Did We Test Enough? Functional Coverage for Post-Silicon Validation

Sebastian Pointner Robert Wille
Institute for Integrated Circuits
Johannes Kepler University Linz, Austria
Email: {sebastian.pointner, robert.wille}@jku.at

***Abstract*—The ever increasing complexity of modern systems remains a challenge for semiconductor companies. Once a new chip has been produced, it has to be ensured that it works properly. To this end, sophisticated test environments and test programs are applied. However, to ensure that the applied test program indeed fully covers all important details of the produced chip remains a big challenge. In this work, we propose a methodology which supports the designer by analyzing the coverage of a given test program. To this end, we utilize accomplishments from coverage analysis for functional verification at other abstraction levels. A discussion of the resulting application scenario eventually shows that this allows for an efficient coverage analysis for test programs with basically no changes in the work-flows of test program developers.**

I. Introduction

The triumphal process of embedded systems seems unstoppable. While already being an essential part of our everyday life, also upcoming applications in domains like Autonomous Driving Vehicles or Smart Cities underline the impact of embedded systems in our world. Within the last decade, their complexity could be strongly enhanced (c.f. Moores Law [1]) which enabled the usage of embedded systems for a series of entirely new applications. Due to the ever increasing complexity of modern systems, designing and especially ensuring that the new system works as intended (i.e. verification and test) has become a very hard task. Ensuring the correct behavior of a new system is of uttermost importance since an unexpected behavior may end up in dangerous situations for humans (e.g. with self driving cars) or may rise enormous costs (c.f. Ariane V crash [2]).

In order to cope with the complexity of modern systems, an elaborated design flow emerged in the past [3]. The design flow as sketched in Fig. 1 illustrates the main steps composed of the implementation of a new design, its functional verification (conducted pre-silicon), as well as its test (conducted post-silicon). The design flow starts with the specification of the new system which is getting used in order to implement the initial algorithmic representation of the desired system at the *Electronic System Level* (ESL). This allows to conduct simulations and, by this, provides the basis e.g. for hardware/software partitioning. After the respectively resulting software/hardware partitions have been extracted, the implementation of the respectively needed *Application Specific Integrated Circuits* (ASICs) starts – first conducted at the *Register Transfer Level* (RTL) and synthesized afterwards to a gate-level netlist. Eventually, final design steps, namely placement and routing, are performed based on the gate-level netlist.

Before the resulting descriptions of the new ASIC can finally be sent to the wafer fabrication, a verification process is conducted. Main task of the verification is to ensure that the obtained description meets the specification in every detail and that no unexpected behavior of the new system is possible. After design and verification have finally been completed, the resulting descriptions are eventually sent to a wafer fabrication which realizes the desired chip. After the first samples of the new ASIC have been produced, it has to be ensured that those samples are working properly – motivating the so-called *post-silicon validation* process.

Here, the fabricated chip (usually called *Device under Test*; DUT) is employed within an *Automatic Test Equipment* (ATE) which allows to apply dedicated test stimuli to the chip [4], [5]. The stimuli are thereby generated e.g. directly from the user or are e.g. generated from other components, sensors, etc. Vice versa, the ATE provides an interface which maps the output signals produced by the DUT to the corresponding user outputs, actuators, etc. Moreover, the user's view is limited to the ATE for testing the DUT, as the ATE represents an entire system. Overall, this allows to test the DUT using different scenarios and considering internal as well as external inputs such as sensor data, prepared data to emulate specific use cases (e.g. corner cases), or random test data to observe the system's behavior.

To automate these tests, usually a *test program* is applied [4], [5]. A test program represents a collection of different test methods – each of which used to test a different part, aspect, or scenario of the DUT. Although not needed before first silicon is available, the development of such test programs already starts earlier in the design process as a parallel task (c.f. Fig. 1). However, this task is non-trivial. To some extent, a good test program requires a re-consideration of all major aspects considered already during the design, implementation, and verification conducted before to make sure that all properties and characteristics of the DUT are properly tested [6].

This obviously rises the question whether the test program is covering all those important aspects of the DUT. For the case that the test program is not covering all characteristics of the DUT, it is very likely that badly produced chips will be delivered to the customers. Unfortunately, it is not possible to conduct measurements inside of the DUT, therefore it can not be classified which parts of the DUT have already been tested based on measurement. This causes a bottleneck in the design flow since coverage is, thus far, evaluated solely on the basis of experience of the involved design team [4].

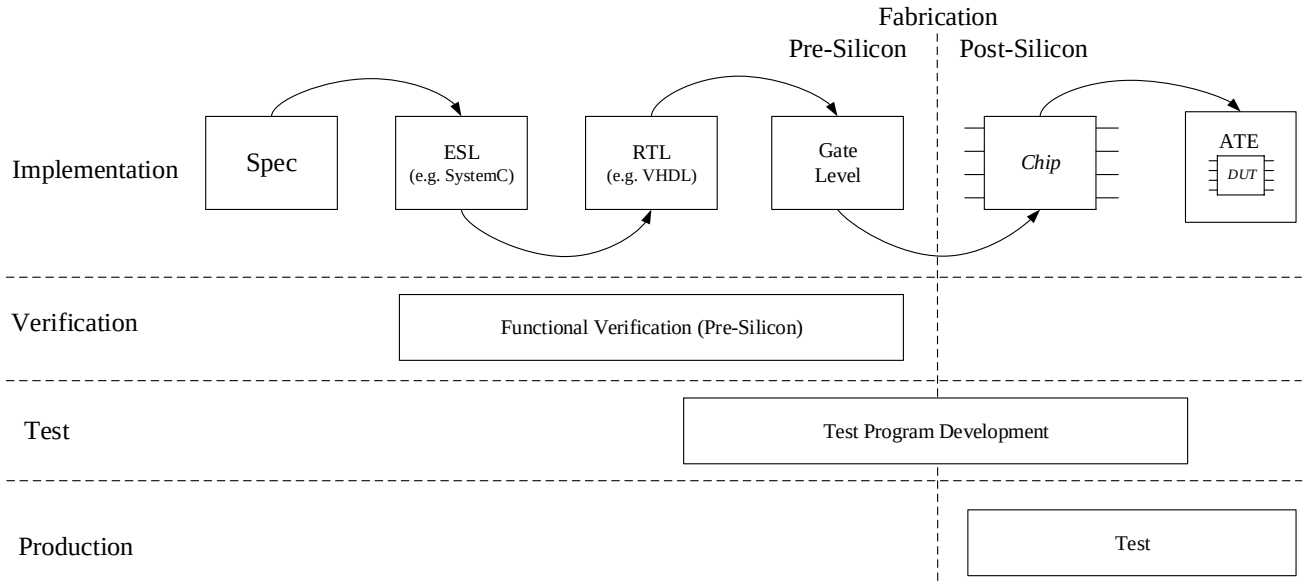

Fig. 1: Design flow.

In this work, we aim to address this problem by proposing a framework for the analysis of the functional coverage of test programs. To this end, we first discuss the problem and its importance during the test program development. Then, we review how coverage is guaranteed in other abstraction levels, particularly during functional verification where coverage analysis has heavily been investigated in the past. Based on that, a methodology is derived which utilizes these existing solutions for coverage analysis for functional verification and adapt them so that, additionally, also test programs can accordingly be analyzed.

A discussion of the resulting application scenario eventually shows that this provides a promising solution for the problem motivated above. In fact, the proposed solution allows to analyze the coverage of test programs with the same efficiency as coverage of the functional verification process has been evaluated before and with basically no changes in the work-flows of test program developers.

The remainder of this work is structured as follows: The next section briefly reviews the development of test programs and the correspondingly used environment. Afterwards, the problem of test program coverage analysis is motivated and, based on existing accomplishments for coverage for functional verification, a corresponding solution is proposed in Section III. Details of the realization of this solution are provided in Section IV. Finally, Section V discusses the resulting application scenario and Section VI concludes the paper.

II. Background

In order to keep this work self-contained, this section briefly reviews the development of test programs as well as the basic architecture of an *Automatic Test Equipment* (ATE). This infrastructure is partially re-used later to conduct the desired coverage analysis.

For the automation of the post-silicon test process, modern ASICs are tested by applying so-called *test programs* [4], [5] on the *Device under Test* (DUT). To this end, an ATE as sketched in Fig. 2 is utilized in order to execute the test program. The ATE executes the test program by invoking a various number of different *instruments* such as a power supply to power up the DUT or a simple measurement unit to observe a pin's voltage. The respective commands are realized over a direct connection to the DUT. Moreover, the test program controls the instruments which are embedded into the tester to execute the intended test scenarios. To this end, test programs are realized as part of an *ATE front-end* which sends it's requests to an *ATE back-end* over a static communication link, e.g. a *firm bus connection*. Afterwards, the ATE forwards the requests to the particular tester instruments, which can directly access the DUT.

The design of a test program depends thereby on the applied technology as well as on the requirements given by the respectively used ATE supplier. Test programs, as they are widely used within industry, are realized using frameworks based on programming languages such as Visual Basic or C++. Such frameworks extend the basic functionality of those languages with advanced support which is needed for industrial semiconductor testing. Examples for extensions can be found in class libraries (e.g. for the evaluation of tests) or additional tools (e.g. visualizations of instruments). Based on those extensions, a comprehensive framework for the entire test program development cycle, as well as for the test process in the wafer fabrication is supported by the ATE supplier [7].

The design of the test program can be conducted in parallel to the actual design of the considered ASIC (c.f. Fig. 1). The goal is the realization of a program which (1) automatically applies several tests on the eventually realized ASIC (after a first silicon realization of it is available) and (2) checks weather the intended outputs are obtained.

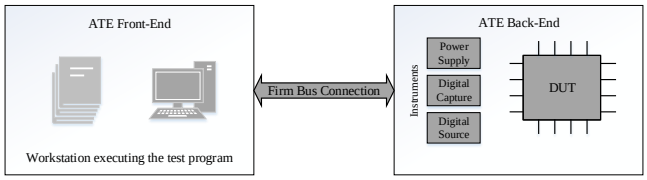

Fig. 2: Automatic Test Equipment (ATE).

Example 1. *Consider the realization of an* Arithmetic Logic Unit *(ALU) to be tested which possesses over a parallel interface and handles data with a bit-width of 8. The ALU supports the four basic arithmetical operators (addition, subtraction, multiplication, and division). During post-silicon validation, the chip has to be tested with respect to both, functional (correct execution of the arithmetic) as well as non-functional parameters (e.g. expected power consumption). For the test process itself, the test program generates particular test stimuli which are getting applied to the DUT and, afterwards, their response is getting analyzed. A typical test for the considered DUT (i.e. the ALU) could e.g. be the execution of all four arithmetic operations.*

III. Motivation & General Idea

In this section, we first motivate the need for methods for coverage analysis of test programs. Afterwards, we discuss how coverage is currently analyzed on other abstraction levels of the design flow. This eventually yields the general idea which is investigated in this work, namely conducting coverage for post-silicon validation by utilizing existing (corresponding) schemes from those abstraction levels.

A. Motivation

Post-silicon validation utilizing an ATE and test programs as reviewed in the previous section is an established method which is used in order to check whether first silicon of the design (i.e. the DUT) indeed has been fabricated as intended [4], [5]. But while test programs and the corresponding DUT communication flow enabled by the ATE usually provide a powerful infrastructure to thoroughly conduct these tests, uncertainties frequently remain about whether the DUT has indeed *completely* been tested or whether some corner cases have been missed.

Example 2. *Consider again the scenario of the ALU and the discussion of a corresponding test program conducted in Example 1. While, using the test program sketched there, the designer can validate e.g. the correct realization of basic calculations. However, the behavior of the DUT for non-obvious cases which go beyond a straight-forward application e.g. of the four arithmetic operations are frequently missed. For example, ALUs usually come with a zero flag or a carry flag whose test for correct functionality can easily be forgotten.*

Now, a test program designer of course can easily add further scenarios to consider into the test program. But, in this process, the main challenge is not the addition of further cases to be considered, but to become an understanding in the first place what cases presumably may have been missed. Of course, in a naive approach to completely test a chip, the test program can be extended to cover every possible state of the chip at least once – obviously leading to complexity issues and becoming infeasible already for very small devices. Beyond that, however, very few methods (e.g. [8]–[10]) for post-silicon validation exist yet which address the simple but rather serious question: *Did we test enough?*

In this work, we are addressing this problem by utilizing existing methods for *coverage analysis* from other abstraction levels of the design flow. To this end, we first briefly review those schemes in the following and discuss how to utilize them in order to address the problem motivated here. Based on that, the remainder of this work describes how to realize these ideas towards a coverage analysis for post-silicon validation.

B. Coverage in Functional Verification

Functional verification as a pre-silicon task and testing as a post-silicon task both aim for ensuring that the chip finally behaves as it is supposed to behave. However, the methodologies which have emerged for functional verification are far beyond what is possible nowadays in post-silicon validation. In fact, verification engineers command over a wide range of tools and methodologies which can be applied in order to observe the designs behavior and to ensure that the design has been comprehensibly verified [11]–[13]. While the post-silicon validation of the new design is entirely based on simulation, design verification engineers can apply simulation-based verification as well as formal verification techniques. Simulation based verification techniques like *Constraint-Random-Verification* (CRV) [14], [15] are applied in functional verification in order to achieve highest possible verification coverage [16]. Since simulation-based verification approaches are often not sufficient standalone, formal verification techniques such as formal equivalence checking [12], model checking [17], or symbolic execution [18], [19] are applied for the functional verification.

Here, a similar problem as sketched above emerges: Did we verify enough? But in contrast to post-silicon validation, coverage issues, i.e. how to make sure that a design has completely been verified, was intensely been investigated in the past years (see e.g. approaches proposed in [11], [12]). Here, functional verification benefits from the fact that verification can be seen as a white box process (i.e. the implementation can be used in order to obtain coverage information) compared to the post-silicon validation which is based on a black box process (i.e. the implementation is not accessible, only the inputs and outputs are usable).

In the following, we focus on coverage metrics as reviewed e.g. in [16] for structural coverage information as well as coverage metrics for assertion coverage or functional coverage [11] – both providing good representatives for measuring the coverage.[1] Structural coverage metrics focus on the structure of the design in terms of how the system/circuits has been executed. In contrast to that, functional coverage metrics focus on the application of the design itself.

[1] Note that the concepts proposed in this work can also utilize other coverage metrics.

Basic structural metrics are e.g. (1) line-coverage which gives the verification engineer feedback if a particular line of the code has been executed or not, (2) branch coverage which gives the verification engineer feedback if a branch has been taken during an execution or not or, (3) toggle coverage which monitors if every single bit of a signal has been at least flipped once. While those metrics are easy to obtain for a given circuit, they are obviously rather simple and, hence, limited in terms of their applications. In contrast, assertion coverage monitors the status of particular signals or internal conditions and gives the verification engineer feedback what percentage of those assertions could have been positively evaluated. While assertion coverage does already takes care about conditions, functional coverage is directly based on the functionality of the new design as it is specified in the specification documents. Moreover, by applying functional coverage it is possible to directly map use cases as described in the documentation and observe if these cases have been executed or not.

Example 3. *Consider again an ALU design. As discussed above, they usually come with commands where a zero-flag states whether a result yielded zero and a carry-out flag states whether the calculation yielded an overflow (i.e. the result can not be represented with 8 bit anymore). In order to evaluate e.g. whether such an overflow case has been triggered at least once, a designer can define a metric where ld(input_a + input_b) $>$ 8 has to be satisfied. Corresponding coverage methods as reviewed above can keep track about what has been verified yet and whether this particular case has already been covered or not.*

All of the discussed coverage metrics are capable for delivering coverage information and are widely applied within the industry. In the following, we are now discussing how those accomplishments in functional coverage can be utilized to also be able to evaluate the coverage of a test program and, by this, the coverage of the post-silicon validation process. We are assuming thereby that, during the design process, the considered system has been completely verified (with *completely* being defined by the designer e.g. through corresponding metrics as discussed above).

C. Utilizing Functional Coverage for Post-Silicon Validation

Utilizing coverage metrics as discussed above to evaluate whether a test program for post-silicon validation indeed completely covers the considered DUT sounds promising. However, a direct application of the solutions employed in higher levels of abstraction on a functional design description to a first silicon DUT integrated within an ATE as reviewed in Section II is a non-trivial task.

In order to obtain coverage information for test programs in a similar fashion as conducted during functional verification, first the actual DUT is replaced by an alternative representation of the design. Here, utilizing the correspondingly used descriptions from the higher abstraction levels (e.g. a SystemC or SystemVerilog description which is available anyway) is an obvious choice[2]. But besides that it is necessary to establish a link between the test program and the verification environment.

[2]In fact, the same idea has been applied for testing test programs even before first silicon is available (see [6]).

This is because, as reviewed in Section II and illustrated in Fig. 2, the test program is not directly communicating with the design but with a bus connection. Hence, the communication flow between the test program and the verification environment has to be re-arranged accordingly. Moreover, the instruments which are getting controlled by the test program and are eventually controlling the DUT have to be modified in order to forward the generated stimuli to the verification environment rather than the DUT.

At the other end of the communication channel, i.e. at the verification environment, the commands which are sent from the test program have to be processed accordingly as well. Moreover, when the test program is applying a particular value to a pin, the stimuli generating part of the verification environment has to do the same. Every time when the test program would process an interaction with the DUT, this interaction has to be routed over the verification environment realized in terms of an identical behavior onto the design.

Once all this is established, we can obtain a fully configured environment for functional coverage analysis for free by simply taking the existing state of the art reviewed in Section III-B.

IV. Realization

In this section, we describe how the general ideas discussed above and, by this, how a coverage analysis framework for post-silicon validation can actually be realized. To this end, we utilize the same idea which has recently been proposed for testing test programs before first silicon is available (see [6]). Here, a DUT is represented by a SystemC description (which is available anyway in the considered development flow as seen in Fig. 1). This methodology does not only allow for testing a test program but, additionally using frameworks such as the *Universal Verification Methodology* (UVM, [20]), for analyzing the coverage of a test program.

In the following, details of the proposed solutions are presented. First, the UVM framework is briefly reviewed. Based on the structure of the UVM, we are then going to connect the test program in such an order that the test program controls the verification environment and can actually execute all tests and instrumentations so that the coverage tools from the verification environment can eventually analyze whether the test program is "complete".

A. The Universal Verification Methodology (UVM)

The *Universal Verification Methodology* (UVM, [20]) has become a well established methodology for design verification [21]. The UVM which has basically been introduced within the SystemVerilog *High Level Verification* (HLV) language provides all the infrastructure which is needed for a comprehensive design verification [12]. In order to create a powerful verification environment, UVM is based on the idea of reusable components and an object-orientated design. Moreover, UVM provides the needed basic classes which can directly be extended using object-orientated derivation.

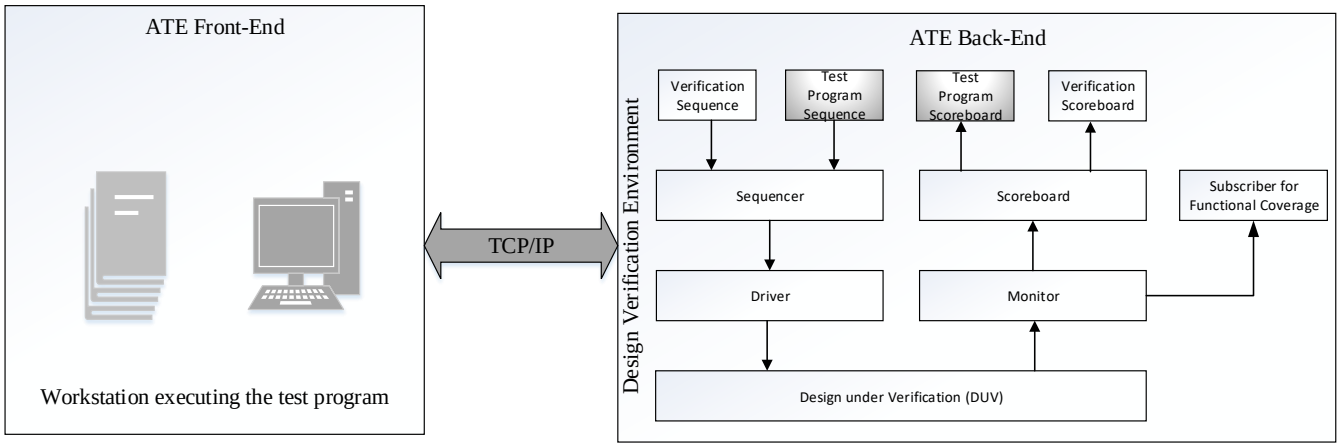

Fig. 3: Realization of Coverage Analysis for Test Programs.

A simplified version of the UVM as it is used in this work is sketched at the right-hand side of Fig. 3[3]. As can be seen in the figure, the model to be verified, (here usually called the *Design under Verification*, DUV) is embedded into the verification environment and can be assessed writable from the *driver* and readable from the *monitor*. The driver gets the stimuli to be applied from the *sequencer* which gets a *verification sequence* and executes it. In order to verify the behavior of the DUV for particular stimuli, the *monitor* reads the response which, afterwards, is getting evaluated by the *scoreboard* using a corresponding *verification scoreboard*. Furthermore, so-called *Subscribers* can be used in order to collect additional information. Referring again to Fig. 3, a subscriber is used in this figure in order to collect functional coverage information.

B. Connecting the Test Program to the UVM

Having the UVM as basis, the desired coverage environment for test programs can now be realized. More precisely, the proposed environment is composed of two components as shown in Fig. 2: (1) The test program which is used to generate the stimuli and to check the corresponding behavior of the chip (sketched at the left-hand side of Fig. 3 and basically identical to the ATE front-end reviewed before in Section II and Fig. 2) as well as (2) the chip itself which, as described above, is now integrated into the UVM (as sketched at the right-hand side of Fig. 3).

What is left, however, is the connection between both components (recall, the test program and the front-end are are still generating and receiving data for/from the ATE back-end). Hence, in order to embed the test program's stimuli requests for reading or writing a certain pin to the UVM, we added a TCP/IP connection between the existing front-end and the newly added verification environment. To this end, we needed to add particular sequence and scoreboard classes which receive and apply requests from the test program which covers the test programs read request, respectively. This can be realized by extending the UVM with according classes as illustrated by the shaded boxes in Fig. 3. They basically provide an interface to the TCP/IP connection and, by this, to the test program (c.f. [6]). All the other classes (including e.g. the subscribers for functional coverage) can remain unchanged and, indeed, used for coverage analysis (but now for the stimuli generated by the test program).

Finally, it has to be ensured that the instrumentation of the test program is correctly translated. Recall that the test program can explicitly call certain instruments of the ATE back-end which are not available in the UVM environment anymore. However, all these instruments can be emulated here by simply translating the respective instrumentation to corresponding stimuli[4].

More precisely, instead of sending the original requests (e.g. read or write pin) to the ATE back-end and, by this, to an instrument, we altered the communication to the verification environment. If the test program now applies any operation (e.g. sets the opcode of the ALU to 0x3), this request gets accordingly translated to the proper verification sequence of the UVM environment and will be properly applied to the DUV.

V. Application Scenario

The coverage analysis method for test programs introduced above has prototypically been implemented as described in Section IV. To this end, UVM-SystemC [22] has been utilized as verification environment and CRAVE 2.0 [14] was used to implement the functional coverage analysis[5]. In this section, we finally sketch the application of the resulting analysis environment using again the ALU considered before as an example.

First, we assume that the ALU has been designed in a similar fashion as discussed before by means of Fig. 1. For the functional verification, also UVM-SystemC [22] has been

[3]Note that the shaded boxes do not belong to the original UVM framework but have been added in order to realize the coverage analysis for test programs. This is described in the following subsection and can be ignored here.

[4]Note that this of course only works for functional instruments. However, as stated in the beginning of this work, we are focusing on functional coverage of test programs for now. How to analyze coverage of non-functional behavior is a significantly more complex task and left for future work.

[5]CRAVE 2.0 offers coverage analysis as an additional feature for the current version of UVM-SystemC and, hence, provides a suitable candidate for this purpose.

utilized and the corresponding completeness of the verification process has been checked with CRAVE 2.0 as well. To this end, a total of 82 cover-points has been defined and analyzed. Each cover-point represents a specific use case which has to be covered. The verification environment then generated a total of 105 stimuli and coverage analysis confirmed that those cover all 82 cover-points.

In parallel to all these endeavors, the development of the test program started (c.f. Fig. 1). Once this has been completed, the coverage analysis method for test programs proposed above can be utilized. Applying the resulting test program (developed by an experienced designer) to the solution described in Section IV and sketched in Fig. 2 eventually generated a total of 90 stimuli[6]. However, the proposed coverage analysis eventually showed that this does not fully cover all the cover-points considered before. In fact, only 61 cover-points out of 82 cover-points (a coverage of 75%) has been achieved – the test program obviously is not complete.

This feedback now significantly helps the test program developer as it shows that further cover-points need to be triggered by the test program. By extending the test program and re-evaluating the coverage of it, eventually a test program with 100% coverage has been developed. The finally resulting test program eventually generates a total number of 117 stimuli and triggers all cover-points.

Overall, in order to achieve full coverage for the post-silicon test program the following *costs/benefits* arise:

- with the same efficiency as coverage of the functional verification process has been evaluated (this is because the same tool-chain is utilized for this purpose) and
- with no changes in the front-end, i.e. the test program developer neither has to adapt nor suddenly requires additional expertise about functional verification (everything needed from functional verification is re-used from the corresponding functional verification team and hided behind interfaces).

Overall, this allows for the evaluation of the functional completeness of a test program, which significantly reduces the risk of errors slipping through during the test process and, as a consequence, the risk of shipping a faulty product to the customer.

VI. Conclusion

In this paper, we proposed a test coverage methodology for the development of post-silicon validation test programs. To this end, we revisited the basic concept of test programs and discussed how coverage is handled at other abstraction layers, e.g. during functional verification. In contrast to the existing development for test programs which is entirely based on the experience of the test engineer, the proposed approach is capable to give the designer feedback in form of coverage information. We discussed the resulting application scenario and the benefits obtained by it. In fact, the proposed approach allows to check the coverage of test programs with the same efficiency as coverage of the functional verification process has been evaluated before and with basically no changes in the front-end.

[6]Note that, since those stimuli are now generated by the test program and not by the verification environment, i.e. they may be completely different to the 105 stimuli generated during functional verification.

Acknowledgements

The authors would like to thank Oliver Frank for making this contribution possible.

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] Schaller, Robert R., "Moore's Law: Past, Present, and Future," *IEEE Spectrum*, pp. 52–59, 1997.

[2] M. Dowson, "The Ariane 5 Software Failure," *SIGSOFT Software Engineering Notes*, pp. 84–84, 1997.

[3] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2007.

[4] M. Burns and G. W. Roberts, *An Introduction to Mixes-Signal IC Test and Measurement*. New York, USA: Oxford University Press, 2001.

[5] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. New York, USA: Kluwer Academic, 2002.

[6] S. Pointner, O. Frank, C. Hazott, and R. Wille, "Test Your Test Programs Pre-Silicon: A Virtual Test Methodology for Industrial Design Flows," in *IEEE Annual Symposium on VLSI*, Miami, USA, 2019.

[7] Teradyne. (2019) IG-XL Test Software. http://www.teradyne.com/products/semiconductor-test/ig-xl-software. accessed: 29.05.2019.

[8] K. Balston and M. Karimibiuki and A. J. Hu and A. Ivanov and S. J. E. Wilton, "Post-silicon code coverage for multiprocessor system-on-chip designs," *IEEE Transactions on Computers*, pp. 242–246, 2013.

[9] E. E. Mandouh and A. Gamal and A. Khaled and T. Ibrahim and A. G. Wassal and E. Hemayed, "Construction of coverage data for post-silicon validation using big data techniques," in *Int'l Conference on Electronics, Circuits and Systems*, Batumi, Georgia, 2017.

[10] T. Bojan and M. Aguilar Arreola and E. Shlomo and T. Shachar, "Functional coverage measurements and results in post-silicon validation of core 2 duo family," in *Int'l High Level Design Validation and Test Workshop*, Irvine, USA, 2007.

[11] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2005.

[12] E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal Verification: An Essential Toolkit for Modern VLSI Design*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2015.

[13] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken and R. Wille, "Completeness-Driven Development," in *Int'l Conference on Graph Transformations*, Bremen, Germany, 2012.

[14] H. Le and R. Drechsler, "CRAVE 2.0: The next generation constrained random stimuli generator for SystemC," in *Design and Verification Conference and Exhibition Europe*, Munich, Germany, 2014.

[15] R. Wille, D. Große, F. Haedicke, and Rolf Drechsler, "SMT-based Stimuli Generation in the SystemC Verification Library," in *Forum on Specification and Design Languages*, Sophia Antipolis, France, 2009.

[16] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. New York, USA: Springer, 2004.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, USA: MIT Press, 1999.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, pp. 385–394, 1976.

[19] P. Gonzalez-de Aledo, N. Przigoda, R. Wille, R. Drechsler, and P. Sanchez, "Towards a Verification Flow Across Abstraction Levels Verifying Implementations Against Their Formal Specification," *IEEE Trans. on CAD*, vol. 36, no. 3, 2017.

[20] R. Salemi, *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology*. Boston, USA: Boston Light Express, 2013.

[21] *Universal Verification Methodology 1.2 User's Guide*, Accellera, 2015.

[22] *UVM-SystemC Language Reference Manual*, Accellera, 2017.