysis. In this respect, it is always advisable to check with Lauterbach in good time.

Summarizing what has been described so far: a bit of source code instrumentation might be required to verify MC/DC by trace-based code coverage.

Workflow for Multi-Mode MC/DC Coverage

Since full instrumentation is only required in corner cases, we will concentrate on the two standard TRACE32 MC/DC coverage modes in the following.

- Trace-based code coverage without instrumentation
- Trace-based code coverage with targeted instrumentation

Figure 3 provides an overview of the workflow, which is organized into the two steps "Build Process" and "Test & Report".

Build Process

Figure 4 on the next page gives a detailed overview of the build process. The tasks of the build process are as follows:

1. Create a classic ELF file for the “Test & Report” step.

2. Create the .eca files that provide TRACE32 with the necessary decision details (as required by criterion #1).

3. Create an instrumented ELF file for the “Test & Report” step, in case that observability gaps have been detected.

All outputs of the build process required for the “Test & Report” step are drawn in turquoise and marked with a downward arrow in Figure 4. To generate the necessary outputs, the following TRACE32 products must be integrated into the build process.

- Clang-based command line tool t32cast; t32cast for C/C++ is free of charge and compiler independent.
- TRACE32 Instruction Set Simulator for the core architecture under test; the use of the TRACE32 Instruction Set Simulator requires a paid floating license.

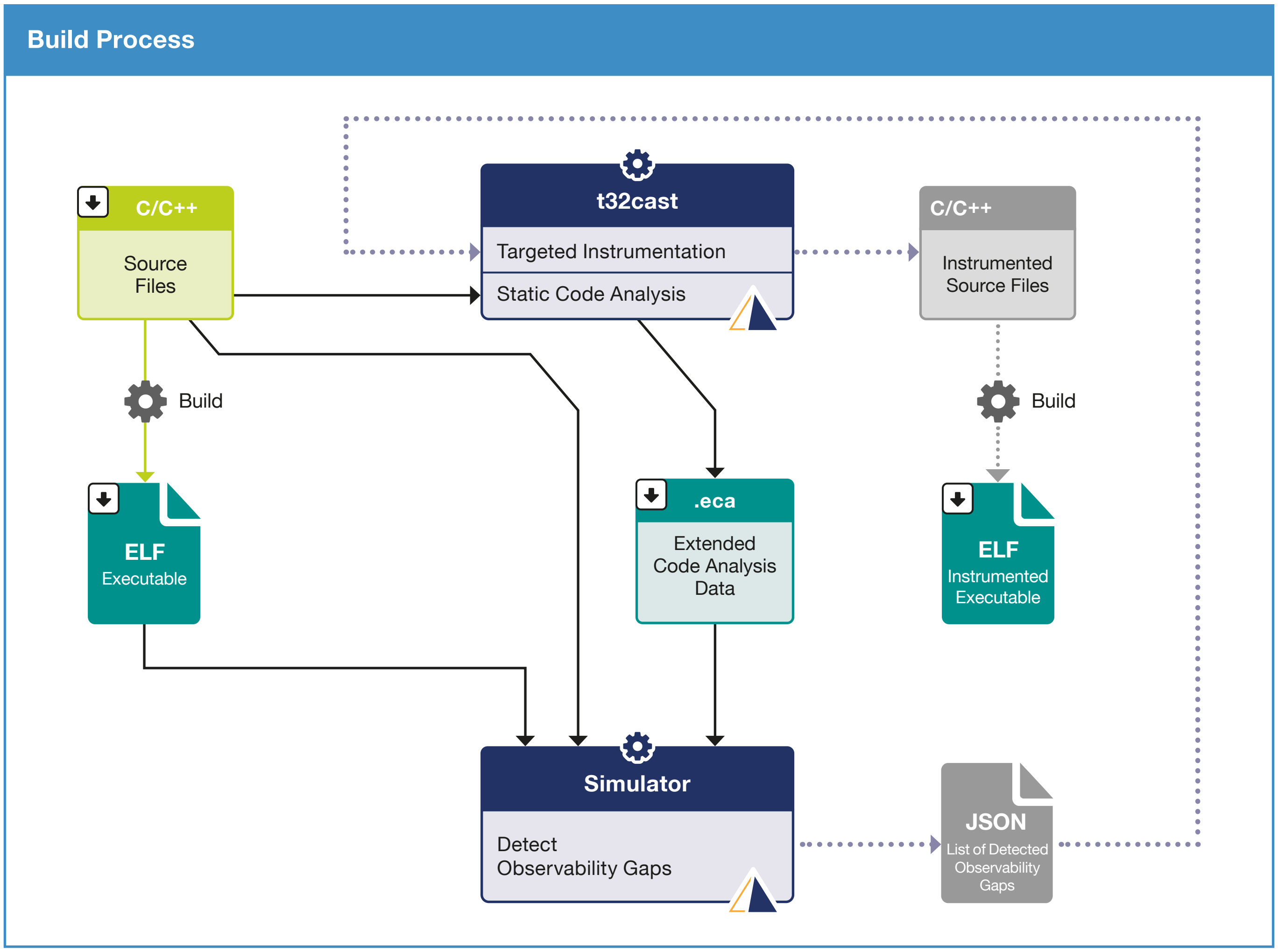


Figure 4: The build process for TRACE32 multi-mode code coverage.

What additional tasks do the two TRACE32 products handle in the build process?

1. t32cast analyzes the C/C++ source code. As a result of this static code analysis, an .eca file is generated for each source code file, which contains the required information about the decision structure.
2. The ELF file and the .eca files must be loaded into the TRACE32 Instruction Set Simulator to check for observability gaps. The result is saved in a JSON file.
3. If the JSON file is empty, the build process is complete.
4. If the JSON file is not empty, the source code must be instrumented at the locations where the observability gaps were detected. This is another task done with t32cast. The build process is completed here with the creation of an ELF file for the instrumented sources.

Before we proceed to the "Test & Report" step, a few words about the instrumentation performed by t32cast.

To ensure that TRACE32 can detect whether a condition evaluates true or false at any point in the recorded program flow, the source code is instrumented with the two instrumentation hooks t32__alpha() and t32__beta() for the detected observability gaps (see figure 6 on the last page). Both hooks are just function calls with an empty function body. TRACE32 evaluates these calls for MC/DC coverage analysis in addition to conditional branches/instructions.

The cost per instrumentation is very low because no additional interface to memory is required for TRACE32 targeted instrumentation. It is also convenient that no additional source code lines are generated, so the original source code can be used in the "Test & Report" step.

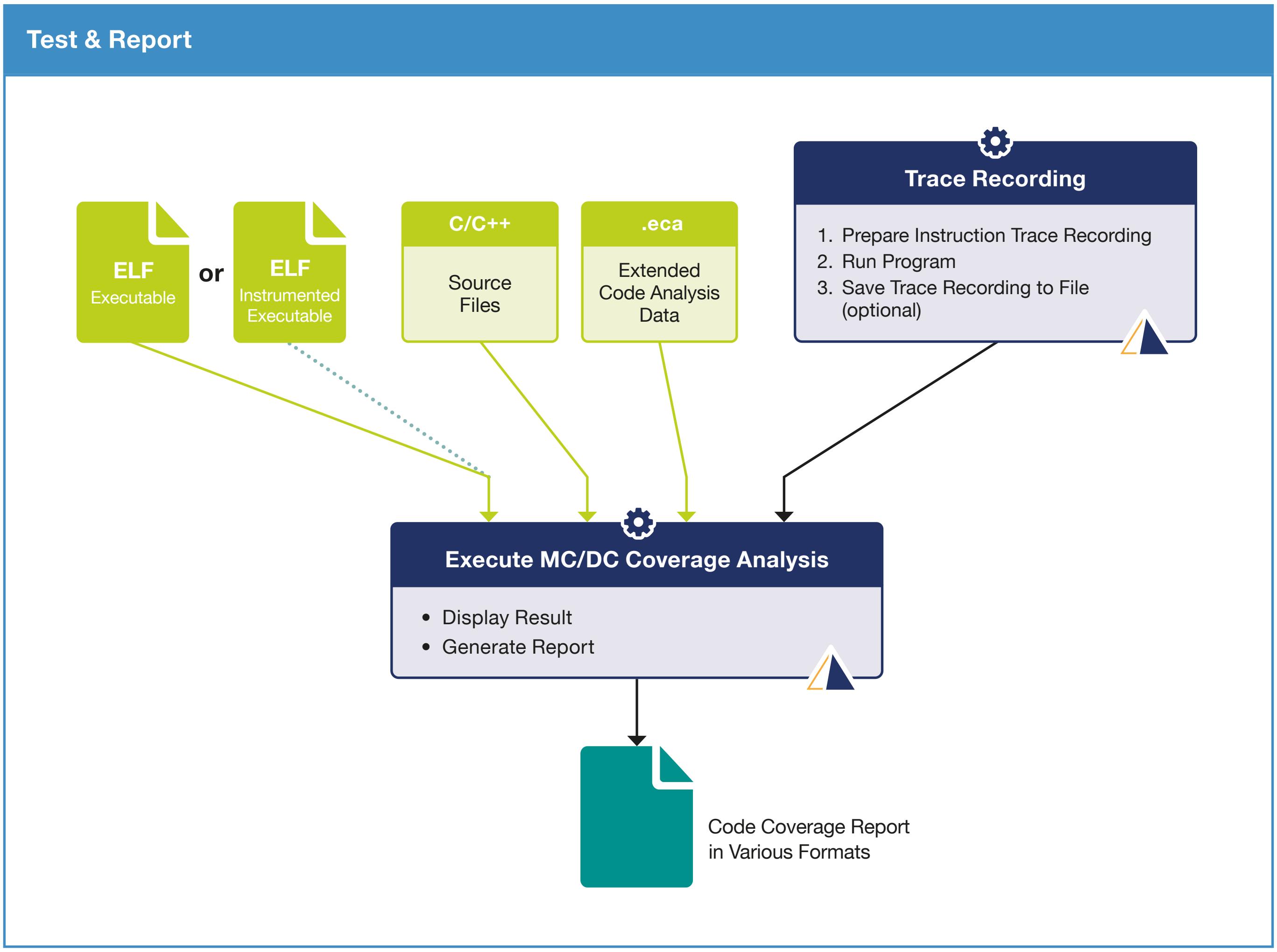

Test & Report
ELF
Executable
or
ELF
Instrumented Executable
C/C++
Source Files
.eca
Extended Code Analysis Data
Trace Recording
1. Prepare Instruction Trace Recording
2. Run Program
3. Save Trace Recording to File (optional)
Execute MC/DC Coverage Analysis
• Display Result
• Generate Report
Code Coverage Report in Various Formats

Figure 5: Test & Report for TRACE32 Multi-Mode Code Coverage.

Test & Report

For the test, the C/C++ sources, the corresponding .eca data and the corresponding ELF file must be loaded into the TRACE32 Debugger/Instruction Set Simulator, which executes the MC/DC coverage analysis (see figure 5). There are two options for trace recording:

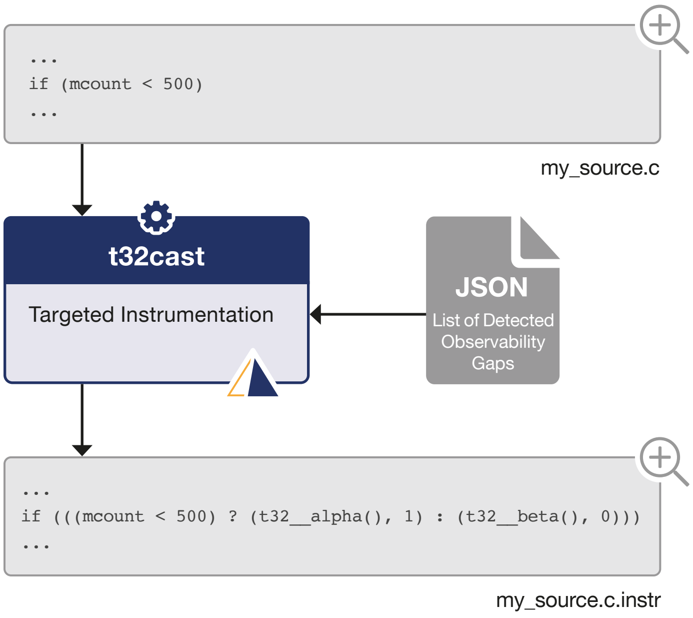
1. The trace recording and the execution of the MC/DC coverage analysis are performed as a sequential task. The MC/DC coverage analysis is executed directly for the trace data recorded in the debugger.
2. Trace recording and the execution of the MC/DC coverage analysis are separate tasks, performed by two different test teams. In this case, the trace data must be saved in a file after recording and reloaded into TRACE32 for the MC/DC coverage analysis at a later point in time.

Recommended Workflow for Safety-Related Projects

As described above, trace-based MC/DC requires reduced optimization and may even require some code instrumentation. It is crucial that the embedded software built specifically for the coverage test purpose behaves in exactly the same way as the production software that will ultimately control the embedded system. Therefore, it is necessary to test both software variants side by side for the entire test life cycle. Figure 7 on the last page shows the testing workflow recommended for safety related projects.

Outlook

TRACE32 Multi-Mode Code Coverage allows us to achieve our goal of supporting MC/DC coverage for a wide range of core architectures, trace protocols and compilers. In many cases, you will get by without instrumentation at all. The targeted instrumentation that may be necessary requires never more than 10% additional memory for the whole object code and has a minimal time overhead.



```
...
if (mcount < 500)
...
```

```
...
if (((mcount < 500) ? (t32__alpha(), 1) : (t32__beta(), 0)))
...
```

Figure 6: Instrumentation example for the multi-mode code coverage.

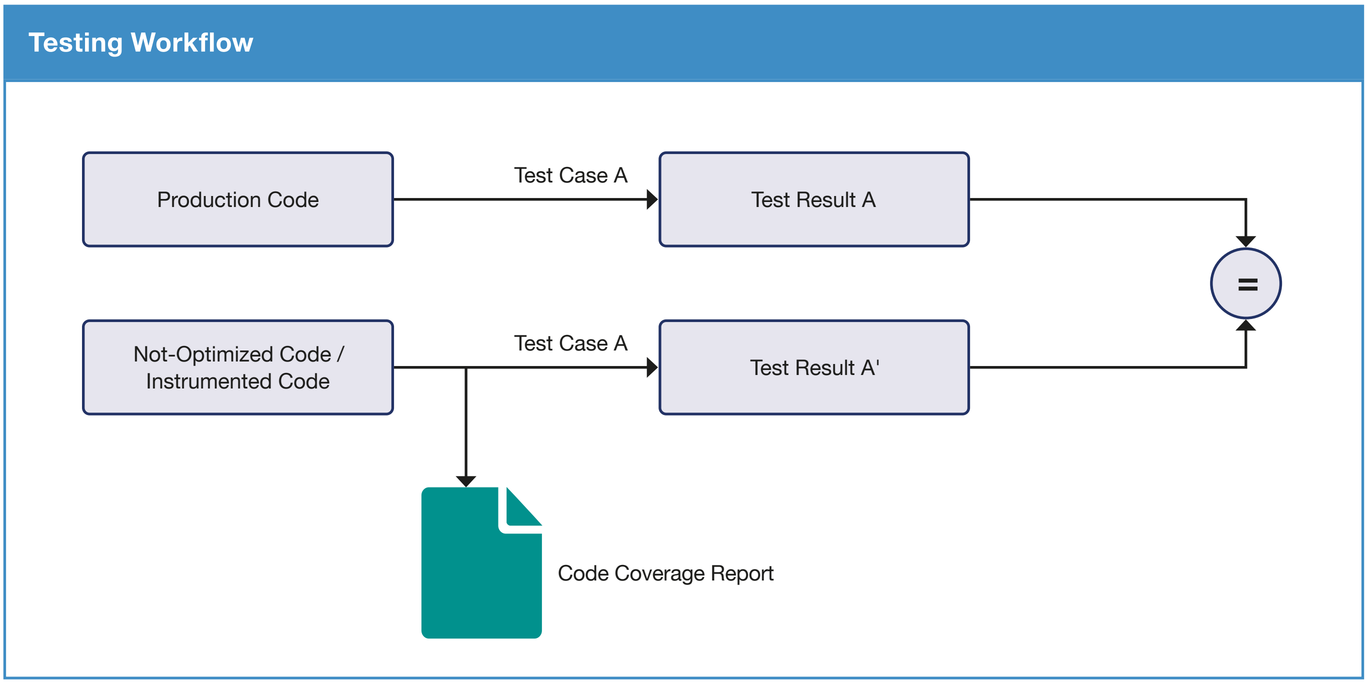


Figure 7: Testing workflow for safety-related projects.