A Practical Approach to Measuring and Improving the Functional Verification of Embedded Software

Stéphane Bouvier, Nicolas Sauzède
STMicroelectronics, Home Entertainment & Displays
12 rue Jules Horowitz, Grenoble, France
{stephane.bouvier, nicolas.sauzede}@st.com

Florian Letombe, Julien Torrès
SpringSoft, Logic Verification Group
6 place Robert Schuman, Grenoble, France
{florian_letombe, julien_torres}@springsoft.com

ABSTRACT

We propose in this paper to apply functional qualification - based on the theory of mutation analysis - to firmware co-verification environments, by integrating the GNU Project Debugger (`gdb`) remote serial protocol (RSP) with the functional qualification engine. More specifically, the Certitude functional qualification tool is applied to the verification of an STMicroelectronics video IP. In this system, the hardware part is generated by High Level Synthesis flow tools mainly implementing the data-flow part, while the firmware part, written in C, is running on an STMicroelectronics embedded processor model devoted to the control part. Both hardware and firmware parts are simulated on a Transaction Level Modeling (TLM) platform with RSP access, modeling a virtual system-on-chip.

In this particular context, the embedded software is only accessible through the RSP. We describe how our functional qualification tool is extended to work in this environment. We present how the use of this technique allowed dead code in the firmware to be identified and to point out critical weaknesses in the verification environment. We show that fixing these issues led to reducing the memory consumption of the firmware and to finding critical bugs in the hardware/firmware. The standard nature of the RSP means that this could be a global technique to apply mutation analysis to embedded software.

Categories and Subject Descriptors

B.1.4 [**Control Structures and Microprogramming**]: Microprogram Design Aids—*Firmware engineering, Verification*; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

General Terms

Verification, Design, Experimentation

Keywords

Functional Qualification; Co-verification; Mutation Analysis; TLM/SystemC Virtual SoC; Embedded Software; `gdb` remote serial protocol.

1. INTRODUCTION

With the increasing complexity and reduced time-to-market of today's embedded systems there is a trend towards developing more of the system in software [1]. Debugging embedded software can be time-consuming and finding bugs late in the development cycle can be very costly. Verification of the software and especially the hardware-dependent software (firmware) can help to find hardware bugs and avoid re-spins [2]. As a result it is necessary to do hardware/software co-verification for these systems to find and fix problems as early as possible. Measuring how well these systems are verified is becoming more important since it can help to estimate and limit the risk of finding costly bugs.

For the purpose of measuring verification, code coverage tools exist, but can be shown to be insufficient in the qualification of a testing environment [2]. Mutation analysis based on fault injection [3] is broadly accepted as a stronger metric for measuring the quality of verification, and has been broadly adopted for Hardware Description Languages. Unfortunately, a code coverage tool or a mutation analysis engine is not necessarily well-suited to the verification of the firmware because of its embedded nature and the difficulty to communicate with these tools.

This paper presents an innovative method to perform embedded software functional qualification. It shows how a mutation analysis engine has been extended to be able to communicate with the firmware.

The paper is organized as follows. Section 2 presents general concepts about developing and debugging embedded software. Section 3 defines code and functional coverage and summarizes the main mutation analysis and functional qualification aspects. Section 4 presents functional qualification applied to embedded software. Section 5 shows the effectiveness of the proposed methodology in measuring the quality of embedded software verification environments of an industrial platform. Finally, the paper concludes and opens on future applications in Section 6.

2. DEVELOPING AND DEBUGGING EMBEDDED SOFTWARE WITH GDB

In this section, a quick overview of embedded software development and debugging is proposed, focusing on the GNU Debugger tool.

Embedded software development is usually based on a host-target approach: the embedded software is developed on a host machine, then transferred to the target machine for test and debug purposes. For the remainder of this paper, the embedded software is called the *target*, the system it is launched on (namely the platform) is the *target machine*, and the machine from which the platform is launched is the *host machine*. In the context of embedded software development, debugging is a crucial aspect. S. Schneider and L. Fraleigh [4] claim that 80% of development effort is spent on debugging.

The GNU Debugger, or `gdb` for short [5], is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix systems and works for many programming languages, including C. `gdb` provides extensive features for tracing, monitoring, and modifying the execution of programs. For example, the internal variables of the program can be checked and modified, the execution of the program can be interrupted via breakpoints, etc.

For the purpose of embedded software debugging, the GNU software system provides a tool called `gdbserver` that allows `gdb` to support remote debugging. `gdb` runs on the host machine while `gdbserver` is executed on the target machine. The execution of the program can be controled from `gdb` commands exactly as if it had been run directly on the host machine. `gdb` and `gdbserver` communicate via either a serial link or a TCP connection, using the standard `gdb` remote serial protocol (RSP). Note that only TCP connections are considered in this paper. The protocol is publicly available [6]. This information is essential for the portability of the techniques.

3. TEST METHODS

Several coverage techniques are considered in this section, and a definition of functional qualification, along with its differences with mutation analysis are proposed. Finally, the Certitude tool is presented.

3.1 Code Coverage and Functional Coverage

Software verification primarily utilizes code coverage to check if the verification is complete. Tools like GNU gcov and others provide statement coverage, call coverage, and branch coverage. Various companies have standards for code coverage that must be met before software is shipped. The main advantage of code coverage is simplicity, and results can be generated with little effort. However, code coverage tools require support by the target environment where embedded software is verified.

Functional coverage is a different technique from code coverage, and is more general [7]. Basically, functional coverage is the determination of how much functionality of the design has been exercised by the verification environment. This technique is typically used with pseudo-randomized test case generation [8]. If pseudo-randomization is being used, functional coverage provides an effective feedback mechanism for test scenarios as they are developed [9].

Again, the target environment needs to be adapted to support functional coverage tools such as FoCuS from IBM [10], BullsEye [11], or Comet [12].

3.2 Functional Qualification

3.2.1 Mutation Analysis vs. Functional Qualification

The main weakness of coverage metrics is that they do not consider the checking of output behavior of the design under verification (DUV). Indeed, it is possible for these metrics to give high scores even if the output behavior of the DUV is not completely checked. To address those problems, mutation analysis and mutation testing [13] have gained popularity in recent decades [14]. Such testing approaches rely on the creation of several versions of the program to be tested, "mutated" by introducing syntactically correct functional changes. These mutated versions of the program are called "mutants". The purpose of such mutations is to change the program to check if the test suite is able to detect the behavioral difference between the original program and the mutated versions. More specifically, the output of the DUV is compared with and without the mutation [15]. If there is a difference observed in the output then the mutant is considered to have been "killed" [3]. The effectiveness of the test suite is then measured by computing the percentage of detected mutations. Similar concepts are also applied in hardware testing to provide more effective test suites for the DUV: verification engineers use high-level fault simulation to measure the quality of test benches [16], and test pattern generation to improve fault coverage. In this case, mutations introduced in the hardware descriptions are referred to "faults" [16].

Functional qualification performed by the Certitude tool (first introduced in 2009 [17]) and discussed in this paper is different. A mutant is considered to have been killed when a test case fails. As in traditional mutation analysis, outputs of the design are still monitored. If all the test cases pass on a mutated version of the DUV, and a difference is observed in the output, this means that checkers are missing in the verification environment. Functional qualification highlights these missing checks. Such checks can include the comparison of expected output behavior and assertions monitoring the program's internal or external behavior. The ability of the verification environment to detect potential bugs is being measured whereas in traditional mutation analysis only the ability of the input sequences to propagate potential bugs to outputs is measured. The term functional qualification has thus been introduced to capture this concept of measuring the bug detection ability.

Verification is required to ensure the quality of the design code and this activity often consumes around 70% of the total design resources [18]. A large amount of code must also be created to implement the verification environment and errors may occur in the implementation. Errors in the verification environment can result in one of three situations:

- The test case fails: in this simplest case, the error in the verification can be found, assuming that the design is correct;
- The test case passes: in this case the test case may hide a real design bug;
- The test case is missing: typically due to a mistake in the test plan.

Functional qualification is a unique technology that identifies passing test cases as potentially hiding real design bugs.

It can also identify a wide range of missing test cases that other techniques cannot. For example, complex temporal sequences may be missing, preventing the effects of hidden bugs from propagating to outputs where they can be detected.

To be effective, functional verification must ensure that the DUV are shipped without critical bugs. To find a design bug, three things must occur during the execution of the verification environment:

1. The bug must be activated; i.e. the code containing the bug is exercised.
2. The bug must be propagated to an observable point; e.g. the outputs of the design.
3. The bug must be detected; i.e. behavior is checked and a failure indicated.

Traditional EDA technologies have focused on item 1, activating the bug. Techniques such as code coverage and functional coverage can help ensure that design code is well-activated. But they can neither guarantee that design bugs will be propagated, nor that the bugs will be detected by the checkers, such as assertions or comparison against a reference model.

3.2.2 *Certitude: a functional qualification tool*

Certitude is a functional qualification tool commercialized by SpringSoft [19]. It provides various front-ends at different levels of abstraction, including the VHDL, Verilog, and C languages. Its overall operation is summarized in this section.

Certitude automatically inserts bugs (also called faults) into the hardware or software models. The fault model contains various types of faults such as operator changes, dead assignments, forced conditions, etc. As an example, the following original code

$$\texttt{a} = \texttt{b} \mid \texttt{c};$$

generates two different kinds of mutations:

$$\texttt{a} = \texttt{b} \,\&\, \texttt{c};$$
$$\texttt{a};$$

Then Certitude determines whether the verification environment can activate the faulty code, propagate the effects to an observable point, and detect the presence of the fault. A known fault that can not be detected points to a verification weakness. If a fault can not be detected, there is evidence that actual design bugs would also not be detected by the co-verification environment.

Certitude operation falls into three phases:

1. The model analysis phase analyzes the design and generates a modified source code with faults injected (instrumented code);
2. The activation phase runs a complete regression and analyzes the behavior of the verification environment with respect to the faults;
3. The detection phase runs selected tests from the verification environment to measure the ability of the verification environment to detect the faults.

At the end of the qualification, faults are classified with the following statuses:

non-activated: the fault has not been exercised by the test suite;

non-propagated: the fault has been activated, but did not impact the outputs of the design;

non-detected: the fault has been propagated, but checkers did not notice the behavior change;

detected: the fault has been propagated, and at least one checker noticed a behavior change;

disabled by user: the verifier decided that part of the functionality did not need to be verified.

Subsequently, Certitude provides users with a complete report of the results in HTML format that highlights the problem areas. This is used to expose shortcomings and guide improvements in the environment to ensure that bugs do not slip through the process.

When integrating Certitude into a project environment, it is important to understand that it works on top of the simulation framework, and can make use of a batch system. Certitude is a point tool that does not require changes to the project environment itself. Only minor modifications to some scripts may be necessary. To adapt Certitude to the verification environment, it needs to have the following information and control: a list of all software models files that make up the system, the ability to recompile the (instrumented) source code, a list of testcase names, a script that can execute a testcase and return a pass or fail result.

4. APPLYING FUNCTIONAL QUALIFICATION TO EMBEDDED SOFTWARE: A PRACTICAL CASE

This Section describes the RSP-enabled Transaction Level Modeling (TLM) development platform, the Certitude tool and the proposed approach to functional qualification of embedded software.

4.1 The RSP-enabled TLM development platform

The main purpose of a TLM platform is to raise the abstraction level of a typical SystemC platform – which works at the signal-level – to the transaction-level. The goal is to abstract the implementation details of the interconnect by manipulating abstract data structures that represent the payload.

The power of TLM is that it allows the abstraction level to be adjusted by providing the flexibility to choose the data granularity: for example, between a simple data word of the size of the bus, to a video macro-block or even a full image.

The TLM development platform used for this experiment is a set of interconnected SystemC components such as memories, bus controllers, CPUs, etc. It can be used to model a Virtual System-on-a-Chip that simulates at the same time the hardware Intellectual Property (IP) and the embedded software running on it.

A TLM platform can embed as many CPU "cores" as needed by instantiating the corresponding number of TLM processor models. The TLM processor models are SystemC components that provide a particular CPU kind (e.g.: STxP70, ST40, ARM CA9, PowerPC, etc.).

The platform we used integrates an STxP70 (STMicroelectronics proprietary micro-controller) processor model that uses a technology that serves as a helper to accelerate the embedded software execution. It consists of compiling the real embedded software source code and executing it within the context of the running TLM simulation, i.e. with a direct access to the simulated hardware resources, such as IP registers, memories, interrupts, etc.

A very important feature of the processor model technology used in the TLM platform, is that it presents to the "outside" a standard RSP connection feature, to let a standard debug client manage the embedded software execution. This is the key point that enables the novel approach described in this paper to actually perform the functional verification of the embedded software.

4.2 Using Certitude on embedded software

STMicroelectronics has been using Certitude since its market introduction [20], and today runs functional qualification on more than 80 percent of its internal IP designs. In this context of historical collaboration, the Certitude team develops customer-specific features on demand when perspectives of such features for future deployment are promising. Since the Certitude tool is widely used within the company for RTL and C designs, STMicroelectronics naturally asked the Certitude development team to adapt the tool for their embedded software requirements.

In the current configuration, Certitude injects faults in the design and communicates with the simulation via control files. All faults are injected together in the design to avoid re-compilations, and control files are used to set the simulation parameters: fault to exercise, result of the simulation, etc., as described in [21]. This method can't be directly applied to embedded software testing environments since the target machine does not necessarily have a file system, and the code injected into the software controls its behavior, and requires several Linux system calls also not likely to be supported by the target system. As a result, using Certitude with its default behavior would result in compilation errors and an inability to communicate between the tool and the target.

A traditional use of a TLM platform that embeds a processor model which offers a standard RSP interface is to connect a `gdb` client to the target firmware that runs inside the platform. Here, for the purpose of functional qualification, we reused this standard RSP interface to connect the Certitude tool directly to the firmware execution. The framework has

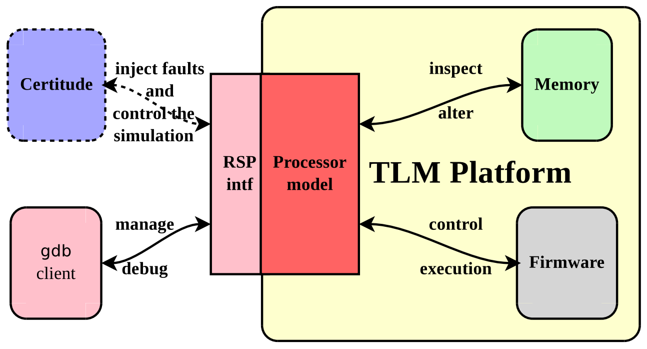

Figure 1: Framework Setup

been setup as shown in Figure 1. The new usage path is drawn with dotted lines.

The Certitude tool behaves as a standard `gdb` client, benefiting from the same debugging functionality, such as: controlling execution (stop and resume), inspecting/altering memory contents, setting breakpoints, etc. As described in section 2, the RSP is generally used for debugging purposes, and to the best of our knowledge, it has never been used before in the context of mutation analysis.

As stated above, qualification of embedded software gives rise to two main issues; namely (1) communication, and (2) compilation.

The first issue is solved by the mechanism described in Figure 1. The fault to be exercised for a simulation can be set by sending the information via the RSP interface, and the simulation result can be checked the same way.

The second issue is tackled by simplifying the instrumentation of the design. Only standard variable types are used, and no system call is invoked, nor is there any file access from the instrumented code. This ensures full compatibility with any target system.

Finally, a finite state machine has been setup to wrap the platform execution and control the simulation status. Its operation consists of: launching the platform and establishing communication, parsing the target Executable and Linkable Format (ELF) code, setting the fault to be exercised (for the detection phase), authorize the target to be launched, check its behavior, get results and close communication. This complex mechanism guarantees correct execution and termination of simulations.

5. EXPERIMENTS

This section describes the machines and the use case considered for experiments. We give details of experimental results, along with improvements carried out on the considered IP.

5.1 Methodology

The experiment to functionally qualify the embedded software testing environment has been done on a High Quality Video Display IP used in the set-top-boxes. This IP gets some video streams from previous design stages (such as H264 video decoder) and sends them to the next stages (picture composition). The input streams are processed using de-interlacing, rescale, and Image Quality Improvements algorithms.

The High Quality Video Display IP used is made of two parts. First, a hardware part is generated by an HLS (High Level Synthesis) flow tool mainly implementing the dataflow part; second a software part running on an STxP70 core is devoted to the control part. This code is known as the embedded software that we want to instrument and cover in this paper.

This embedded code is mainly a control part of the hardware design. It deals with the start conditions, reads a command in memory and processes it. It then configures the IP internal registers and cadences the hardware block inputs and outputs. This leads to high number of interactions with the hardware.

Usually, the hardest parts to verify in a design are the state machines. According to this new architecture, these state machines are mainly located in the embedded software. Having good coverage on the embedded software that implements a significant part of the control is key to finding verification holes that may hide some hardware bugs. Before this qualification by error injection in the embedded software, there was almost no coverage analysis performed on these software control state machines. Indeed, the only possibility we had was to trace the program counter of the STxP70 core and analyze the never reached values.

The platform environment used to perform the functional verification is a SystemC TLM virtual platform where both CPU's (host and embedded micro-controller) are replaced by a processor model allowing execution of the software under test on a Linux PC.

The test suite used for the High Quality Video Display IP is made up of 10,434 test cases. This test suite runs in TLM in about 4 hours using 200 Linux CPUs, and has been used with the Certitude tool. The empirical results presented in this section were obtained on the 200 Linux CPUs.

The hardware has been modeled in SystemC TLM too. The error injection using Certitude has already been made on this SystemC TLM model. The current experiment is expanding the concept to the embedded software.

5.2 Results

This embedded software is made up of 27 files. The Certitude model phase injected 6781 faults.

We disabled 592 of the injected faults. The TLM platform which uses a processor model to emulate the STxP70 micro-controller is a pure SystemC platform with untimed models. As the IP is designed to cope with real-time constraints, the embedded software contains some dedicated code to deal with real-time constraints. As the SystemC platform is an untimed execution one, this platform can't exercise real-time related mechanisms. To reach them, we need a timed view of the hardware. This is done later in the verification process. For this experiment, we disable it to avoid uninteresting non-detected faults.

The Certitude activation phase has been run using this complete test suite. This leads to the first results in about one night: 59 faults non-activated and 116 non-propagated. This 59 non-activated faults result is equivalent to line coverage: such faults represent lines of code never reached by any of the tests. The non-propagated faults result gives us additional information. For example, details are provided on the "if" statements partially reached ("if" conditions always true or always false) and on the writes to variables that don't change their values.

Running a full detection phase of the Certitude tool with this full test suite would be too long. For each exercised fault, the tool will launch every test that activates this fault until it detects it or all tests have been run. In the worst case, this may lead to about 6000 times the regression which is beyond the time available for the project.

To get an overall idea of the detection results, we first ran a statistical metric provided by the Certitude tool. It randomly selects a sample of faults and runs randomly chosen tests on each fault of the sample. This results in an estimation (with a margin of error) of the number of detected and non-detected faults in a much shorter time than a full detection. After running for one night, this statistical metric gave an estimated number of detected faults of about 5200 or, put another way, about 700 non-detected faults.

The statistical metric only gives an estimate of the number of non-detected faults, but it can not find which faults are not detected. To optimize the run time of the detection phase, with Springsoft support, we identified a small sub-set of tests based on the activation results. In our case, this sub-set of tests only contains 40 test cases. We thus ran the detection phase with this short test suite and then got a first result of 713 non-detected faults in about one night. This result can be considered "pessimistic" because a non-detected fault may have been detected by another test case that is not in the short test suite. Consequently, we needed about two additional weeks to extend the sub-set of test cases and thus detect some of these remaining faults. Adding these tests reduced the number of non-detected faults to 694.

To summarize, after all these steps, we obtained the following result status:

- 59 faults non-activated (0.9%),
- 116 faults non-propagated (1.7%),
- 694 faults non-detected (10.2%),
- 5320 faults detected (78.5%), and
- 592 faults disabled by the verifier (8.7%).

At this stage, we started to analyze why these faults were not detected, keeping in mind that we have not run all the tests on the non-detected faults. So we analyzed the Certitude report from the activation and detection phases. Activation first showed that there was some dead code that cannot be activated. This code has been removed and it saved about 2% of room in the program memory. This is a very good result since the embedded control code needs to be optimized to fit on small memories. Activation showed us that some tests were missing. We then wrote 4 additional tests to cover some lines. These tests are mainly checking functionality that was missed by the previous test suite. One of these tests showed a hardware bug in a corner case and avoided a re-spin of the silicon.

This hardware bug concerned one mode of our High Quality Video Display IP called panoramic mode and, more precisely, one specific option of this mode. The first specifications of the IP did not mention this possibility which appeared later. Unfortunately, it had never been added to the test plan. The testcase generator was able to generate this case but none of the tests were using this possibility. The verification hole was found analyzing a fault in an "else" branch of an "if". When the new tests implementing this feature were written, we found that the development version of the RTL part of the design behaved incorrectly.

Detection also highlighted missing tests. We then wrote 6 new tests to detect the non-detected faults. These tests did not show new bugs. The tests written after the detection phase are mainly checking initialization of the embedded software between two commands in some corner cases. Thanks to these new tests the functionality of the IP is better verified and so far no bugs have been found in the production version of the IP.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a framework for the functional qualification of embedded software testing environments, and showed its application on an industrial design. Nowadays, (i) the close integration between hardware and software parts in modern embedded systems, (ii) the development of high-level languages suited for modeling hardware and software, (iii) the need for developing verification strategies to be applied early in the design flow, require the definition of mutation analysis-based strategies that work at system level, where hardware and software functionality are not completely independent and separate.

We already knew that Certitude was a very efficient tool to qualify verification environments (this tool is already deployed on hardware verification projects) when running on RTL or on the C standalone models used in the HLS design flow. Extending the usage of this tool to hardware control implemented by software is a must to avoid critical hardware bugs.

We have run Certitude on the embedded software of our High Quality Video Display IP. It has allowed us to remove some dead code and to add some missing tests. Some of them have uncovered a hardware bug and prevented a respin of the silicon. It hence allowed STMicroelectronics to keep delivering quality-products to the market.

Experiments focused on the GNU `gdb` remote serial protocol, for its simplicity and the fact that it is a proven, defacto industry standard. However, the principle described in this paper could in fact be deployed with any embedded system including an external debug interface (for example ARM Cycle Accurate Debug Interface (CADI) [22], Lauterbach TRACE32 [23], Power Standard for Common Debug Interface (CDI) [24], etc.), provided that the debug protocol is standard/open. The only requirements are to be able somehow to manage the embedded software execution, and access the target memory, registers, and breakpoints. This is future work to extend the Certitude tool in this direction.

7. REFERENCES

[1] E. A. Lee, "Embedded software," *Advances in Computers*, vol. 56, pp. 56–97, 2002.

[2] G. D. Guglielmo, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, "On the functional qualification of a platform model," in *Proc. of the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 182–190.

[3] A. J. Offutt, "A practical system for mutation testing: Help for the common programmer," in *Proc of IEEE International Test Conference*, 1994, pp. 824–830.

[4] S. Schneider and L. Fraleigh, "The ten secrets of embedded debugging," EETimes Design, 2004, http://www.embedded.com/showArticle.jhtml?article ID=47208538.

[5] "GDB: The GNU Project Debugger," http://www.gnu.org/software/gdb.

[6] GNU Remote Serial Protocol, http://sourceware.org/ gdb/current/onlinedocs/gdb/Remote-Protocol.html.

[7] G. J. Myers, *The Art of Software Testing.* Wiley - Interscience, 1999.

[8] P. Mishra and N. Dutt, "Functional coverage driven test generation for validation of pipelined processors," in *Proc. of the conference on Design, Automation and Test in Europe*, 2005, pp. 678–683.

[9] M. Hampton, *Functional qualification: a technical brief*, 2009, http://www.edadesignline.com/215600203.

[10] Focus, http://www.alphaworks.ibm.com.

[11] BullsEye, http://www.bullseye.com.

[12] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User defined coverage - a tool supported methodology for design verification," in *Proc. of the 35th Conference on Design Automation (DAC'98)*, 1998, pp. 158–163.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE computer*, vol. 11, no. 4, pp. 34–41, 1978.

[14] D. Hyunsook and G. Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," *IEEE Transaction on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[15] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert, "Benchmarking Testing Strategies with Tools from Mutation Analysis," in *IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 360–364.

[16] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design.* New York: Computer Science Press, 1990.

[17] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton,

and F. Letombe, “Functional Qualification of TLM Verification,” in *Proc. of the conference on Design, Automation and Test in Europe*, 2009, pp. 190–195.

[18] J. Bergeron, *Writing Testbenchs: Functional Verification of HDL Models.* Kluwer Academic, 2000.

[19] Certitude from SpringSoft, http://www.springsoft.com/products/functional-qualification/certitude.

[20] O. Haller, “Deploying Functional Qualification at STMicroelectronics, Methodologies & Case Studies,” IP&Design, Functional Verification Group of the Conference on Design Automation, 2008.

[21] M. Hampton, “Procédé et Système d’Évaluation de Tests d’un Programme d’Ordinateur par Analyse de Mutations,” French Patent FR2873832 for Certess SARL, 2006.

[22] CADI, http://infocenter.arm.com/help/topic/com.arm.doc.dui0444d/index.html.

[23] TRACE32, http://www.lauterbach.com/tutorial.pdf.

[24] CDI, https://www.power.org/resources/downloads.