Achieving Complete Structural Test Coverage in Embedded Systems Using Trace-Based Monitoring

Alexander Weiss[1]
Accemic Technologies GmbH, Kiefersfelden, Germany

Albert Schulz
Accemic Technologies GmbH, Kiefersfelden, Germany

Martin Heininger
Heicon Global Engineering GmbH, Schwendi, Germany

Martin Sachenbacher
Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Germany

Martin Leucker
Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Germany

Abstract

This paper presents a systematic approach to achieving, in a well-defined sense, 100% structural test coverage for large embedded software projects. In embedded systems, high code coverage is a critical part of the testing process to ensure that the system works correctly. Measuring code coverage provides insight into the effectiveness of the testing process, the quality of the software, and can help identify untested or partially tested areas of the code. Traditionally, coverage is often measured when unit tests are executed. The proposed approach instead uses integration tests as the starting point for determining test completeness. Measuring code coverage at the integration test level in embedded systems can be challenging due to the limitations of software instrumentation (additional memory requirements and additional CPU load). To overcome these limitations, embedded trace technology is used to measure code coverage continuously and non-intrusively. The use of these techniques will help to increase the reliability of embedded software and reduce the likelihood of missed integration tests, missed high-level requirements, and undetected software defects.

2012 ACM Subject Classification Hardware → Reconfigurable logic applications; Hardware → Coverage metrics; Software and its engineering → Software testing and debugging

Keywords and phrases structural tests, integration tests, code coverage, embedded trace

Digital Object Identifier 10.4230/OASIcs.DX.2024.19

Category Short Paper

Funding TRISTAN received funding from the EU Key Digital Technologies Joint Undertaking (KDT JU) under project ID 101095947, and the German Federal Ministry of Education and Research (BMBF) under project ID 16MEE0273. Martin Sachenbacher has been partially funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) under grant number 01MC22005C.

1 Introduction

A crucial phase in software development is requirements management, which includes the gathering, analysis, modification, and especially the refinement of requirements. Requirements management is a complex task, a distinct research field in computer science, and many projects fail due to unclear and imprecise requirements. Requirements define what the system

[1] corresponding author

to be built should do. Therefore, it is essential to check whether the resulting system meets its requirements. A widespread method for verifying this is testing. The goal here is to use appropriate test cases to verify that the developed system meets its requirements.

Another phase in software development is the so-called high-level design of the application, in which the overall system architecture is defined. The high-level design is refined into a low-level design, where individual units, known as "units", and their interactions within the system to be developed are specified. In the V-model, which is a process model describing the chronological sequence of various phases [20, Appendix B], the requirements and design phases are located on the left side of the "V". The individual units are tested and gradually integrated into larger units until the complete system is formed. The idea of the V-model is to thoroughly test each unit and then verify the correctness of component integration. This testing phase is located on the right side of the "V".

As mentioned above, the requirements represent exactly what the system should accomplish. Therefore, it is usual and standard practice to develop corresponding test cases for each individual requirement, which demonstrate in the overall system that the requirements are met. It is possible that a test case may check aspects of various requirements, so an $n : m$ relationship between requirements and test cases is established.

From a maintenance perspective, it is important that the implemented system does not perform more than what is required by the specifications. A well-established approach in practice is to measure how well test cases cover the system's code. If code exists for which no test cases are present, this is a sign that the test suite is incomplete – either requirements have been implemented that are not covered by test cases, or unnecessary functionality has been implemented. Therefore, it is desirable for 100% of the code to be covered by test cases. Code coverage is defined as the degree (in percent) to which the source code of a program is executed/accessed when a particular test suite is run. Coverage can be measured using different criteria such as statement coverage, branch coverage, condition coverage, MCDC coverage, etc. These criteria, simply put, require varying levels of thoroughness and should be used depending on the criticality (risk) of the system being tested. High code coverage of the test suite reduces the chance that the program contains undetected software defects and is an indication of test completeness. Achieving high code coverage is therefore important for rigorous system testing, and standards often specify which coverage criteria must be met based on the risk of the subsystem.

The requirements and their corresponding test cases must match the granularity of the subsystem being tested. Requirements for the entire system result in test cases for the entire system, known as system tests. Requirements for larger subsystems lead to tests that check the integrated subsystems, called integration tests. Requirements for individual units result in unit tests. Since a fully integrated system and corresponding integration tests check the entire system, integration tests are of particular importance for verifying the whole system. Therefore, conducting integration tests is crucial.

In practice, the focus is however often on testing individual units. These can generally be tested well, i.e., with high coverage and efficiency. It is then assumed that testing the entire system can be done with less effort due to the correctness of the individual units. Consequently, integration tests are performed only to a limited extent. However, especially for embedded systems, the behavior of integrated systems is not fully defined by individual units, as aspects like timing, memory usage, performance, concurrency, etc., only become challenges in the integrated system. There are in fact several advantages of measuring test completeness at the integration test level compared to the unit test level, particularly in the domain of embedded and safety-critical software. For example, integration tests replicate

real-world scenarios by combining multiple software components. This approach provides a more accurate representation of how the software will function in its intended environment, which is crucial for safety-critical systems. Moreover, integration tests cover interactions and interfaces among different software modules, ensuring that the entire system functions cohesively. This is especially important in embedded systems where multiple components must work together seamlessly. These tests help identify potential problems that may not surface in isolated unit tests, contributing to higher defect detection and risk mitigation. A vivid example of this is the Ariane V spacecraft, which was destroyed during its maiden flight after a deviation from the required flight path was detected. The cause was identified as a well-tested function that had worked successfully in the predecessor Ariane IV. Due to the changes in the overall system, however, this functionality was no longer adequate, leading to the catastrophic failure. This is another reason why integration tests should receive greater importance in practice than they currently do. Overall, integration tests play a critical role in validating safety-critical software, ensuring regulatory compliance, and evaluating complex use cases while providing insight into system performance.

Testing a fully integrated system and measuring coverage on an embedded system, however, is a major challenge. Usually this requires a test system on the target platform and software instrumentation, i.e. additional code that is executed alongside the actual system on the target platform and provides the necessary information for analysis; an example is gcov [11]. However, for embedded systems integration tests often cannot be conducted because the resources of the target platform are only sufficient for the actual application and not for both the application and the test system.

The instrumentation and simultaneous execution of the test system and the actual application distorts the behavior of the application to some extent, as additional code (even if it only outputs variables) affects the behavior of the system. This so-called *probe effect* is particularly problematic when analyzing timing behavior and concurrency-related errors such as race conditions on multi-core platforms. It will potentially cause resource-critical requirements (e.g., timing, memory) to be mistakenly marked as fulfilled or unfulfilled by test cases. The modification thus limits the measurement of code coverage during the execution of integration tests, restricting the meaningful measurement of code coverage to the execution of unit tests only.

In this paper, a non-invasive monitoring method for application execution is presented, which is based on recent technological advances [18, 1] that allow the monitoring and analysis part to essentially run on additional specialized hardware (FPGAs), so that the application itself is not affected. This ensures that integration tests can be conducted without running a test system on the target hardware, and the test results apply to the actual runtime, allowing code coverage to be measured during the execution of integration tests without additional instrumentation. We build on this technical solution for continuous and non-intrusive monitoring of processors to propose a systematic approach for determining and achieving test completeness in terms of code coverage at the integration level. In practice, achieving complete coverage of code, even with the simplest coverage criterion, statement coverage (where every single line of code must be executed once), is usually not feasible for several reasons laid out in the next section. Therefore, achieving 100% code coverage requires either an appropriate test case linked to the requirement or a rationale explaining why certain coverage criteria cannot be met. In other words, coverage is measured as the sum of coverage by test cases and coverage by justification, making 100% coverage achievable. Combined with running tests on separate hardware, this ensures an adequate verification of requirements across the entire system. We claim that the approach brings us significantly

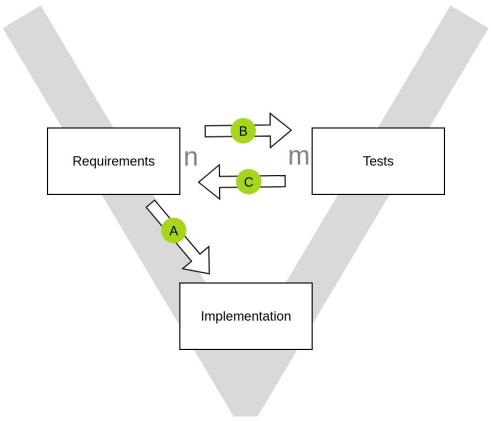

Figure 1 The triad of requirements, implementation, and tests.

closer to the maxim, *what you run, and run what you test* (in the aerospace domain, *test like you fly, and fly like you test* [13]). The actual system is tested, without any additional annotations, and it can later run unchanged in the production environment.

The presentation is structured as follows: in the next section, we review the well-known software development process based on requirements, code implementation, and testing. We then describe our systematic approach for identifying and closing gaps between requirements, tests, and the implementation. The third section provides an introduction to the technical implementation, including the requirements for the observation of integration tests and an introduction to the capture and processing of embedded traces.

2 A Systematic Approach for Reaching 100% Code Coverage

2.1 Software development process

Figure 1 shows the classic V-model, consisting of requirements, implementation, and their associated tests.

In software engineering, the transition from requirements to implementation (A) is complex. Engineers write code to implement requirements, but it is important to note that this process is not one-to-one. To verify the correctness of the implementation, appropriate tests are developed and executed based on the requirements.

This practice, known as *requirement test coverage*, ensures thorough software validation. Each requirement requires at least one test (usually several in practice) to systematically validate, improve quality, and meet industry standards such as DO-178C (Software Considerations in Airborne Systems and Equipment Certification) [4], ISO26262 (Functional Safety of Road Vehicles) [8], EN50128 (Railway applications) [5] and IEC 61508 (Functional safety of electrical/electronic/programmable electronic safety-related systems) [3].

There is an *n:m* relationship between requirements and tests, where for each requirement there must be at least one corresponding test (B), and likewise for each test there must be at least one corresponding requirement (C). In the industrial area, established application lifecycle management tools such as Polarion [14] or Codebeamer [9] provide the management of requirements and tests, but there are also new solutions such as the AI-supported MappingSpace [12]. The first step (A) to (C) serves as an essential test completeness criterion, but it is not sufficient to guarantee that all implemented code has been tested.

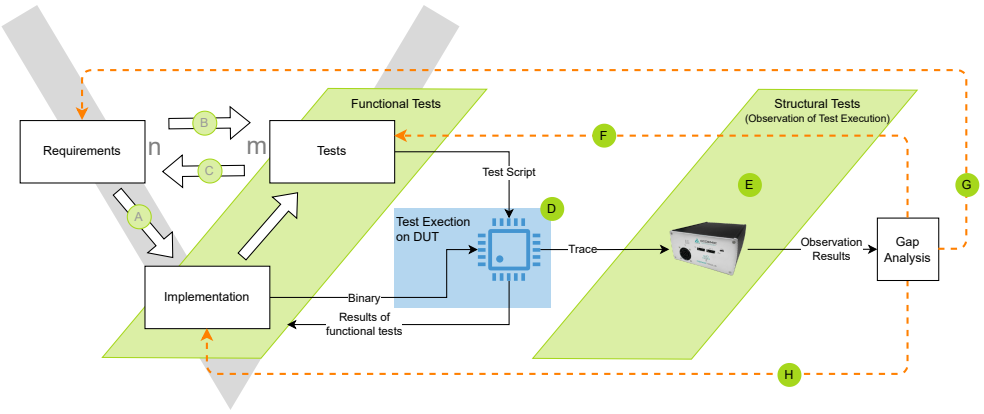

Figure 2 Systematic gap analysis identifies and eliminates gaps between requirements, testing, and implementation.

During the execution of the functional tests (D), the code coverage is measured to determine if all code elements have actually been executed (also called structural tests (E)). Typically, coverage gaps are observed at this stage (code coverage $< 100\%$) and it is necessary to carefully analyze the coverage gaps to identify potential problems or areas for improvement. Figure 2 illustrates the process of test execution, completeness measurement, and gap analysis.

In practice, achieving 100% code coverage is usually not feasible and there are often plausible explanations for coverage gaps, some of which we will outline here:

- Defensive programming: These techniques aim to catch errors and handle specific situations that should not actually occur, leading to code that cannot be reached. An example in the C programming language is the typical pattern `if (p == NULL) { ...`, which is to ensure that `p` is only accessed if it is not a null pointer. If the program is correct, `p` will never be a null pointer, and there will be no test case that makes the condition true.
- Test environment economics: In some cases, it may not be feasible or cost-effective to test certain aspects of the code, such as functional behavior outside of laboratory temperatures or simulating all external failure cases. Especially in cases of low system criticality, it is acceptable if some requirements are not met.
- Safeguards against critical environmental conditions: To ensure the safe operating range of sensors, subsystems, etc., environmental parameters are checked, and undesired or uncertified system usage is blocked by corresponding error routines. A measuring system, for example, might be certified for operation at ambient temperatures, and refuse to operate above that temperature. It may not be safe to test certain aspects of the code such as safeguards against critical environmental conditions, or even though the system could technically be tested in that environment, it is unnecessary as it would not be used as intended.
- Modular nature of the system: From an economic perspective, well-tested units should be reused as much as possible. Units are designed to be used in different contexts, but if only part of the unit's functionality is used in the overall system, the other parts cannot be covered by tests. For instance, consider a controller function that can be set to operate in proportional, differential, or integral modes; if only differential mode is used in the current application, the other modes cannot be tested by integration tests. However,

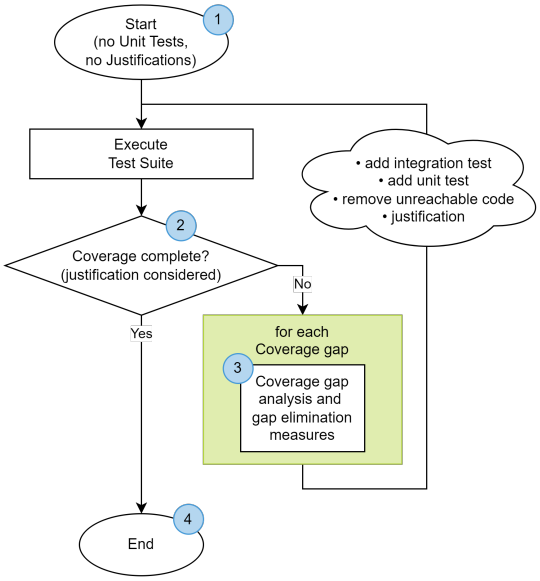

Figure 3 Systematic process for identification and elimination of coverage gaps.

reusable units should have corresponding unit tests to ensure their quality independently of the current application. In this case, unit tests can justify that parts of the code not covered by integration tests have nevertheless been sufficiently tested.

Therefore, achieving 100% code coverage requires either an adequate test case linked to the requirement, or a rationale explaining why certain coverage criteria cannot be met. In other words, coverage is measured as the sum of coverage by test cases and coverage by justification, making 100% coverage achievable.

The measured code coverage of multiple test executions is aggregated to identify the remaining coverage gaps (Figure 4).

If it is determined that there is a plausible explanation for certain coverage gaps, those gaps can be excluded from the measurement. This means that the coverage gaps will not be counted as part of the overall coverage and will not be included in the calculation of the coverage percentage.

However, there must be detailed documentation of the reason for each justification.

To support the exclusion of justified coverage gaps from the measurement, and to identify missing tests (F), missing requirements (G), and unreachable code (H), a systematic process for identifying and excluding these gaps must be implemented. This process is a critical step in the testing process and is designed to ensure that coverage results are accurate.

2.2 Rigorous identification and elimination of coverage gaps

Figure 5 illustrates the systematic process for incrementally identifying and eliminating of coverage gaps. The process typically follows these steps:

(a) Measure the initial test coverage of all integration tests (excluding unit tests) (1). This provides a baseline for the testing process and is used as a starting point for coverage gap analysis.

(b) Execute tests and measure code coverage (2).

(c) Review and analyze each coverage gap (3).

Figure 4 Aggregation of multiple coverage results.

(d) If the gap is justifiable, exclude it from the measurement. If the gap is not justifiable, run additional tests to increase coverage and close the gap.

(e) Repeat the process until all coverage gaps have been reviewed and analyzed, and the sum of covered and excluded code sections equals 100% of the code (4).

It is important to emphasize that this work should not be done too early in the development cycle, as it can be very costly. Instead, it is preferable to establish a stable software base and achieve full requirements test coverage before starting this work.

Each coverage gap must be carefully reviewed and analyzed to determine the cause of the gap (see label 3A in Figure 5). If it is determined that the gap is caused by a missing integration test (3B), an additional integration test (3E) can be added to increase coverage and close the gap. Before adding the additional integration test, it is important to check if there is also a missing requirement (3C) that needs to be added. If there is a missing requirement, it will also be added to the project (3D).

If the coverage gap analysis (3A) does not identify any missing integration tests, then the presence of unreachable code (3F) must be checked. Unreachable code is code that is not executed or used during the testing process and must be removed from the code base to improve the overall quality and maintainability of the code (3L).

If there is no unreachable code, consideration should be given to whether the gap can be filled by a unit test (3G). Code may not be testable by an integration test, requiring a unit test instead, under conditions such as (but not limited to) specific physical constraints. For example, Figure 6 illustrates a situation where the operating range of a sensor prevents it from being run in an integration test, but this range can still be examined by a unit test.

Adding a unit test (3H) to the test suite is only allowed in justified circumstances and must be thoroughly documented (3J). Otherwise, there's a risk that unit tests will mask the absence of integration tests - a common problem in traditional unit test-focused coverage evaluations. It is critical to distinguish between integration and unit tests to avoid masking integration-level coverage gaps with unit tests.

However, the initial exclusion of unit tests and their inclusion only in justified cases should not lead to a general undervaluation of unit tests. They remain essential, especially during development.

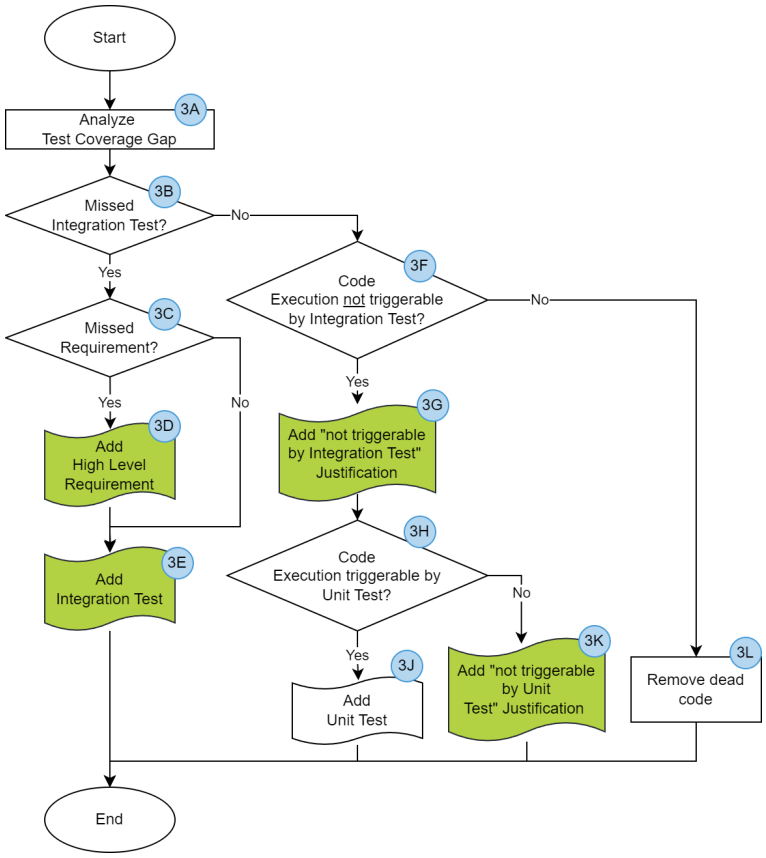

Figure 5 The process of identifying and eliminating potential coverage gaps. The unit tests added here (with a detailed explanation of why they are needed at this point) are added to the test base, which initially contains no unit tests. The green boxes highlight the potential benefits of using this approach compared to proving full code coverage based on unit testing alone. The highlighted blocks represent the improvement by integration test coverage achieved with the proposed method.

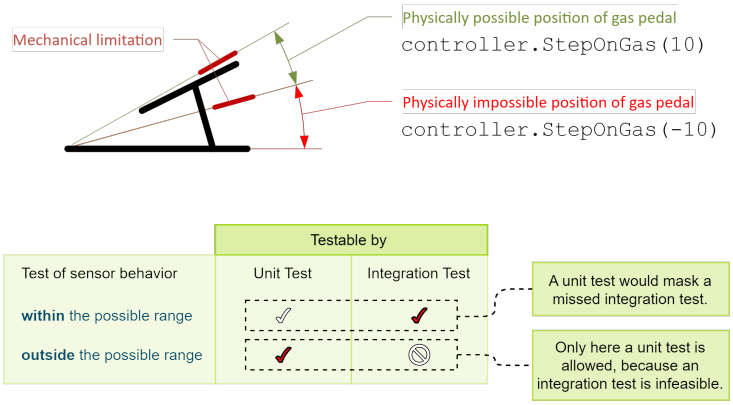

Figure 6 Example of a case where a unit test is acceptable because the test cannot be performed as an integration test. The throttle can only be moved within a mechanically defined range; tests with irregular sensor values must be performed as unit tests.

	Test Level	
Observation Requirements	***Unit Tests***	***System and Integration Tests***
Long Observation	Not required	Required
No (or minimal) change in Timing Behavior		
No (or minimal) change in Memory Footprint		

Figure 7 Observation Requirements for System, Integration, and Unit Testing.

If the gap cannot be covered by a unit test, a justification for the coverage gap is documented (3K). This documentation provides a record of the gap and the reasons why it was considered justifiable and excluded from measurement.

The goal of complete structural test coverage at the integration test level can be achieved by systematically adding missing tests, removing code sections that cannot be tested and carefully documenting the justification of code sections that are necessary but cannot be tested.

In particular, measuring code coverage is a mandatory requirement for certification according to standards such as DO-178C (Software Considerations in Airborne Systems and Equipment Certification) [4], ISO26262 (Functional Safety of Road Vehicles) [8], EN50128 (Railway applications) [5] and IEC 61508 (Functional safety of electrical/electronic/ programmable electronic safety-related systems) [3]. However, these standards leave open the level at which code coverage must be verified.

3 Technical Solution

3.1 Requirements

The process described seems intuitive and obvious. The question arises as to why structural coverage has mostly been determined at the unit test level and why the consistent step to higher test levels has not yet been taken for embedded systems.

The answer can be found by analyzing the requirements for observing an embedded processor (Figure 7). These are low for unit tests; only the control flow in an isolated code element needs to be examined. The method of automated software instrumentation has been used successfully for years. All relevant places in the software are provided with additional code that logs the execution of that place. This typically involves reading an execution counter from memory, incrementing it, and writing it back. The associated bus operations are non-deterministic on multiple processors.

However, software instrumentation negatively impacts integration testing: the additional code required for instrumentation consumes memory and processing time, both of which are limited resources in embedded systems. If there is not enough memory for the instrumentation, the instrumented application will not run. Even if this is still the case, the changed runtime behavior of the instrumented software version represents a significant limitation in the validity of the tests performed.

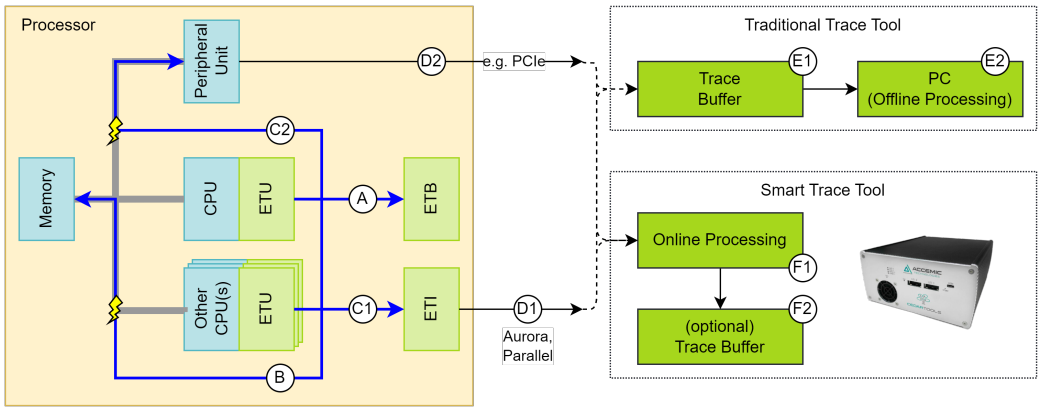

Figure 8 Embedded trace elements.

3.2 Embedded Trace

A suitable solution is to use the embedded trace [19] resources available on almost every processor. This technology is implemented through specific hardware structures that are tightly coupled to the CPU(s) and are used to capture data about the execution of code. The major advantage of using embedded trace is that the embedded trace unit (ETU) does not affect the CPU when monitoring the program flow, i.e. from the application's point of view it is not possible to tell whether the ETU is active or not. Figure 9 provides an overview of the individual elements and the trace data flow of a processor equipped with embedded trace.

There are several options for transferring the trace data generated by the ETU to an external tool. The trace data can be temporarily stored in the processor in a special embedded trace buffer (ETB) (A) or in the system memory (B). Trace data can also be output via dedicated embedded trace interfaces (ETI) (C1) or system interfaces (C2) to avoid the limited observation time caused by the memory limitations.

The direct output of the trace data via ETI (C1) is technically the most elegant, since it does not interfere with the processor. However, implementing an interface with the required high bandwidth is expensive, so some processor designs compromise by storing the trace data in the main memory (B) or outputting it via a system interface (C2). However, this is also done at the expense of the desired non-intrusiveness - the required bus operations interfere with the application (indicated by the yellow flashing symbols).

As shown in Figure 8, trace data is output directly using dedicated trace interfaces (parallel [6] or high-speed serial like Aurora [2]) (D1) or fast system interfaces (usually PCIe, sometimes USB or Ethernet) (D2).

This trace data is then received by an external trace tool. There are simple trace buffers and advanced smart trace tools.

Trace buffers (E1) typically contain several GB of memory where the received trace data is temporarily stored before the decoding and further processing in a PC (E2). The downside is that the size of the trace buffer is limited, which limits the amount of time that can be observed. This conflicts with the requirement of an integration test to be able to observe and analyze a system for as long as possible.

This problem is solved by a new generation of smart trace tools (F1) that perform live reconstruction of the control flow and compute online all the information needed to determine the code coverage of the observed integration tests (Figure 9). In principle, such a system

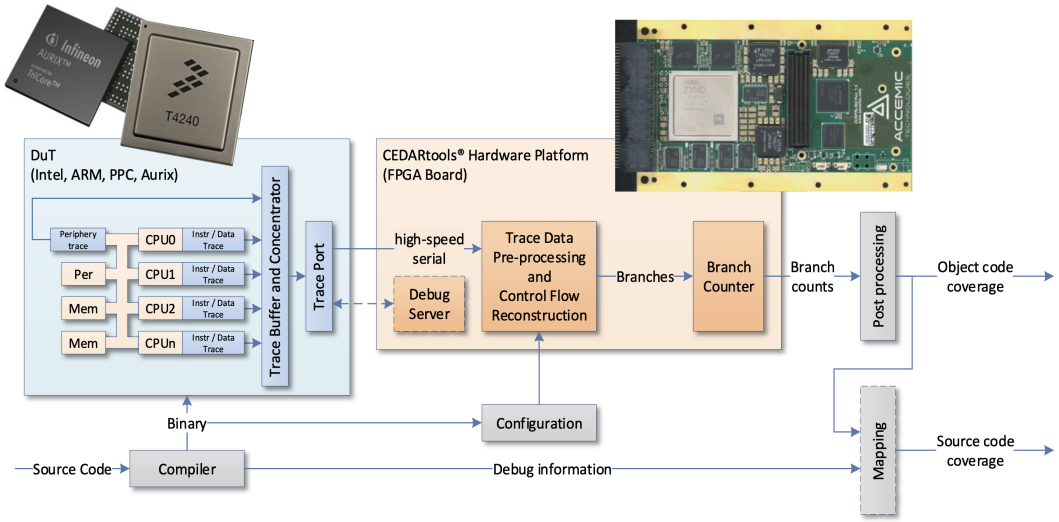

Figure 9 Embedded Trace-based Coverage Generation.

does not require a trace buffer, since the incoming trace data is processed immediately and continuously online. However, the system also has a trace buffer with a capacity of several GB (F2) for evidentiary purposes and other potential applications such as test impact analysis or hybrid WCET measurement. Since the control flow of the target system has been reconstructed in the system, the existing trace buffer can store the relevant trace snippets in a finely segmented manner more efficiently than with simple trace buffers.

All trace-based tools determine the control flow at object code level. Using the DWARF debug information [7], the object code coverage can be transformed into the source code coverage.

This transformation is not trivial, because an optimizing compiler can change the control flow. This is done by removing, adding (e.g. stack protection, vectorization), or rearranging code elements. These compiler-induced changes to the control flow should also be systematically tested. So far, these gaps have not been addressed in the applicable standards. The new state of the art, which makes this additional coverage analysis possible, must be brought to the attention of experts and taken into account in binding standards.

In addition, especially for safety-critical applications, it must be considered that a compiler may also be incorrectly implemented [10].

Trace-based code coverage and thus the method presented here works well for simple coverage levels such as call function coverage, source line coverage, or statement coverage. More sophisticated coverage metrics, such as branch coverage or MC/DC [17], require control-flow preserving compilation and reliable debug information. The authors are not currently aware of any viable solutions in this area, but intensive development work is underway to achieve a broadly applicable solution.

The previous discussion has shown how important embedded trace is to achieving the highest possible software quality for embedded systems. Established processors offer complex embedded trace solutions. A closer analysis of various use cases (such as the one presented here) reveals some gaps and functional limitations in the established embedded trace solutions. As part of the TRISTAN project [16], the authors are working on the development of an embedded trace IP (open source and commercial version) for the open standard instruction set architecture RISC-V [15] that comprehensively addresses the use case presented here.

4 Summary

We showed that by systematically detecting and eliminating coverage gaps, and missing integration and unit tests, it is possible to identify and eliminate missing requirements and unreachable code. A key enabler of this approach is smart trace tools that provide continuous and non-intrusive monitoring of embedded processors, ensuring the integrity of the program flow during execution. These technical developments now also allow code coverage to be measured during the execution of integration tests. The resulting approach thus brings us closer to fulfilling the important principle *test what you run, and run what you test.*

References

1 CEDARtools. Available: `https://accemic.com/cedartools/`.
2 Aurora Protocol Specification, 2007. Technical Report, Xilinx Inc.
3 IEC 61508:2010 Functional safety of electrical/ electronic/programmable electronic safety-related systems, 2010.
4 DO-178C, Software Considerations in Airborne Systems and Equipment Certification, December 2011.
5 EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, 2011.
6 MIPI PTI v2.0 Specification for Parallel Trace Interface, May 2011. Technical Report, MIPI Alliance, Inc.
7 DWARF Debugging Information Format Version 5, February 2017. Available: `http://www.dwarfstd.org`.
8 ISO 26262:2018. Road vehicles – Functional safety, 2018.
9 Codebeamer. Available: `https://www.ptc.com/en/products/codebeamer`.
10 GCC Bug-tracking System. Available: `https://gcc.gnu.org/bugzilla/`.
11 Gcov. Available: `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.
12 MappingSpace. Available: `https://www.ytdevops.com/home`.
13 NASA lessons. Available: `https://llis.nasa.gov/lesson/1196`.
14 Polarion. Available: `https://plm.sw.siemens.com/en-US/polarion/`.
15 RISC-V. Available: `https://riscv.org/`.
16 The TRISTAN Project. Available: `https://tristan-project.eu/`.
17 K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage. Technical report, NASA Langley Technical Report Server, 2001.
18 Martin Leucker, Martin Sachenbacher, Simon Wegener, Alexander Weiss, and Albert Schulz. Non-intrusive, continuous trace-based monitoring and its applications for system correctness, safety, and resilience. In *Workshop on Principles of Diagnosis (DX'23)*, 2023.
19 T. B. Preußer, S. Gautham, A. D. Rajagopala, C. R. Elks, and A. Weiss. Everything you always wanted to know about embedded trace. *Computer*, 55(2):34–43, February 2022. `doi:10.1109/MC.2021.3098965`.
20 Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker. *Model-Based System Architecture*. Wiley Series in Systems Engineering and Management. John Wiley & Sons, 2nd edition, 2022.