embeddedworld2020
Exhibition&Conference
...it's a smarter world

Conclusive On-the-fly Validation of High-Level Functional Tests

Thomas B. Preußer
Accemic Technologies GmbH
Dresden, Germany
tpreusser@accemic.com

Alexander Weiss
Accemic Technologies GmbH
Kiefersfelden, Germany
aweiss@accemic.com

***Abstract*—Structural testing is an important acceptance criterion for safety-critical embedded and cyber-physical systems (as in Aerospace, Transport, and Critical Infrastructure). The coverage of both all specified requirements and, vice versa, all code to be deployed makes testing very difficult and costly. We elaborate on a solution that validates the conclusiveness of high-level functional tests on fully-integrated safety-critical applications in a non-intrusive fashion. Our approach performs an online analysis of hardware processor trace data in real-time to establish coverage proofs on-the-fly during test runs. It offers deep insights into the completeness of both the tests and their underlying requirements. Establishing the validity of tests on a high functional level reduces the effort enormously that is required on lower, less integrated levels to achieve and justify conclusive coverage statements. The savings achieved by our approach in the development process will be demonstrated and quantified.**

***Index Terms*—Processor Execution Trace, Online Monitoring, Structural Tests, Requirements-Based Tests**

I. Introduction

The continuously increasing complexity, even in the domains of safety-critical and cyberphysical systems, pushes the number of defects, many of which are only observed after the release of a product. Fighting this late and most costly manifestation of defects, the use of more capable, thorough methods on higher test levels helps to establish and maintain economically competitive and effective test procedures.

Both the measurement of the structural test coverage and the automated runtime analysis during the execution of functional integration and system tests are powerful testing instruments. Their associated requirement for comprehensive and continuous system observability, however, poses difficult challenges. Classic software instrumentation manipulates the application under test, heavily impacting its timing behavior so that the confidence in the test is reduced or the test itself becomes unfeasible on higher test levels. Solutions based on hardware execution traces can avoid the massively intrusive instrumentation of the tested application. However, they are currently limited to snapshots over short time spans. This harshly limits their capability to derive conclusive statements about tests executed on higher, more integrated levels.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732016.

It is the live online analysis of execution trace data which enables the non-intrusive continuous monitoring of test runs to establish desired test properties. In order to be able to use this technical option, high-bandwidth access to the trace data output by the processor must be available, and adjustments must be made to the software architecture. This paper describes such a hardware solution and its corresponding testing workflow.

The remainder of this paper will first motivate the increasing demand for capable test procedures in Sec. II. Sec. III will continue by giving an overview of the state of the art with a strong focus on coverage metrics and their current dynamic measurement approaches. The proposed live analysis of the execution trace will be introduced in Sec. IV, which will name the specific challenges this approach has to master and discuss the opportunities provided by this novel capability in detail with a strong focus on coverage measurements. Finally, Sec. V will list the enabling design requirements that need to be considered in the architecture of a system platform that is to benefit from the described opportunities before Sec. VI concludes the paper.

II. Motivation

Embedded systems are becoming increasingly powerful and complex. This is accompanied by the unpleasant fact that the relative error probability of software increases with its complexity [1]: The more code, the disproportionately more defects are initially present in the software and the worse is the efficiency of error correction. With the increasing use of multicore and networked multiprocessor systems as well as third-party software components, more and more non-deterministic error patterns, which are difficult to reproduce, are added. Assuming an error correction ratio of about 95% during the development process [1], a proud 5% of defects remain in the release code - in absolute numbers, this can mean several thousand defects. Nobody has to be ashamed of this, even NASA has this problem during their missions [2].

> McKinsey & Company 2018 [3]:
> "Snowballing complexity is causing significant software-related quality issues, as evidenced by millions of recent vehicle recalls."

These errors range from errors in the requirements specification over trivial implementation defects all the way to

complex, non-deterministic error patterns. Due to missing or overly simplified models, static analysis quickly reaches its limits, so that the dynamic analysis of embedded systems becomes increasingly important.

The freedom from defects is an illusion although this ideal should be the goal of any serious project management (not only) in the embedded area. In addition to a well-thought-out system architecture and good implementation, (a) tests that are as complete as possible and (b) precautions for dealing with errors in the field are important prerequisites for being able to develop and market a product on time and in the best possible quality.

III. State of the Art

A. Structural Coverage Criteria

The standards for the development of safety-critical systems, such as DO-178C [4] and ISO 26262:2018 [5], define concrete requirements for the test process, the test techniques to be applied and the proof of the completeness of these tests (structural coverage). Depending on the criticality of the application, it must be demonstrated as completely as possible that all instructions (*statement coverage*), all branches (*branch coverage*) or all relevant combinations of conditions (*MC/DC*) were used during the tests. The standards leave it largely open, on which test level the measurement of structural coverage are performed.

In addition, there is a structural test criterion, data flow coverage and the analysis of the coupling of data and control flow, which has been largely neglected up to now [6]. Although these test criteria are only dealt with in passing and not very specifical in various standards, they appear to us to be an essential supplement to the measurement of control flow coverage.

In current practice, the proof of structural coverage is usually provided by means of ”hand-tailored” functional tests on the module test level. This is often accepted as proof for the test completeness as required by the standards. However, structural tests should not serve the purpose of being able to hand over impressive reports to the certification authorities but should demonstrate the completeness of the conducted functional tests so as to raise test confidence and enable an effective hunt for the many defects still present in the system.

Therefore, we advocate (a) the measurement of structural test coverage during the execution of integration and system tests and (b) the measurement of the coupling of data and control flow.

The measurement of structural test coverage during the execution of higher-level (functional) tests (HLT) has the great advantage that an exact statement about the completeness of these tests can be made. For non-executed program code, it must be determined why it has not been reached by the test. In some cases, one will come to the conclusion that this code can only be checked on module test level but regularly, also HLT gaps will be revealed. Since large parts of the program code have already been exercised by the execution of HLTs, it is no longer necessary to generate and document the corresponding tests on the module test level for the purpose of a test coverage proof.

The dynamic measurement of the coupling of data and control flow provides another powerful test completeness criterion. For each variable, the test must show that it is initialized and later used (def-use pair). Just as the structural control flow coverage provides a statement about the completeness of tests on the control flow level, a similar statement can thus be achieved for the data flow.

B. Software Instrumentation

Software instrumentation adds code to the application that logs the execution of the program during a test or a debug run. The program code can be instrumented automatically by using off-the-shelf tools. However, integration and system tests, which examine the interaction of components, lose their validity and confidence as they are based on an altered application with a different memory layout and a changed temporal behavior during execution.

C. Offline Analysis of Embedded Trace

Almost all modern processors have a standardized embedded trace unit (e.g. ARM® CoreSight [7], Intel® Processor Trace [8], Infineon ED [9], NXP QorIQ® [10]). This unit outputs information about the executed program flow without influencing it. Depending on the architecture and trace configuration, the temporal behavior and data accesses can also be reconstructed. In addition, further units can support a lightweight hardware-supported instrumentation (which is therefore acceptable in release code) as well as the tracing of peripheral units (memory controllers, communication units [11]).

In classic embedded trace solutions, the high-bandwidth trace data (several Gbit/s) is usually stored in a buffer memory. After the end of the test run, the program flow is reconstructed on a PC to determine, for example, the structural coverage. The limits of this procedure are the observation time, which is constrained by the available buffer memory, and the additional computing time required for the offline reconstruction of the program flow.

Although the concrete trace protocols of different processor architectures differ, they always convey all the information that is necessary for the reconstruction of the program control flow.

IV. Live Analysis of Embedded Trace

A. Challenges

The live analysis of the execution trace at run time is a quantum leap enhancement over the offline analysis of the recorded trace data as it effectively eliminates the bottlenecks imposed by the need for the intermediate buffering. However, there are two major technical challenges to be overcome:

1) The highly compressed trace data stream must be decompressed and the control flow of the CPU(s) must be

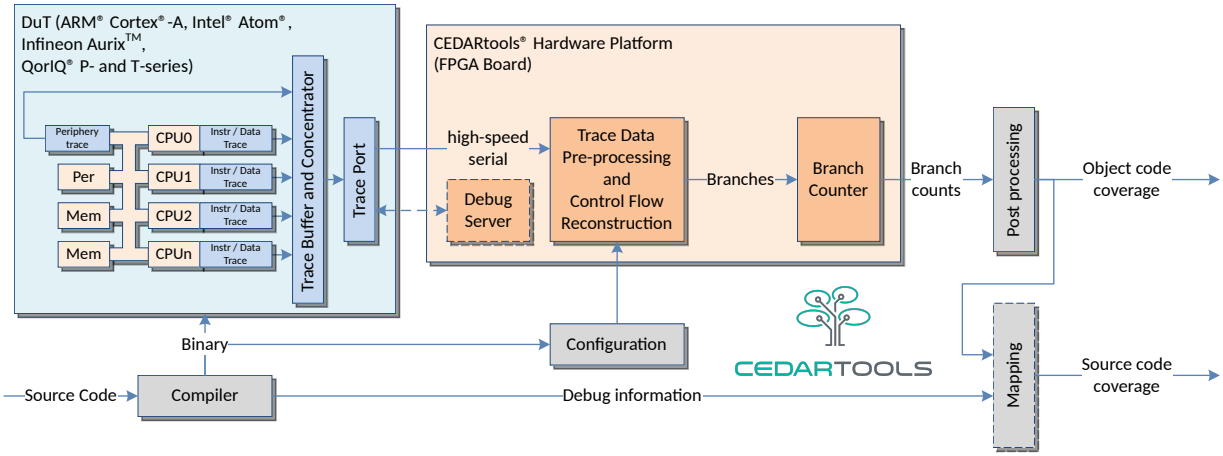

Fig. 1. Overview of the Live Structural Coverage Measurement Flow

reconstructed. This demanding computation must often cope with the execution trace from multiple fast CPUs that are running at nominal clock frequencies above 1 GHz. This decoding may further be challenged by additional abstractions and indirections introduced by different operating systems.

2) The reconstructed control flow must be analyzed into an apt event stream abstraction that is suitable to drive the desired of various possible backend task. For example, (a) branch information for the coverage analysis may be recorded or (b) dynamic properties over the event stream may be computed and validated against a temporal logic specification.

The live analysis of trace data over arbitrarily long program runs enables (a) the measurement of the control flow coverage during the execution of HLTs, as well as (b) the dynamic constraints monitoring, which can be used to validate (b1) the correct coupling of the data and control flow as well as (b2) the runtime behavior of an application.

B. Coverage Measurements

As discussed in detail by the Certification Authorities Software Team (CAST) [12], measuring the structural coverage on the object code level provides different information than the measurement on the source code level. By evaluating the debug information generated by the compiler, the measured object code coverage can be backannotated to provide a source code level view on the obtained coverage data.

The corresponding overall workflow is illustrated in Fig. 1. The target device under test (DuT) is a microprocessor executing regularly compiled, native code. This execution is monitored via the platform-specific trace port. The CEDARtools® hardware platform decompresses this trace on the fly, reconstructs the application control flow in real time and logs sufficient execution data for a conclusive labeling of the application's control flow graph with respect to the desired coverage criteria. The bulk of this data comprises branch execution counters. Depending on the particular architecture and execution environments, statistics about other control-flow-changing events, such as timer interrupts or task switches, may have to be included for a consistent final picture. Observe that this object code coverage can be measured plainly on the basis of the executed binary. The presentation of the obtained results can be refined by the backannotation to the application sources as long as these sources are available and the compiler has produced debug information along with the executable.

Fig. 2 depicts an example of such an annotated HTML coverage report. It shows the C source code lines, which can be optionally expanded into their corresponding assembly statements. Each line is prefixed by its observed execution count. For branches, this count is differentiated into the number of sequential continuations and the number of taken control flow changes. The latter appears behind an arrow symbol and is, in fact, the only figure in the case of unconditional branches and function calls. Conditional branches are further annotated by a plus-minus pair indicating whether they have ever been observed being *taken* and *not taken* during the test. In order to improve the visual perception, failures to meet coverage criteria are color-coded.

Additionally, a (typically partial) control flow graph annotated with the measured coverage data may be produced to understand the structure of a test. An example is depicted in Fig. 3. Implied sequential control flow is largely pruned. It is, however, identified as blue edges from conditional branch

```
17                          // Unfold Collatz sequence and return its length.
18                          // - n <= 1 will not execute the while loop at all.
19                          // - n = 2^k will never trigger the 3*n+1 path.
20              1970        unsigned collatz_depth(unsigned n) {
21              1970        unsigned depth = 0;
22    [+,+]     13987       while(n > 1) {
                ↪ 1970        40089B: eb 24 jmp 4008c1
                13987         4008C1: 83 7d ec 01 cmpl $0x1,-0x14(%rbp)
      [+,+]     1970 ↪
                12017         4008C5: 77 d6 ja 40089d
23    [+,+]     12017       n = (n&1)? 3*n+1 : n/2;
                12017         40089D: 8b 45 ec mov -0x14(%rbp),%eax
                12017         4008A0: 83 e0 01 and $0x1,%eax
                12017         4008A3: 85 c0 test %eax,%eax
      [+,+]     3152 ↪ 8865   4008A5: 74 0e je 4008b5
                3152          4008A7: 8b 55 ec mov -0x14(%rbp),%edx
                3152          4008AA: 89 d0 mov %edx,%eax
                3152          4008AC: 01 c0 add %eax,%eax
                3152          4008AE: 01 d0 add %edx,%eax
                3152          4008B0: 83 c0 01 add $0x1,%eax
                ↪ 3152        4008B3: eb 05 jmp 4008ba
                8865          4008B5: 8b 45 ec mov -0x14(%rbp),%eax
                8865          4008B8: d1 e8 shr %eax
                12017         4008BA: 89 45 ec mov %eax,-0x14(%rbp)
24              12017       depth++;
25                          }
```

Fig. 2. Coverage Data Backannotated into Source-Code View

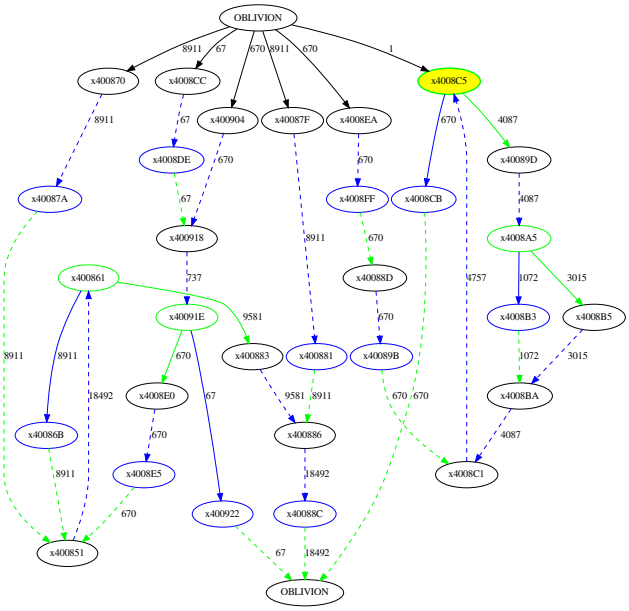

Fig. 3. (Partial) Control Flow Graph Annotated with Coverage Data

nodes and other code points that have happened to be branch destinations. Branching control flows, on the other hand, are depicted in green.

The implemented coverage measurement is capable of live consistent data snapshots in the middle of executing tests. Hence, also the coverage progress across a test can be monitored as a meta metric. The node highlighted in the graph of Fig. 3 is the point of the execution observed by the depicted snapshot. This can be easily verified by comparing the total in- and out-degrees of the edges terminating in this node.

Due to the non-intrusive continuous measurement, this approach can be used to conveniently measure the code coverage during integration and system tests. This allows an exact statement about the completeness of these tests. In addition, the effort for the development and documentation of structural tests on the module test level can be reduced significantly if the test coverage for the corresponding code segments is already proven on a higher test level.

Measuring the coverage directly on the deployed and executed object code has numerous benefits with respect to the confidence in the validity of the derived quality claims. Most of all, the compiler is removed from the coverage interpretation as it is now based on the control flow produced by the compiler rather than on the control of the source code provided to the compiler. Secondly, *not* relying on software instrumentation eliminates its inherent dilemma of either (a) bearing the cost of added instrumentation code in its productive deployment or (b) deploying a more efficient, uninstrumented but hence altered code base in the field. Both of these options are undesirable. The engineering choice is made on the bases of assumed criticality. Trace-based coverage certification removes the need for this trade-off between cost and safety altogether.

Beyond these hard safety-related benefits, many aspects of engineering a system are simplified. As the coverage-based test qualification is working on object rather than source, it is agnostic to the choice of source code language. The cost of migrating between programming languages or even using different ones within one project is greatly reduced. Language choices are no longer dictated by legacy but can be made flexibly selecting the best fit for each individual given task.

The trace-based coverage measurement allows shifting much weight away from module up to integration and system tests as it scales painlessly. It, thus, allows to deliver the coverage-based test qualification on the level of or close to the original system requirements specification. The effort to derive and validate decomposed lower-level test specifications is reduced to corner cases that cannot be triggered on a higher integration level. More importantly, the comprehensive high-level coverage measurement provides an immediate feedback to the quality of the tests on the respective test level. It reveals incomplete requirements specifications and incomplete functional tests. A strong incentive to enhance the high-level tests is created as it allows to avoid walking down and up the underlying hierarchy levels.

The main drawback of working on the object-code level is that the traceability of results back to the source code may be difficult. This, at least, requires the used compiler to emit debug information along with the generated object code. Difficulties in the result interpretation may be introduced by aggressive compiler optimizations, which may make it harder to understand why a coverage criterion is missed at a certain point of the test. Closely related to this issue is also our ongoing work on the reliable tracking of the individual flags contribution to a multi-condition.

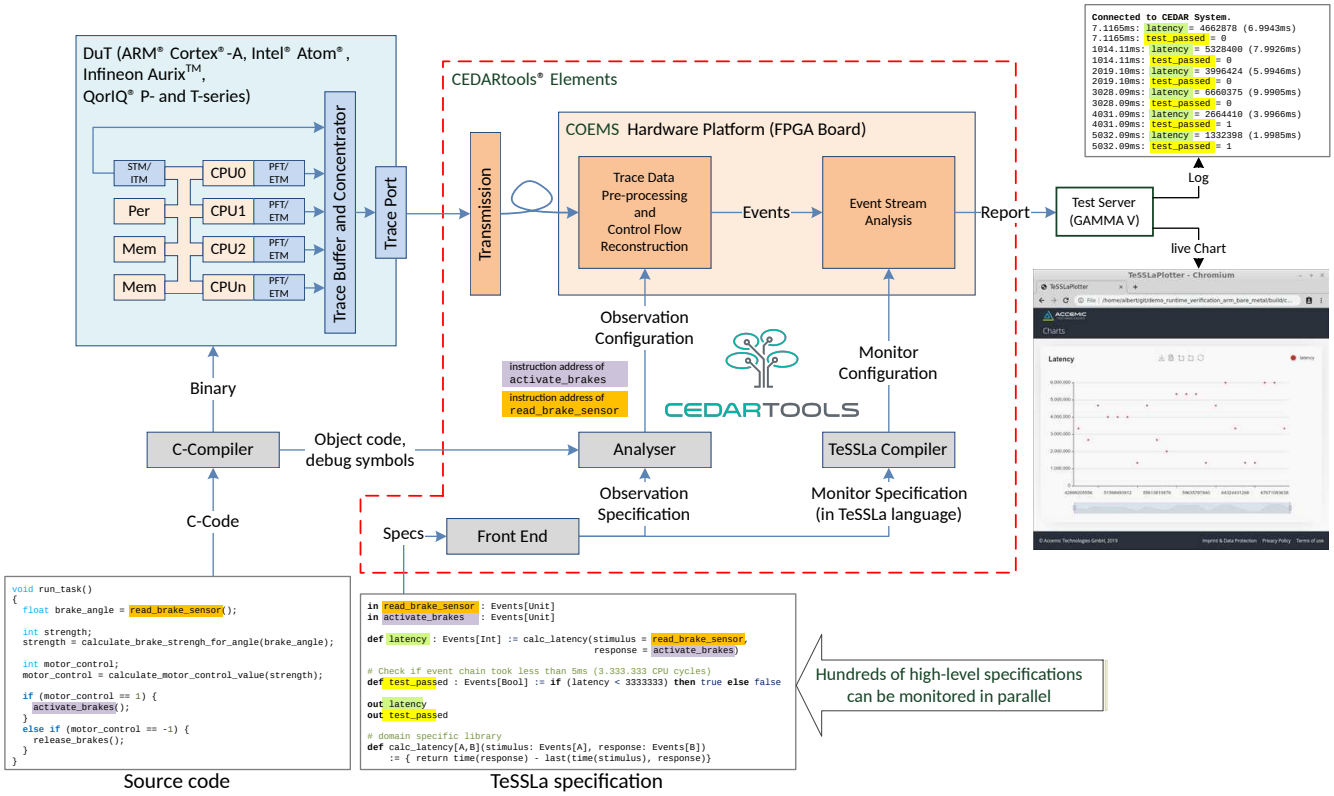

Fig. 4. Overview of the Runtime Verification Flow

C. Runtime Verification

Runtime verification is another use case of the online execution trace analysis. It establishes a powerful tool for testing and troubleshooting complex systems. During the continuous reconstruction of the control flow, it is possible to mark certain instruction addresses. When these instructions are executed, elements are inserted into the emitted event stream, which can then be examined online for specified temporal properties. The event processing units used can be configured in the high-level language TeSSLa [13], [14] and a large number of temporal properties can be monitored in parallel. The TeSSLa macro support also enables the easy adoption of industry standards like the AUTOSAR Timing Extension (TIMEX) [15] or AMALTHEA [16] to describe temporal behavior.

The CEDARtools® solution leverages event processing units that execute low-level TeSSLa operators natively. So, they are merely re-programmed for a given monitoring task. A time-consuming, application-specific synthesis of FPGA logic structures is not required. Thus, a change of the high-level property description can be applied to a trace data stream within seconds.

The overall workflow of the dynamic runtime verfication is illustrated in Fig. 4. The physical hardware setup matches the coverage measurement flow. Only within the FPGA, the coverage statistics unit is replaced by a programmable event stream analysis. Its configuration is compiled from a TeSSLa specification, an additional input to the workflow. It describes the monitoring task, the properties to compute and outputs to emit from the consumed control flow events.

V. Designing Custom Platforms for Testability

Responsible project planning must meet certain precautions to ensure the comprehensive observability for test and debug purposes. This implies for the architecture of the designated system platform:

- The trace interface must be accessible. For Intel® processors, the corresponding USB port should be available; for other architectures, the access to the corresponding trace interfaces (mostly Aurora or parallel) must be considered in the hardware design. Short-sighted cost savings at this point may tremendously complicate the critical system integration and debugging and, consequently, even endanger the success of the project.
- The initialization of the trace interface must be ensured. This is done either during the startup routine or by external access to the corresponding control registers (for example, via JTAG). For both options, the consequences for the

safety and/or security architecture must be considered carefully.

- Since many processor architectures only provide a limited data trace, further observability requirements should be considered. This may, for example, result in leveraging hardware-supported instrumentation (e.g. Intel® Processor Trace [8] PTWrite instructions, NXP QorIQ® [11] Data Acquisition Messages) or in mapping relevant variables to designated memory areas that are observable by the data trace.

VI. Conclusions

This paper has descibed an innovative approach that exploits execution trace data for the online monitoring of embedded processors for the on-the-fly system analysis. As a promising use case for this capability, the coverage measurement of long-running integration and system tests has been proposed and its numerous benefits have been described. The dynamic verification of runtime properties has been suggested as another possible use case. The requirements for system platform designs to enable the leveraging of the described benefits have been given.

References

[1] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley, 2011.

[2] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, p. 447–456.

[3] Burkacky, J. Deichmann, G. Doll, and C. Knochenhauer, *Rethinking Car Software and Electronics Architecture*. McKinsey, Feb. 2018.

[4] "DO-178C: Software considerations in airborne systems and equipment certification," RTCA, Dec. 2011.

[5] "ISO 26262:2018: Road vehicles – functional safety," International Organization for Standardization, 2018.

[6] "ED-12C: Software considerations in airborne systems and equipment certification," EUROCAE, 2011.

[7] "CoreSight™ architecture specification v2.0 ARM IHI 0029B," ARM Ltd., 2013.

[8] "Intel® 64 and IA-32 architectures software developer's manual," Intel Corporation, 2016.

[9] "TC29/7/6/3x ED Emulation devices target specification," Infineon Technologies AG, 2013.

[10] "IEEE-ISTO 5001TM-2012: The Nexus 5001 forum - standard for a global embedded processor debug interface," IEEE-ISTO, Jun. 2012.

[11] "P4080 advanced QorIQ debug and performance monitoring reference manual, Rev. F," Freescale Semiconductor, Inc., 2012.

[12] "Position paper CAST-17: Structural coverage of object code," FAA Certification Authorities Software Team (CAST), Jun. 2003.

[13] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "TeSSLa: Temporal stream-based specification language," in *Formal Methods: Foundations and Applications*, T. Massoni and M. R. Mousavi, Eds. Cham: Springer International Publishing, 2018, pp. 144–162.

[14] "TeSSLa: Temporal stream-based specification language," Universität zu Lübeck. [Online]. Available: https://www.tessla.io/

[15] "AUTOSAR-TIMEX: Specification of timing extensions," AUTOSAR. [Online]. Available: http://www.autosar.org/

[16] "AMALTHEA - An open platform project for embedded multicore systems." [Online]. Available: http://www.amalthea-project.org/