Automatic Firmware Verification for Automotive Applications

Torsten Andre, Infineon Technologies Austria AG, Villach, Austria

Daniel Valtiner, Infineon Technologies Austria AG, Villach, Austria

***Abstract*— Automotive applications require thorough testing of integrated circuits (ICs) to ensure reliability and safety. Companies aim to reduce time-to-market of their products through parallel development of hardware and firmware. Firmware engineers are challenged to develop and verify resource-constrained firmware without the final hardware available. While verification on host systems allows parallel development and verification, it prohibits profiling the firmware's resource consumption. We present a case study of an internally developed verification framework used in a recent project supporting test-driven development (TDD). The framework uses an instruction set simulator (ISS) to overcome limitations of host system verification. Designers fully benefit from TDD and available profiling information considering resource consumption in constrained environments. The verification framework supports ISO 26262 compliant FW verification.**

Keywords*— *Firmware; verification; ISO 26262; testing; automotive; embedded system; test driven development; TDD

I. Introduction

We consider highly optimized integrated circuits (ICs) such as sensors comprising hardware (HW) and firmware (FW). In automotive applications, often FW and HW are tightly coupled and optimized to ensure the high safety and reliability standards required by ISO 262626 [1], an automotive standard which defines methods and processes for safety critical electrical and/or electronic systems. While the HW is developed, the FW designer is challenged to reduce time-to-market while ensuring the safety of the FW by thorough testing and verification.

FW and HW can be developed in parallel. Hence, the FW must be verified on a HW substitute because the HW is not available yet. ISO 26262 acknowledges parallel development by defining unit and integration testing which do not have to be executed on the target HW. However, the test environment shall resemble the target architecture as closely as possible [1]. In the absence of the fully implemented target HW, FW verification usually starts on a host system using HW abstraction [2]. FW designers benefit in multiple ways. They may select from a wide range of unit testing frameworks [3]. Frameworks reduce the time to specify and alter tests. Verification can be fully automated simplifying the implementation of a regression strategy. Lastly, execution time is often reduced by running the FW on a more powerful host than the target HW.

Verification on host systems has three major drawbacks. Firstly, profiling of the FW is delayed until the target HW is available. Profiling is important considering the limited availability of resources often found in DSPs such as execution time for real time applications or available storage such as ROM or flash. Detecting bottlenecks early during the design phase has the advantage that the HW specification can be adapted if the available FW resources are not sufficient. Secondly, testing on the host system prohibits verification of FW written in assembler specific to the target HW. While using high level languages such as the C programming language is often preferable to increase readability, it might be necessary to implement parts of the FW in assembler. Thirdly, a high confidence level in the correctness of the target HW compiler is required, i.e., the FW is required to behave identically on the host and target HW platform. Usage of an instruction set simulator (ISS) [4] removes the drawbacks. Instead of compiling the FW for the host system, the FW is compiled for the target architecture and simulated by the ISS. It allows accurate execution of the FW used in the target HW.

In this paper we introduce a solution to FW verification using ISSs in a fully integrated verification framework supporting ISO 26262 compliant testing for automotive applications. Its key advantages are cycle accurate verification of the FW abstracted from the (not yet available) HW. The framework allows designers to use test-driven development (TDD) [5] [6] that facilitates a combination of simple test specification and automated, fast test execution. An HTML report presents a graphical overview of the test results. It additionally presents profiling information to designers allowing early prototyping and optimization of resource constrained FW. In cases where available resources are scarce, the HW may be adapted early on. The framework's sole purpose is FW verification. Verification of HW behavior and integration of FW and HW are out of scope.

The verification framework was used in recent projects and proved to reduce the number of bugs early on while speeding up FW development significantly.

II. Related Work

The hardware design process of ICs on a given technology platform can be grouped in two main disciplines: analogue and digital design. Digital design is driven by increasing complexity and strongly reduced development time. Therefore hardware description languages (HDL) such as VHDL or SystemVerilog became popular. The increasing complexity impacts FW development. Heinen and Joost deal with this problem and give an overview on FW verification methods and related process flows in the field of telecommunication electronics [7]. They use an approach with an ISS and transaction level modeling (TLM) [8] [9], but apply it on a different type of system with a test bench designed for specific purposes of radio frequency networks. A general overview on test driven development and continuous integration for embedded software is shown in [10]; further they discuss the mechanisms and functionality of *CMock* [11], which is a similar concept compared to the framework approach presented in this paper. They show an approach using *Rake* [12], which in our case is not applicable due to the fact that the target core architecture and resulting timing behavior cannot be analyzed because an executable file for a host system is generated.

In order to keep efficiency within continuously practicing test driven development, [13] demonstrate a reuse oriented approach for testing of safety critical systems in the domain of medical devices. Reusability is important also for our framework. The framework will be, besides future extensions on its functionality, reused for several projects. Minimizing the effort of developers to configure and adapt the framework increases productivity and acceptance. The authors' approach on testing is to decouple a conceptual model from implementation details by abstraction, where our model is capable of this decoupling as well as performing tests on full system level. For further investigations on the topic of functional safety and ISO 26262, a comprehensive overview on the standard and its different parts is shown in [14].

III. Verification of Resource Constrained Firmware in Automotive Applications

Development using TDD encourages engineers to first design tests based on requirements that the design needs to implement. Safety and reliability or resource constraints often require strict timings such as processing delay, acceptable jitter, or refresh rates of data. Implementation of functionally, which may seem straight forward, often becomes challenging in the light of timing constraints.

Verification processes are organized hierarchically to reduce development costs [2]. At the start of the development phase, FW units are tested to ensure correct working of the isolated unit. A FW unit encapsulates basic functionality of high cohesion which is subject to standalone testing. Depending on the complexity and context, FW units may be broken down to smaller code pieces which may be tested independently.

Once development progresses, the complexity of the verification increases by integrating multiple FW units (FW integration testing). The goal of FW integration testing is to ensure the correct interaction between units. Verification focuses on unit interfaces, control and data flow instead of basic functionality already covered during unit testing.

Having completed FW integration verification, the FW should be fully compliant to its specification. Nevertheless, its integration with HW (system integration testing) needs to be verified. HW and FW designers may have different understandings of data passed between HW/FW interfaces or flows. The correct working of FW and HW is verified in top level (TL) simulations. TL simulations include the actual digital HW and FW, not models thereof. The simulations take comparably long time and the FW debugging capabilities are often limited.

While such top level simulations including HW and FW may be used to identify problems at system integration level, they are too time-consuming for unit testing. The verification framework described herein fills the gap. It is optimized for FW unit and FW integration verification. System integration testing can focus on integration by using the already verified FW. Simulations for system integration testing are performed on the actual HW and not on models to prevent modeling errors. The verification of HW behavior is out of scope of the framework .

IV. Firmware Verification Framework

A. Architecture Overview

Figure 1 depicts the architecture of the verification framework. It can be split into *inputs*, *HW abstraction*, *test bench*, and *outputs*. Note that manual specification by users is reduced to a minimum. A configuration file may need adaptation once per project. It defines basic parameters controlling the behavior of the test framework, e.g., data access times of the persistent memory or clock rate. The tester, described below, needs to be implemented once per unit test. The test spec, also described below, is subject to continuous modification if new test cases need to be added. Other changes to the framework are not required.

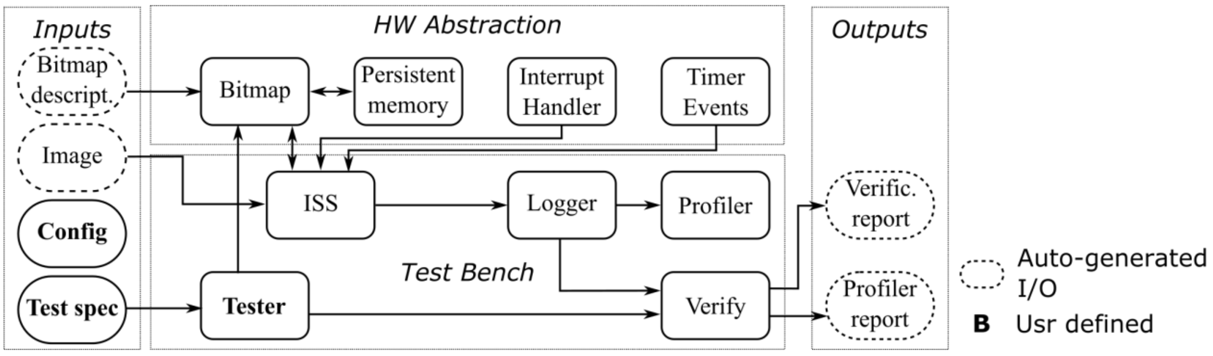

Figure 1 Verification Framework Architecture

The heart of the verification environment is the ISS that executes the binary FW image cycle accurate. Different ISSs may be plugged into the framework. ISSs are available for in-house developed architectures and ARM. Most ISSs are implemented in SystemC, the verification framework in approximately 15.000 lines of C++ code. It was developed in parallel to a product, which required verification capabilities not yet available, over a period of six months. Development continues to add further capabilities. Communication between the ISS and framework is handled by callbacks. Events may be registered at the ISS triggering function callbacks. The callbacks are used for data exchange. Information passed from the ISS to the framework includes executed opcodes, contents of internal registers, program counter, interrupts, or debugging and error signals. Using the same callbacks, any of the information provided may be altered by the verification framework enabling abstract HW modeling and fault injections. Using callbacks instead of e.g. TLM, as suggested in [8] [9], has numerous advantages. Firstly, FW designers do not need to know SystemC. Instead they may implement models in a familiar language C/C++. Secondly, designers do not need to install SystemC headers and libraries. The ISS is offered as a library already including SystemC. Thirdly, using callbacks ensures backward-compatibility to previous ISS implementations still in use.

B. Hardware Abstraction

Commonly the HW is mocked by e.g. linking mock implementations [10] or extensive usage of inline functions [2]. Instead, we propose to model the interaction between FW and HW using a well-defined HW-SW interface. The interface comprises all registers which are accessed by FW and HW for processing. We denote the set of all registers accessible by the FW as bitmap. The bitmap fully abstracts the FW from the HW and is used to feed input values to the FW. The verification flow initializes the bitmap with values defined in the test specification. After test execution, the values stored in the bitmap by the FW are compared to reference values also stored in the test specification. Persistent memory such as EEPROM or flash memory extends the bitmap. In comparison to accessing HW registers mapped in the bitmap, accesses to persistent memory may require retrieving data by paging which delays execution. Resulting delays are modeled accordingly.

The bitmap description defines the interface between HW and SW. It is derived from a central bitmap specification defined in an internal format such as IP-XACT or SystemRDL. The specification is a single point of definition outside the verification framework ensuring that FW, HW, and verification framework always use the same version of the bitmap specification. From the single bitmap specification, various representations are generated: the definition of variables for the FW; the HW implementation of the registers; and the bitmap description for the verification framework. The framework automatically generates the bitmap based on its specification. No manual editing is required.

Additionally, timing events may be registered at the verification framework to simulate peripheral HW. All times are measured by means of executed clock cycles automatically tracked by the framework. While execution time (in reference to actual time) depends on oscillator frequency and drift, the number of executed clock cycles remains the same. Counting clock cycles generalizes the verification w.r.t. oscillator parameters. Conversion between clock cycles and absolute time (or vice versa) can be done by multiplication of a factor.

Currently HW abstraction supports timers and interrupts. Timer values may be mapped to bitmap registers. Their content is automatically in- or decreased by the framework every clock cycle. Interrupts may be registered to fire at a defined clock cycle counter, after defined intervals, or (delayed) at specific events such as specific bitmap register values. Once the ISS reaches the defined number of clock cycles, the verification framework fires the interrupt which will be handled by the ISS the next clock cycle.

The current status of the HW abstraction is sufficient to verify FW with high complexity and tight integration into HW. It scales to any number of peripherals and bitmap size. However, specification of input parameters, as discussed below, is practically limited by their presentation. If the number of interrupts and timers assume large values, their handling may become impractical.

C. *Inputs*

The verification framework supports definition of test specifications using Microsoft Excel. Table 1 includes an exemplary test specification.

Table 1 Microsoft Excel Table Specifying input and expected output parameters.

	A	B	C	D
1		**Inputs**		**Expected output**
2	**dsp1_algo_value**	**eep1_lower_limit**	**eep1_upper_limit**	**Limit_error**
3	-15	-25	25	0
4	-35	-25	25	1
5	…	…	…	…

Each row corresponds to the specification of a single test. The columns define I/O values for e.g. bitmap values. For example, row 3 defines the input parameters and expected output for the first test run. Once a FW run is complete, the framework will automatically transition to the next row, i.e., row 4, and start the next FW run. Multiple units may be defined. Each unit has its own worksheet in which its I/O data is defined.

The data may be inserted directly by the FW designer or retrieved from other tools like Matlab. Microsoft Excel was selected for multiple reasons. Firstly, it is well-known by designers. Secondly, reliable libraries are available to allow the framework to parse the file contents. Thirdly, reference implementations to obtain reference values can be implemented directly in Excel using Excel formulas.

Designers register FW units and their corresponding tests at the verification framework. The setup needs to be done only once. The framework is designed to automate common use cases while giving designers the flexibility to implement very specific test cases. Methods are implemented to easily map the test specification to actual bitmap fields. The example in the following section illustrates how to map the values.

D. *Test Bench*

The logger logs the execution process of the ISS including bitmap accesses, interrupts, etc. making the data available to the profiler and for verification. The profiler reports statistics on FW execution. ISO 26262 requires proper profiling and distinguishes statement, branch, and modified condition/ decision coverage (MC/DC). The report includes statement and branch coverage to determine test coverage. Additionally, it includes statistics on coverage of the individual tests, configuration parameters for reproducibility, and data for change management.

In order to support FW designers, the framework implements additional analyses of the application's resource consumptions. Register accesses of the FW are logged. Designers get a graphical overview which units access which memory addresses at what time. White-listing of addresses allows detection of memory access violations.

Additionally, specific hardware accesses can be monitored. For example, monitoring EEPROM accesses enables optimization of the code execution in order to reduce the number of paging processes. By logging information when, how often, and in which order a FW unit accesses EEPROM pages, the developer gets a quick overview and thus a suggestion on how to optimize the code to minimize EEPROM pagings.

E. *Verification Process*

The verification process is fully automated once setup and verification specification are complete. It supports regression. First, the environment is initialized by generating the bitmap from the bitmap specification and reading the test specification. For each test run the HW models, ISS, and bitmap are reset and reinitialized to ensure independence of executed tests. The ISS executes the image while the test bench logs the progress. Note that the image is identical to the version used on the DSP. It ensures that the verified version of the FW equals the version used in the final product.

Verification runs continue as long as I/O values are defined in the Excel verification specification. During post-processing the stored test results are compared and evaluated. Finally, the verification and profiling reports are generated and the verification process concludes.

V. Application Example

The following three examples demonstrate the ability of the verification framework applied on an architectural sub part of an IC. They are real-world examples from a recent project selected to demonstrate the working and the benefits of the verification framework.

A. *Example 1 – Timing Constraint*

Figure 2 illustrates relevant parts of the IC comprising two DSPs each connected by a separate digital bus to a common multiplexer handling access to a joint EEPROM.

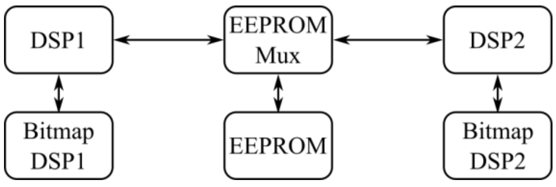

Figure 2 IC comprising two DSPs with a joint EEPROM.

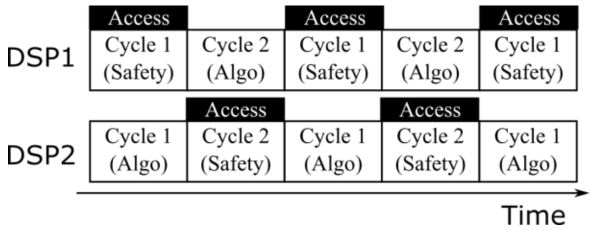

Figure 3 Scheduling with repetitive time slots.

The two DSPs operate independently. However, access to the shared EEPROM needs to be organized. Figure 3 illustrates a repetitive scheduling scheme of equally sized time slots allowing EEPROM access to either DSP1 or DSP2. In cycles 1 and 2 DSP1 and 2 get access to the EEPROM, respectively.

DSP1 and DSP2 execute diverse FW images of the same functionality. The FW has two tasks. Firstly, it implements an algorithm converting an input value to an output value. The computed output value is stored in the bitmap. Secondly, a safety check shall determine whether the computed output value is within valid limits. During the computation of the algorithm, the FW does not require access to the EEPROM. However, during the safety check, the FW requires data from the EEPROM to retrieve lower and upper limit of the safety interval. The FW tasks algorithm and safety can be split to match the time slots as indicated in Figure 3.

```
1   void safety_dsp1()
2   {
3     VERIFICATION_CMD(SAFETY_DSP1);
4     limit_check();
5     regbist();
6     VERIFICATION_CMD(SAFETY_END_DSP1);
7   }
8
9   void limit_check()
10  {
11    VERIFICATION_CMD(LIMIT_CHECK_DSP1);
12    // retrieve value from bitmap
13    int16_t value = dsp1_algo_value;
14    int16_t eep_lower_limit = GET(EEP1_LOWER_LIMIT);
15    int16_t eep_upper_limit = GET(EEP1_UPPER_LIMIT);
16
17    int16_t ll = eep_lower_limit << 2u;
18    int16_t ul = eep_upper_limit << 2u;
19
20    if((value < ll) || (value > ul))
21    {
22       SET_LIMIT_ERROR();
23    }
24  }
25
26  void regbist()
27  {
28    VERIFICATION_CMD(REGBIST_DSP1);
29    uint16_t reg0_read = 0;
30    uint16_t reg1_read = 0;
31    __asm__ volatile ("LD #123, R0");
32    __asm__ volatile ("ST R0, %0" : "=r" (reg0_read));
33    if(#123 != reg0_read)
34    {
35       SET_REGBIST_ERROR();
36    }
37    __asm__ volatile ("LD #123, R1");
38    __asm__ volatile ("ST R1, %0" : "=r" (reg1_read));
39    if(123 != reg1_read)
40    {
41       SET_REGBIST_ERROR();
42    }
43  }
```

Listing 1 C-Code FW Safety Mechanism

The scheduling of the FW is triggered synchronously to the EEPROM mux, i.e., once EEPROM access is given to DSP1, DSP1 and DSP2 are triggered synchronously to execute the safety and algorithm FW, respectively. Read access for DSP2 triggers the swapped FW functions.

Verification needs to ensure that the execution of the FW functions algorithm and safety for both DSPs never exceed the duration of the time slot. The FW of both DSPs can be verified independently simplifying the verification process. Note that the correct scheduling of the mux and the FW is done by HW, i.e., it is out of scope of FW verification.

During the safety time slot, the C-function `safety_dsp1()` is executed by DSP1 (see Listing 1 line 1). It includes the aforementioned limit check and a built-in self-test. Both functions are not relevant at this time and are discussed in the following examples. Execution of the safety function must never exceed the time slot length. Notice the VERIFICATION_CMD macros in lines 3 and 9. They implement special instructions allowing the verification framework to track execution progress of the FW. The instructions are marked volatile to prevent the compiler from rearranging or deleting them. A numeric constant is passed to the macro allowing unique identification. The instructions remain in the code after release to ensure the verified and the released FW image are identical. Using the markers ensures correct mapping of code to units. Tracking executed assembler may not allow unique assignment to units due to compiler optimizations, i.e., function calls may not be assignable to specific FW parts. Instead, the markers are transparent means to allow communication between the FW code and the verification framework. Designers have full control on setting boundaries of individual units. The added overhead in negligible.

The VERIFICATION_CMD macro triggers a callback from the ISS to the verification framework. Using the numeric constants passed to the macro, the framework can determine the interval length and a timing constraint can be defined. The framework will automatically flag a test run erroneous if the timing constraint is violated.

B. Example 2 – Test Setup

The safety function in Listing 1 calls the `limit_check()` and `regbist()`. Both functions are marked as individual units using the VERIFICATION_CMD macro. The macros indicate start of a new unit by passing the numeric constant to the framework using a callback. Upon their start, the bitmap is updated with its corresponding values defined in the Excel specification. A previously running unit is automatically evaluated. Consider the verification specification in Table 1 defining input and expected output values for the `limit_check()` unit. The values defined in the Excel spec need to be mapped to bitmap registers. Instead of automatically mapping values by register names, designers implement and register setup and evaluation functions which are automatically called by the framework (see Listing 2).

```
void limit_check_init()
{
  put("dsp1_algo_value", "A"); // signature: put(bitmap register name, Excel test specification column)
  put("eep1_lower_limit","B");
  put("eep1_upper_limit","C");
}

TestResult_t limit_check_eval()
{
  return is_equal("limit_error","D");
}
```

Listing 2 Methods to initialize and evaluate bitmap values from test specification.

The `limit_check_init()` function is executed once the limit check unit starts (triggered by an ISS callback). The `put()` functions implement the mapping of the Excel specification to the bitmap registers. They specify the bitmap register to be initialized and from which column in the Excel to retrieve the value. In the first iteration, the framework will initialize the bitmap register *dsp1_algo_value* with the value stored in cell *A3*, i.e., -15. The row is automatically tracked by the framework in each test iteration. For evaluation, the `is_equal()` function is executed. It retrieves the field `limit_error` from the bitmap and checks equality with the value specified in column *D* returning the corresponding test result.

Currently the framework supports value comparisons and timing constraints. Values are compared (equal, larger, smaller) between the expected and actual output values. Optionally tolerances may be defined. Convenience functions for common tasks reduce the test specification to a minimum. Additionally, users have the freedom to implement arbitrary test criteria such as combination of multiple test criteria, data evaluation over time, or timing constraints. Within the setup and evaluation functions, designers may access all data logged by the framework. The selected approach has proven itself to reduce effort for setup to an acceptable level while maintaining flexibility to implement arbitrarily complicated verification criteria.

C. Example 3 – Target HW Specific Behavior

The limit check in Listing 1 retrieves lower and upper limits from e.g. an EEPROM (lines 15+16) and checks if a passed value is within these limits (line 21). If the value exceeds either limit, an error flag is raised (line 23). To save memory in EEPROM, the limits are shifted (lines 18+19). Note that the limits are signed and require arithmetic shifts and not logical shifts. However, the usage of the correct shift depends on the compiler. The C-language does not specify the expected behavior but leaves it open for the compiler. The compiler for the target HW will translate the code correctly. A host compiler may behave differently. While this example will be handled correctly by most C-compilers, it demonstrates the limitation of host compilers with respect to special functionality implemented by a target HW compiler. Since target compiler and architecture are highly optimized w.r.t. each other, such language constructs allow efficient resource usage at the cost of generality. However, the FW is optimized for a specific HW and does not require platform independence.

Another example for HW specific behavior is the implemented assembler (Listing 1, ll31-32, 37-38). While usage of assembler should be reduced, some functionality can only be implemented in assembler. The register built-in safe-test in Listing 1 lines 26ff is such an example. Special patterns, here represented by the constant *123*, are written to the internal working registers and additional operations may be performed (omitted in Listing 1), and the result is read back for comparison to a reference value. The tests need to be performed on all internal working registers R0 and R1. If left to the compiler by implementing the functions in C, the compiler may firstly remove the operations for optimization; secondly it may not test all registers R0 and R1 but select any.

Special FW and HW features are implemented to guarantee the high safety standard required in automotive applications. Verification of such functionality cannot be achieved using host systems. In the absence of an ISS, verification needs to be delayed until the corresponding HW is available.

VI. Conclusions

We introduced a framework for automatic firmware (FW) verification with respect to functional safety focusing on automotive applications. Its usefulness was proven in a recent project in which the framework was used by multiple designers in parallel. By using an instruction set simulator, it is possible to decouple FW and hardware design by writing models for specific hardware, which provides an additional degree of freedom for FW developers. One key motivator is to be compliant with the requirements of ISO 26262, which describes the current state of the art development processes for safety critical, automotive applications. The fact that an instruction set simulator is used in combination with our framework creates the possibility of performing bit and cycle accurate test situations for FW interacting with very specific hardware components on a chip. Further, the FW can be fully tested on unit and integration level and thus two main verification points of ISO 26262 for software development are covered.

References

[1] ISO, *International Standard ISO 26262 Road Vehicles - Functional Safety,* 2011. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[2] V. B. Kleeberger, S. Rutkowski and R. Coppens, *Design & Verification of Automotive SoC Firmware,* ACM Digital Library, 2015. K. Elissa, “Title of paper if known,” unpublished.

[3] P. Cordemans, S. V. Landschoot, J. Boydens and E. Steegmans, "Test-Driven Development as a Reliable Embedded Software Engineering Practice," in *Embedded and Real Time System Development*, Springer, 2014, pp. 91-130.

[4] F. Brandner, N. Horspool and A. Krall, "DSP Instruction Set Simulation," in *Handbook of Signal Processing Systems*, Springer, 2010, pp. 679-705.

[5] K. Beck, Test Driven Development by Example, Pearson Education, 2003.

[6] D. Astels, Test Driven Development: A Practical Guide, Prentice Hall Professional Technical Reference, 2003.

[7] S. Heinen and M. Joost, "Firmware Development for Evolving Digital Communication Technologies," in *Hardware-dependend Software*, Springer, 2009, pp. 151-171.

[8] Q. Wei and M. Sharad, "Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation," in *Design,Automation and Test in Europe Conference and Exhibition*, 2003.

[9] T. C. Meyerowitz, *Single and Multi-CPU Performance Modeling for Embedded Systems,* University of California, Berkeley, 2008.

[10] M. Karlesky, W. Bereza, G. Williams and M. Fletcher, "Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns," in *Proc. Embedded Systems Conference*, 2007.

[11] S. Raffeiner, "Embedded Unit-Tests und Mocking mit CMock," Verifysoft Technology, Appenweier

[12] "RAKE - Ruby Make," [Online]. Available: http://rake.rubyforge.org/. [Accessed 15.03.2017]

[13] M. Poonawala, S. Subramanian, W.-T. Tsai, R. Mojdehbakhsh and L. Elliott, "Testing Safety-Critical Systems - A Reuse-Oriented Approach," in *Proc. Int. Conf. on Software Engineering and Knowledge Engineering (SEKE),* 1997.

[14] S. Hermann, D. Duerholz, R. Staerk and S. Kriso, SAFETY Essentials: ISO 26262 at a glance (E/E Engineering Essentials) (English Edition), KUGLER MAAG CIE, 2015.

[15] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel and B. Medina, "Architecture-Based Unit Testing of the Flight Software Product Line," in *Software Product Lines: Going Beyond,* Springer, 2010, pp. 256-270.