2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)

Non-intrusive MC/DC Measurement based on Traces

Faustin Ahishakiye, Svetlana Jakšić, Volker Stolz
Department of Computing, Mathematics and Physics
Western Norway University of Applied Sciences
Bergen, Norway
firstname.lastname@hvl.no

Felix D. Lange, Malte Schmitz, Daniel Thoma
Institute for Software Engineering
and Programming Languages
University of Lübeck
Lübeck, Germany
lastname@isp.uni-luebeck.de

***Abstract*—We present a novel, non-intrusive approach to MC/DC coverage measurement using modern processor-based tracing facilities. Our approach does not require recompilation or instrumentation of the software under test.**

Instead, we use the Intel Processor Trace (Intel PT) facility present on modern Intel CPUs. Our tooling consists of the following parts: a frontend that detects so-called decisions (Boolean expressions) that are used in conditionals in C source code, a mapping from conditional jumps in the object code back to those decisions, and an analysis that computes satisfaction of the MC/DC coverage relation on those decisions from an execution trace. This analysis takes as input a stream of instruction addresses decoded from Intel PT trace data, which was recorded while running the software under test. We describe our architecture and discuss limitations and future work.

***Keywords*-Code coverage, MC/DC, Software testing, Software verification**

I. Introduction and Motivation

In order to prevent disastrous events, certification standards, for example the DO-178C [1] in the domain of avionic software systems, are used by certification authorities, like the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA), to approve safety-critical software and ensure that the software used in the systems follows certain software engineering standards. DO-178C requires that structural coverage analysis is performed during the verification process mainly as a completion criterion for the testing effort and to identify design faults as well as finding dead code.

Software with the highest safety level (Level A) in avionics systems is required to show modified condition decision coverage (MC/DC) [2]. Unlike weaker coverage criteria, MC/DC is sensitive to the complexity of decisions, because every condition in each decision has to show its independent effect on the decision's outcome.

Usually MC/DC is measured by instrumenting the source code (see III) in order to observe information about taken paths, executed statements and evaluated conditions. Instrumentation is intrusive (it may change characteristics like memory consumption, affect the cache and scheduling) and it is necessary for certification purposes to show that the behavior of the code does not change after the coverage is measured and the instrumentation is removed. Alternatively it is possible to leave the instrumentation in the release code but that consumes resources which are especially valuable in embedded systems, which are widely used in the domain of safety-critical systems.

We present an approach how MC/DC can be measured non-intrusively by analyzing program traces. Our novel approach is based on the idea that every condition in the source code is translated into a conditional jump on the object code level. We first record the trace of an executing program and then analyze it offline [3]. Program traces contain information about taken jumps during the execution and make it possible to reconstruct the evaluation of each condition without instrumentation.

The rest of this paper is organized as follows: Section II introduces coverage criteria of safety-critical software. An overview of state-of-the-art solutions is given in Section III. Section IV describes Intel Processor Tracing (Intel PT) and trace reconstruction. Section V explains the idea and the implementation of our tool. Section VI presents the experiment setup. Finally, we provide related work (Section VII) and concluding remarks and future work in Section VIII.

II. MC/DC in Context of Safety-Critically Software

Depending on the software safety-level, which is assessed by examining the effects of a failure in the system, different coverage criteria have to be fulfilled during software verification:

Software level C (major effect) requires *statement coverage* and software level B (hazardous effect) requires *decision coverage* [4]. Statement coverage is a relatively weak criterion, because it only requires that every statement has been executed but it is insensitive to control flow. Decision coverage is a fairly stronger criterion because it makes sure that every possible outcome of each decision (e.g. the Boolean expression in an if-then-else) has been executed at least once, and therefore there is no unexpected behavior caused by an unexpected outcome of a decision.

Software Level A (catastrophic effect) requires *modified condition/decision coverage* (MC/DC). The coverage criterion has been chosen as the coverage criterion for the highest

This work was supported in part by the European Horizon 2020 project COEMS under number 732016 and the BMBF project ARAMiS II with funding ID 01 IS 16025.

978-1-7281-3342-3/19/$31.00 ©2019 IEEE
DOI 10.1109/TASE.2019.00019

safety level software because it is sensitive to the complexity of the structure of each decision [2] – a decision is made up of one or more conditions. Compared to even stronger criteria like multiple condition coverage (MCC), that requires every possible combination of all conditions which leads to an exponential growth of the minimum numbers of test cases, MC/DC may be satisfied with only $n + 1$ test cases for a decision with n conditions. The following definition has been provided in the DO-178C [4]:

Definition 1 (Modified condition/decision coverage):

- Every point of entry and exit in the program has been invoked at least once,
- every condition in a decision in the program has taken all possible outcomes at least once,
- every decision in the program has taken all possible outcomes at least once, and
- each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

Additionally, the terms *Condition* and *Decision* are defined as:

Definition 2 (Condition): A Boolean expression containing no Boolean operators except for the unary operator (NOT).

Definition 3 (Decision): A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition.

For example, in Figure 1a the if-statement contains a decision `a<5||(b==5&&c>5)`. This decision is composed of three conditions `a<5`, `b==5` and `c>5`.

By showing the independent effect of each condition, MC/DC assures that condition's defined purpose. The most challenging and most discussed part in the definition of MC/DC is showing this independent effect: item (2) in the definition has been introduced in the DO-178C to clarify that the so called *Masked MC/DC* is allowed [1], [5]. Masked MC/DC means that it is sufficient to show the independent effect of a condition by holding fixed only those conditions that could influence the outcome. This is important for programming languages that use short-circuit evaluation, because certain executions of decisions are not distinguishable, if the outcome of the decision is determined before every condition has been evaluated.

III. State-of-the-Art

Today there is a number of testing tools for measuring coverage developed for both industrial use and academic purpose. For instance, a survey conducted in [6] described and compared 17 tools primarily focusing on, but not restricted to, coverage measurement. These tools are focusing on weaker coverage criteria for C, C++ and Java programs. Most of them are used only for code coverage, but some, such as Agitar, Dynamic, JCover, Jtest and Semantic Designs, provide debugging assistance as well.

Currently, there are not so many testing tools which focus on MC/DC measurement. VectorCAST/MCDC [7] is a well established tool for measuring MC/DC coverage for C/C++. The tool supports both unique cause and masking MC/DC analysis. Beside reporting and documenting the results, the tool supports automatic test case generation to quicken the development of a full set of MC/DC test cases. Parasoft C++test [8] is a C/C++ testing tool that is capable of measuring MC/DC. MC/DC is evaluated by calculating the ratio of the number of conditions with independence effect and the total number of conditions in all decisions. Testwell CTC++ tool [9] measures line, statement, function, decision, multiple condition, MC/DC and condition coverage for C, C++, Java, and C# on target and on host. The generated report is showing coverage percentage. CodeCover [10] is an open-source, white-box testing tool developed at the University of Stuttgart. It implements the Ludewig term coverage and they claim that it is similar to MC/DC (subsumes MC/DC).

RapiCover [11] analyzes code coverage including MC/DC on-host and on-target. They visualize coverage by folder, file, function and test case, and filter results to highlight missing coverage. All these tools measure MC/DC intrusively by instrumenting the source code..

In [12], SmartUnit tool which supports statement, branch, boundary value and MC/DC coverage is described. They aim at the unit coverage-based testing and automatically generating MC/DC coverage test cases in industry environment. The percentage of MC/DC coverage is calculated as the ratio of covered conditions and the total conditions in the source code. The commercial Lauterbach tool [13] (see Sec. VII) uses a dedicated hardware-interface to transfer tracing data from the system-under-test into the developer's machine for analysis. They support a variety of trace sources, among others also Intel PT, and use it to measure MC/DC in a similar manner as we describe below.

Another alternative, non-intrusive approach is running the system-under-test within an emulator. The QEMU emulator has been used to this end within the Adacore community [14], and in the RTEMS operating system [15]. Through the emulator it is easy to observe the execution of a program on the object code level, very much like through Intel PT that we will present below. The obvious threat to validity is of course how closely the emulator setup can reflect the real system, especially when considering certification.

In the following we propose a novel approach how to measure MC/DC without instrumentation and a tool that implements this approach on a live system without the need for additional hardware.

IV. Trace-based Approach

The main idea of our trace-based approach is that each condition in the source code is translated into a single conditional jump in the object code. If we can accurately trace execution, we will be able to reconstruct the evaluation of conditions

along the execution paths. On modern Intel processors, through the IntelPT framework, we are able to unobtrusively record the execution traces of applications. Through operating system support, tracing can be easily enabled for a single application. We first describe the general mode of operation of IntelPT, and continue then with our analysis of the recorded traces.

A. Intel Processor Tracing (Intel PT)

Program tracing is an important mechanism for developers in the context of gathering useful information for debugging, monitoring and performance analysis of a program executions.

Intel Processor Tracing (Intel PT) is an extension of the Intel Architecture that traces program execution with low overhead [16]. It can be used by modern Intel CPUs such as Intel Broadwell (5th generation) CPU or better. Intel PT was introduced to provide an accurate and detailed trace with triggering and filtering capabilities [17]. Intel PT works by capturing information about software execution on each hardware thread using dedicated hardware facilities so that after execution completes software can do processing of the captured trace data and reconstruct the exact program flow.

Intel PT uses an extremely compact format that makes it possible to overcome the small bandwidth and limited buffer space by basically only storing information about taken and not taken branches, indirect branches, function returns and interrupts. Based on these the complete program execution flow can be reconstructed. With Intel PT, it is easy to extract and report a much deeper view on loop behavior, from entry and exit down to specific back-edges and loop trip-counts. The traces contain instructions executed by the processor, but there are no data values. For example, for the C-level instruction `x = y + z`, as the trace essentially only consists of instruction pointers, we can only reconstruct the assembly instructions for loading the values, summation and storing the result in memory, and maybe even map them onto the source-code, but we have no information about the actual values of x, y and z or their location in memory during execution.

IntelPT has some drawbacks related to trace file size and speed since the trace bandwidth can exceed 10 Gbits/s. That means that the program trace data and the decoded trace become huge, fast. Recording program executions of more than a few milliseconds requires large and fast writable storage, so that information can be stored quickly enough for offline- or parallel processing, without losing events due to full buffers.

B. Trace Analysis

By analyzing program traces it is possible to see if the jumps corresponding to conditionals in the source code have been taken during the execution and to reconstruct how the conditions have been evaluated. If the statement following the conditional jump in the trace equals the target of this jump, the jump has been taken.

Which conditional jumps occur in the object code depends on the condition in the source code. Because the compiler sometimes uses the negation of the operator, there are two assembly instruction possible for each relational operator.

Nr.	A	B	C	$A \vee (B \wedge C)$
1	**false**	**false**	?	**false**
2	**false**	**true**	**false**	**false**
3	**false**	**true**	**true**	**true**
4	**true**	?	?	**true**

TABLE I: Short-circuit evaluation for $A \vee (B \wedge C)$

```
1  if (a<5 || (b==5 && c>5)){
2    return 1;
3  }
```

(a) C code with decision containing three conditions.

```
1  400494: cmpl   $0x5,-0x8(%rbp)
2  400498: jl     4004b2
3  40049e: cmpl   $0x5,-0xc(%rbp)
4  4004a2: jne    4004be
5  4004a8: cmpl   $0x5,-0x10(%rbp)
6  4004ac: jle    4004be
7  4004b2: movl   $0x1,-0x4(%rbp)
```

(b) Object code with three conditional jumps.

Fig. 1: C code and corresponding Object code compiled with clang version 5.0 on default parameters.

From tracing execution in the object code, we can then reconstruct the outcome of an entire decision by analyzing the trace. If the decision statement is followed by the instructions corresponding to the then-branch, the decision has been evaluated as True, otherwise False. Figures 1a and 1b show a decision as part of a C program with three conditions, and the corresponding assembly code (with compiler optimizations disabled). The comparisons (`<`, `>` and `==`) are translated by the compiler into small sequences of assembly instructions. Typically these consist of a compare operator (`cmpl`) and a conditional jump (`jl, jne`). This structure makes it possible to map a conditional jump to each condition.

C. Short Circuit Evaluation

In C (as in most modern programming languages) short-circuit evaluation is used to evaluate Boolean expressions. That means that the expression is evaluated from left to right and if the left-hand operand of a conjunction is **false** or, respectively, if the left-hand operand of a disjunction is **true**, the right-hand operand is not further evaluated. As mentioned in Section II, Masked MC/DC is accepted by the DO-178C. Because short-circuit evaluation skips exactly those conditions that cannot influence the outcome of a decision, it is possible to measure Masked MC/DC based on traces.

D. Condition Reconstruction

The decoded program trace contains information about each executed instruction and therefore whether each jump has been taken or not. This makes it possible to look at each execution of a decision and to note which jumps have been taken. A table can be generated where each row contains one evaluation of a decision and each column contains the assignment of each condition during that evaluation. The last column shows

Relational Operator:	Possible Conditional Jumps:		Condition Value of Detected Jump:
	x86-64	ARM	
no operator	`jne`	`bne`	**True**
	`je`	`beq`	**False**
==	`je`	`beq`	**True**
	`jne`	`bne`	**False**
<	`jl`	`blt`	**True**
	`jge`	`bge`	**False**
<=	`jle`	`ble`	**True**
	`jg`	`bgt`	**False**
>	`jg`	`bgt`	**True**
	`jle`	`ble`	**False**
>=	`jge`	`bge`	**True**
	`jl`	`blt`	**False**

TABLE II: Multiple interpretations of jumps in the x86-64 and ARM instructions sets compiled with clang version 5.0

the outcome of the decision during the execution. The table has n rows and $m + 1$ columns for a decision that has been executed n times and has m conditions. Table I shows the table for the decision $A \vee (B \wedge C)$ with some example observations/outcomes that satisfy MC/DC (see explanation below).

Because of short-circuit evaluation not all conditions are generally evaluated during one execution and can therefore not be reconstructed by analyzing the trace. In the table these entries are filled as "?".

Depending on the relational operator in the condition (<, <=, ==, etc.) two different possible conditional jumps can be generated by the compiler because conditions can be translated to their negation (it is up to the compiler to choose "jump-if-equal" or "jump-if-not-equal"). If a condition is translated as its negation, this has implications for the reconstruction of the assignments by analyzing the trace as a taken jump shows that the condition has been evaluated as **false**. The possible combinations for the Intel x86-64 instruction set and its ARM counterpart are shown in Table II, which have to be taken into account when the reconstruction is performed. We call the addresses of instructions relevant to our analysis *watch-points* (i.e., conditional jumps and their targets).

E. MC/DC Measurement

After we have recorded all reconstructed condition values in a table per decision, MC/DC can be measured as follows. All rows with a different outcome are compared. If they contain a different entry for exactly one condition, these two assignments show the independent effect for this condition. Two entries for a condition are considered different if one contains a **true** and another one contains **false**. If one of them contains the unknown reconstruction **"?"**, the independent effect of this condition cannot be shown based on these cases.

For the example in Table I with short-circuit evaluation, the independent effect of condition A can only be shown by ignoring the other two conditions, because they cannot influence the outcome, if A is **true**. So executions number 1 and 4 show the independent effect of condition A. The independent effect of B can only be shown, if A is **false** and therefore B is actually evaluated. The outcome and value of B changes in this example in executions number 1 and 3. Likewise, the independent effect of condition C can be shown with executions number 2 and 3, because the value of condition C and the outcome are changing.

This corresponds as Masked MC/DC and hence complies with the definition of MC/DC in the DO-178C.

We define the measured coverage as the ratio of all decisions satisfying MC/DC and the number of all decisions in the source code.

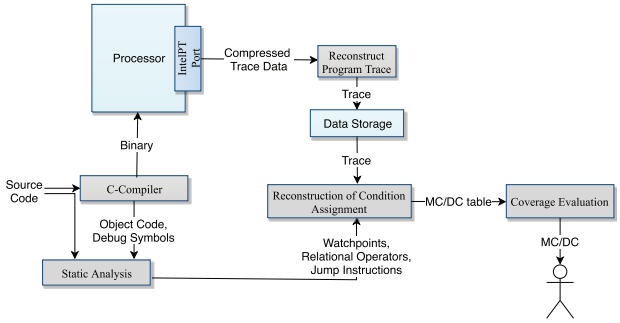

Fig. 2: Overview of the implementation.

V. Implementation

An overview of the implementation is provided in Figure 2. The source code is analyzed by our tool in order to detect decisions and their conditions. Additionally, we extract information about their corresponding conditional jumps from the object code. With this information and the program trace provided by Intel PT it is possible to reconstruct the condition assignments and measure MC/DC.

A. Decision Detection with LLVM

In the first step, decisions in the source code have to be found. In order to find decisions in the source code we use the *Abstract Syntax Tree* (AST) representation provided by LLVM. With *LibTooling* and the *AST-matcher* [18] we have built a tool that detects all if-, for- and while-statements in the source code and we gather corresponding information such as line and column numbers and then-statements. We focus on finding traditional branch points (if-, while-, for-statements), but we are aware that certification authorities require other structures, for example assignments containing Boolean expressions, to be covered as well [19].

B. Mapping with Debug Symbols

After the decisions and their conditions in the source code have been detected, debug-symbols are used to map the conditions to conditional jumps in the object code.

The direct mapping is possible by utilizing debug symbols provided by the compiler. We use *clang 5.0* because this compiler provides rich debug symbols containing line and column information with the compiler option `-g -XClang`

`-dwarf-column-info`. Combined with the detected decisions from the LLVM-tool we then can detect all conditional jumps that are needed for measuring MC/DC based on traces.

Because the outcome of the decision during the execution has to be reconstructed as well, it is necessary to find the *then-statement* which is the statement executed in case of a decision being evaluated as **true**. This statement is also mapped using debug-symbols to its corresponding instruction in the object code.

The result are the decisions, conditions and then-statements in the source code and their translation in the object code.

C. Program Trace Generation

We use Intel Processor Trace (Intel PT) to generate a trace of the execution of a program. The technology is widely available, which makes it suitable for this proof-of-concept tool. With *perf*[1] the Linux-kernel provides an easy-to-use implementation of the recording and reconstruction of Intel PT traces. The reconstructed traces become quiet large even for short execution times. To reduce the size, we filter the trace against the watch-points and only store those parts of the trace that are relevant for measuring MC/DC.

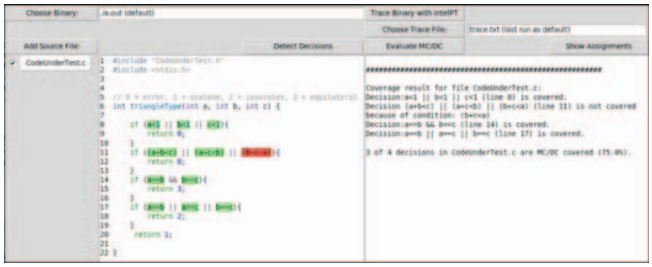

Fig. 3: Screenshot of the GUI.

D. Graphical User Interface

The tool chain of detecting all decisions, mapping conditions to conditional jumps, running and tracing the program and measuring MC/DC based on the trace can be used via a graphical user interface (GUI, see Figure 3) or through the command line as described in Section VI. Via the GUI, we show the detected decisions and the measured coverage in the source code, allowing the user to directly see which conditions are not covered. This should help developers in finding new test cases that cover the missing combinations.

A typical workflow with our tool is the following:

1) *Choose Binary* opens a file dialog and the binary can be selected.
2) *Add Source File* opens a file dialog and source files can be added for which MC/DC should be measured.
3) The source files are listed and can be viewed by clicking on them.
4) *Detect Decisions* detects the decisions in the selected source code and maps the conditions to their corresponding conditional jumps.

[1]https://perf.wiki.kernel.org

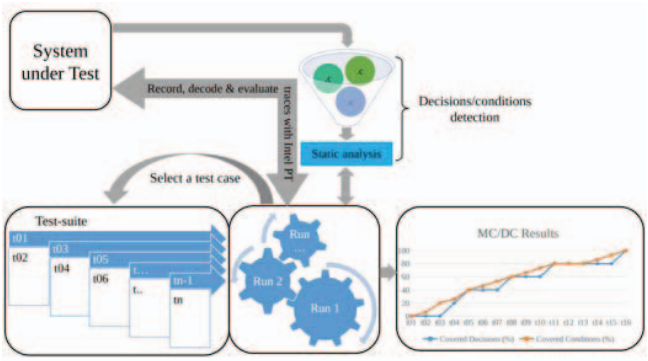

Fig. 4: MC/DC measurement experiment setup.

5) *Trace Binary with Intel PT* calls Linux' *perf* and saves the trace in the file chosen in *Choose Trace File*.
6) *Evaluate MC/DC* analyzes the recorded trace and measures MC/DC of the detected decisions. The result is shown directly in the source code.
7) *Show Assignments* opens a new window containing an overview of all detected decisions, conditions and their reconstructed values.

The tool is available for academic evaluation purposes[2]. On the website you can find an example application and a trace recorded with Intel PT, which can be analyzed with the tool.

VI. Experimental Setup

Our experimental setup for MC/DC coverage measurement consists of two examples as C code, together with their unit tests. The function in the first unit has four decisions (if-statements), containing in total eleven conditions. The second unit contains one decision (also an if-statement) with three conditions. The entire test suite contains 16 test cases. Our tooling allows us to execute the entire test suite and measure coverage, or to just run and measure a single test case. The test suite contains twelve test cases for the first unit where MC/DC coverage is achieved with eleven test cases. Note that this is not directly related to the number of $n+1$ test cases before, as the decisions in subsequent if-statements are not independent. The second unit has four test cases, and all four test cases need to be executed to achieve MC/DC coverage. In addition to the use of our tool via GUI as described in Section V, in this section we set up our experiment for MC/DC measurement via the command line on a Linux OS. After the compilation of the program under test, we conduct the experiment in the following steps as shown in Figure 4:

First, we conduct a static analysis in order to find out which conditions in the source code correspond to which conditional jumps in the object code. The static analysis results in a JSON file with all information related to decisions and conditions and their location (line and column), as well as their mapping to the object code. That is, conditions are mapped to addresses and conditional jumps in the object code. This mapping is

[2]https://www.coems.eu/mc-dc/

necessary because MC/DC is a criterion that is defined on the source code level and there are no equivalent metrics defined on the object code level. In other words, we ignore conditional jumps in the object code that do not directly come from conditionals in the source code. With this information, it is possible to reconstruct the assignment of the conditions during an execution by analyzing the performed jumps and inferring if a condition has been evaluated as true or false. If the program address following a jump instruction in the trace equals the target address that is recorded in the conditional jump instruction, that jump has been performed, otherwise it has not. We use this information to reconstruct the assignment of the condition.

Secondly, we created a wrapper that allows to easily run one particular test from the command line. For each particular test, we record and decode the trace using Intel PT, and we incrementally evaluate the trace with respect to previous results, measure MC/DC and query the MC/DC results. We track the percentage of MC/DC coverage that is achieved through the incremental runs. The tool iterates randomly through the test cases, selecting one at a time and it stops once 100% MC/DC coverage is achieved, otherwise it continues picking other test cases, i.e., we run a test case at most once.

Finally, the tool reports the MC/DC result with the set of test cases that have been executed. From the recorded data, it is easy to plot curves as to which test case contributes to decision or condition coverage. Note that we are not replacing the unit tests, but rather see this as a way to minimize testing overhead: in practice, one would suggest a run of all unit tests without measuring MC/DC, and having occasional runs with tracing enabled that verify that a known set of test cases achieves a predetermined threshold of MC/DC coverage.

VII. Related Work

The interesting discussion on the applicability of MC/DC to software testing for safety-critical systems have been introduced by Chilenski in [2]. Different comparisons for code coverage metrics have been investigated in the context of structure based metrics [20], data-flow metrics [21], decision coverage and MC/DC [22], comparison of multiple condition coverage (MCC) and MC/DC with short-circuit evaluation [23]. MC/DC and object branch coverage (OBC) criteria were compared in [24] and [25]. Even though aforementioned research gives the foundation, none provides a deep MC/DC analysis based on the trace of the program-under-test.

A non-intrusive online monitoring for multi-core systems based on the embedded trace of the system under test is proposed in [26]. Online reconstruction and analysis of debug trace data are based on FPGA and TeSSLa [27]. This combination could be used to implement coverage-calculation on the FPGA, instead of doing it on the host or offline, as in our setting here.

Lauterbach offers the `t32cast` command line tool for MC/DC measurement based on a real-time trace recording, which can analyze the C/C++ source code [13]. The user must ensure that the selected compiler translates each condition in the source code into a conditional jump at the object code level, e.g. by disabling optimizations. In contrast to our approach, which uses features present in most modern Intel processors, the trace data are transferred through a dedicated hardware-connection to a monitor.

VIII. Conclusion

We present a tool that shows the feasibility of measuring MC/DC without instrumentation based on program traces. The tool is able to detect decisions and conditions in C source code and to find their corresponding conditional jumps in the object code. MC/DC can be measured by reconstructing condition assignments based on Intel PT traces.

The advantage of our approach is that there is no need for intrusive software instrumentation. Traditionally, the coverage of the instrumented code is measured, and the instrumentation has to be removed before release, but with our approach it is possible to measure coverage directly on the release code by only using debug symbols that are not altering the behavior of the code and therefore are not considered intrusive.

Our approach of measuring MC/DC based on traces complies with the position of CAST-17, that provides certification authorities' concerns and position regarding the analysis of structural coverage at the object code level [28]. With the mapping between conditions and conditional jumps we provide traceability between source and object code and the reconstruction of condition assignments on the source code level, we can provide the same level of assurance as measuring directly on source code level via software instrumentation.

However there are some limitations. It is necessary to disable optimizations during the compilation because even on low optimization levels conditions are usually not directly translated into conditional jumps but into conditional moves, jump tables or indirect branches [29]. Because regular program traces contain no information how these instructions are evaluated, they cannot be used to reconstruct the evaluation of conditions. This limitation is less severe in the domain of avionic, because other requirements, for example source code to object code traceability in DO-178C, make it already hard for developers to use high optimization levels [30]. Also it is necessary to have a modern compiler like clang version ≥ 5.0 because the DWARF debug symbols need to have column and line information.

Another problem of our approach is that the trace data becomes excessively large for longer executions. Here, we used an offline tracing approach [3], where available storage effectively limits the size of traces. In future work, we want to apply this approach to online trace reconstruction which would enable us to observe much longer execution times because only the very events that are used for coverage measurement are reconstructed. We also want to support other architectures and instead of Intel PT, use tracing technologies such as ARM CoreSight and NEXUS for PowerPC since these processor architectures are widely used in avionics and automotive industry, which would benefit the most from this new approach of MC/DC measurement.

REFERENCES

[1] F. Pothon, "DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements," AdaCore, Tech. Rep., 2012, available at https://www.adacore.com/books/do-178c-vs-do-178b.

[2] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[3] F. D. Lange, "Modified Condition/Decision Coverage based on jumps," 2018, master's thesis, available at http://www.isp.uni-luebeck.de/thesis/modified-conditiondecision-coverage-based-jumps.

[4] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[5] Certification Authorities Software Team (CAST), "Rationale for Accepting Masking MC/DC in Certification Projects," *Technical Report: Position Paper CAST-6*, 2001.

[6] Q. Yang, J. J. Li, and D. Weiss, "A Survey of Coverage Based Testing Tools," in *Proc. of the 2006 Intl. Workshop on Automation of Software Test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 99–103.

[7] Vector Software, "Vectorcast/mcdc," available at https://www.vectorcast.com/software-testing-products/embedded-mcdc-unit-testing.

[8] A. Trujillo and A. Stuchlik, "Reviewing coverage information," Parasoft C++test documentation, available at https://docs.parasoft.com/display/CPPDESKE1033/Reviewing+Coverage+Information.

[9] Testwell, "Testwell CTC++: Test Coverage Analyzer for C/C++," available at http://www.testwell.fi/ctcdesc.html.

[10] CodeCover, "CodeCover: an open-source glass-box testing tool," available at http://codecover.org/.

[11] Rapita Systems, "RapiCover: Low-overhead coverage analysis for critical software," available at https://www.rapitasystems.com/products/rapicover.

[12] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, and G. Pu, "Smartunit: Empirical evaluations for automated unit testing of embedded software in industry," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183554

[13] Lauterbach, "Trace-based MCDC Coverage," 2018, available at https://www.lauterbach.com/new2018_cov_mcdc.pdf.

[14] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, "Object and source coverage for critical applications with the COUVERTURE open analysis framework," in *Proc. of Embedded Real Time Software and Systems Conference (ERTS)*, 2010. [Online]. Available: http://www.open-do.org/wp-content/uploads/2010/06/couverture_ertss2010.pdf

[15] H. Felbinger, J. Sherrill, G. Bloom, and F. Wotawa, "Test suite coverage measurement and reporting for testing an operating system without instrumentation," in *17th Real-Time Linux Workshop*, 10 2015. [Online]. Available: https://gedare.github.io/pdf/FelShe15A.pdf

[16] A. Kleen, "Cheat sheet for Intel Processor Trace with Linux perf and gdb," April 2017, available at http://halobates.de/blog/p/410.

[17] J. Thalheim, P. Bhatotia, and C. Fetzer, "INSPECTOR: Data Provenance Using Intel Processor Trace (PT)," in *2016 IEEE 36th Intl. Conf. on Distributed Computing Systems (ICDCS)*, June 2016, pp. 25–34.

[18] The Clang Team, "Matching the Clang AST," Clang documentation, available at https://clang.llvm.org/docs/LibASTMatchers.html.

[19] Certification Authorities Software Team (CAST), "What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?" *Technical Report: Position Paper CAST-10*, 2002.

[20] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Transaction on Software Engineering*, vol. 14, no. 6, pp. 868–874, 1988.

[21] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Comparison of Data Flow Path Selection Criteria," in *Proc. of the 8th Intl. Conf. on Software Engineering*, ser. ICSE '85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 244–251.

[22] K. Kapoor and J. Bowen, "Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria," in *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, Sept 2003, pp. 185–194.

[23] S. Kandl and S. Chandrashekar, "Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation," *Computing*, vol. 97, no. 30, pp. 261–279, Mar 2015.

[24] C. Comar, J. Guitton, O. Hainque, and T. Quinot, "Formalization and Comparison of MCDC and Object Branch Coverage Criteria," in *Proc. of Embedded Real Time Software and Systems Conference (ERTS)*, 2012.

[25] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. Heimdahl, "Toward rigorous object-code coverage criteria," Technical Report, University of Minnesota, MN, USA, Tech. Rep., 2017.

[26] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss, "Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems," in *Formal Methods: Foundations and Applications*, S. Cavalheiro and J. Fiadeiro, Eds. Springer, 2017, pp. 179–196.

[27] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss, "Online analysis of debug trace data for embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 851–856.

[28] Certification Authorities Software Team (CAST), "Structural Coverage of Object Code," *Technical Report: Position Paper CAST-17*, 2003.

[29] Free Software Foundation, "Options That Control Optimization," GCC documentation, available at https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Optimize-Options.html.

[30] Certification Authorities Software Team (CAST), "Guidelines for approving source code to object code traceability, position paper 12," Certification Authorities Software Team, Tech. Rep., 2003.