Introducing New Verification Methods into a Company's Design Flow: An Industrial User's Point of View

Robert Lissel, Joachim Gerlach

Robert Bosch GmbH
Automotive Electronics
Tuebinger Strasse 123, 72762 Reutlingen, Germany
{robert.lissel , joachim.gerlach}@de.bosch.com

Abstract

Today the task of design verification has become one of the key bottlenecks in hardware and system design. To address this topic, several verification languages, methods and tools, which address several issues of the verification process, were developed by multiple EDA vendors over the last years. This paper takes an industrial user's point of view and explores the difficulties introducing new verification methods into a company's "naturally grown" and well established design flow – taking into account application domain specific requirements, constraints given by the existing design environment and economical aspects. The presented approach extends the capabilities of an existing verification strategy by powerful new features while keeping in mind integration, reuse and applicability aspects. Based on an industrial design example the effectiveness and potential of the developed approach is shown.

1 Introduction

In the area of hardware and system design, the task of design verification is taking over more and more a leading part. Today, it is expected that about 70% of the overall design effort is used for verification activities. Therefore, increasing the efficiency of the verification task will make a significant contribution to reduce time-to-market.

Against that background, a broad range of verification languages, methods and tools, which address several aspects of verification using different techniques, was developed by multiple EDA vendors over the last years. They include hardware verification languages like SystemC [1][2], SystemVerilog [3] or e [4] that are able to support verification issues much better than traditional hardware description languages like VHDL or Verilog. New verification strategies using object-oriented mechanisms as well as assertion-based techniques built on top of simulation-based and formal verification enable to implement a verification environment much more compact and reusable.

While advanced verification methods include high potential for increasing the level of verification efficiency and quality, their introduction into an existing and well established industrial development process often holds several difficulties. Taking an industrial user's point of view, several aspects have to be considered carefully. They include specific requirements that come up with the specific application area. Furthermore, the reusability of available verification components and commercial aspects like costs for new tool licenses and training of the design teams have to be considered. All those aspects potentially hinder that new and advanced verification approaches enter a "naturally grown" and well established company's design flow.

The paper at hand addresses the gap outlined above and critically discusses the difficulties introducing new verification methods into an existing company's design flow. In regard of the specific requirements of the automotive electronics design domain, the paper identifies verification tasks that include high potential and open challenges to be solved. For the example of a verification strategy built up at Bosch, the paper works out the specific requirements and environmental constraints that need to be considered. Finally the integration into our industrial design flow is shown taking into account practicability and applicability aspects.

The paper is organized as follows: Chapter 2 describes the landscape of verification methods and identifies challenges that are highly relevant. Chapter 3 works out our strategy for providing new and advanced verification mechanisms. Chapter 4 illustrates our approach in terms of an industrial design example, and discusses extended features resulting from our strategy.

2 Verification challenges

Over the last years, a large number of verification tools and methods were developed, which address several aspects

978-3-9810801-2-4/DATE07 © 2007 EDAA

of verification using different techniques. In the area of digital hardware verification, metrics for an assessment of the verification status as well as simulation-based and formal verification approaches are taking central positions. Figure 1 gives an overview of the approaches and the derived methods. Different design and verification languages and EDA solutions of different vendors are covering this "verification landscape" to different degrees and in different parts.

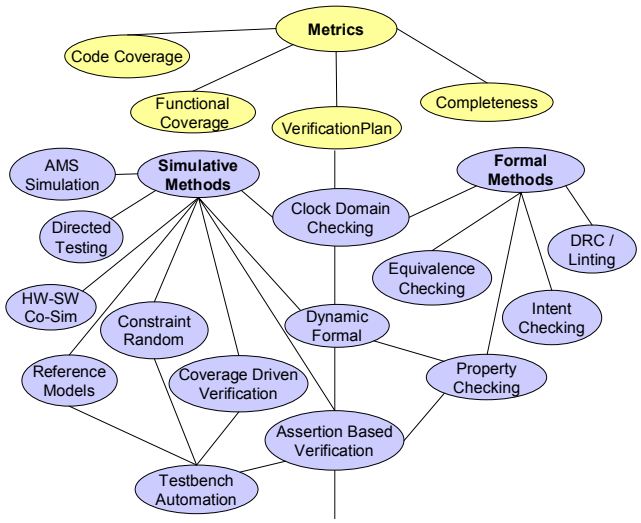

Figure 1 Verification landscape

Introducing new verification languages and methods into a well established design and verification flow requires not only pure technical discussions. Also the acceptance among the developers as well as the risk of touching a well working process has to be considered. Therefore, a smooth transition and the ability to reuse legacy verification code are essential. Especially, existing testcases contain a lot of information on former design issues. Since the majority of automotive designs are classified as safety critical, even a marginal probability of missing a bug due to a new verification method is not acceptable. On the other hand the reuse of legacy code should not result in multiple testbench approaches within one project. So the use of legacy testcases should ideally be part of a new approach. Thus it would be possible to enhance existing testcases instead of writing new ones. Besides, reuse often requires slight changes in existing testcases due to new features that have been implemented. Hence, if a new verification method does not support an adaptation of legacy testcases, they have to be re-implemented.

The second important challenge is to convince the designer to apply new methods and languages. Designers are experienced and work efficiently with their established strategy. Loosing this efficiency is a serious risk. In many cases there is no strict separation between design and verification engineers. Hence, a large number of developers are affected when changing the verification method. Furthermore, new methods require training activities and cause a considerable overhead during their first applications. But most of the projects have a tough time schedule and do not allow trying out and possibly rejecting a new method. To overcome those difficulties, it is important to carefully collect all the requirements and to evaluate new approaches outside critical projects. A possible way is to introduce new methods as add-on to the existing approach. Thus a new method or tool may improve the quality but would never make it worse. As a consequence, the evolution of verification methods might be more desirable than a completely new solution.

Considering the technical aspects of verification, automotive designs show some interesting special requirements: The variety of digital designs reaches from a few thousand gates to multi-million gates System-on-Chip designs. Typical automotive ICs implement analog, digital and power semiconductors on the same chip. The main focus for those mixed signal designs is the overall verification of analog and digital behavior rather than a completely separated digital verification. But also pure digital IC's e.g. in the area of car multimedia have to be covered.

In practical use, the functional characteristics of the design to be checked determine the most appropriate verification method: If the calculation of the expected behavior is "expensive", directed tests may be the best solution. If there is an executable reference model available or if the expected test responses are easy to calculate, a random simulation may be the first choice. Instead of defining hundreds of directed testcases, a better approach could be to randomize the input parameters with a set of constraints allowing only legal behavior to be generated. In addition, special directed testcases could be implemented by appropriately constraining the randomization. The design behavior is observed by a set of checkers. Functional coverage points are necessary to achieve a visibility of what functionality has been checked. Observing functional coverage and manually adapting the constraints to meet all coverage goals leads to *Coverage-Driven Verification* (CDV) techniques. Automated approaches built on top of different verification languages [1][2][3][4][5] result in *Testbench Automation* (TBA) strategies.

A directed testbench approach might be most suitable for low complexity digital designs, in case that reference data is not available for randomizing all parameters or the given schedule does not allow implementing a complex constraint random testbench. Furthermore, mixed-signal designs may require directed stimulation. Often the function is distributed over both analog and digital parts, e.g. an analog feedback loop to the digital part. Verifying the digital part separately makes no sense in this case. In fact, the interaction between analog and digital parts is error-prone. Thus, the integration of analog behavioral models is necessary in order to verify the whole function. One technique to deal with this requirement is mapping the analog function to a VHDL behavioral description and simulating the whole design in a directed fashion. In other cases, the customer

delivers reference data originating from a system simulation (e.g. done in Matlab [6]). Integrating that reference data within a directed testcase is mandatory. Since each directed testcase may be assigned to a set of features within the verification plan, the verification progress is visible without implementing functional coverage points. Hence, the implementation effort is much lower compared to a constraint random and CDV approach up to a certain design complexity. Anyway, for some parameters not affecting the expected behavior (e.g. protocol latencies) it makes sense to introduce randomization.

Formal verification techniques like property checking provide the opportunity to prove the correctness of a design characteristic in a mathematically correct manner. In contrast to simulation-based techniques, which only consider specific paths of execution, formal techniques allow for an exhaustive exploration of the state space. On the other hand, formal techniques are usually strongly limited in circuit size and temporal depth. Therefore, formal and simulation-based techniques need to be combined carefully to optimize the overall verification result while minimizing the verification effort to be spent. This contribution concentrates on the simulation part of the problem.

In summary, the challenge is to apply the different verification techniques where they fit best. In order to achieve visibility of the verification progress and the contribution each technique provides, we need powerful metrics. The question is how to achieve the best result regarding the available time, money and manpower budget rather than finding the theoretically best solution. The requirements on verification methods reach from mixed-signal simulation, simple directed testing to complex constraint random and formal verification as well as hardware/software integration tests. Nevertheless, a uniform verification method is desired, providing the flexibility to satisfy all the needs of verification within an automotive environment.

3 Verification strategies

For an illustration of the aspects discussed before, this chapter demonstrates how SystemC has been applied to enhance a company-internal VHDL-based directed testbench approach while meeting the challenges defined within the previous chapter. This SystemC-based approach allows the introduction of constraint random verification techniques but also the reuse of existing testbench modules and testcases. Thus a smooth transition towards a new method is possible.

3.1 VHDL-based testbench approach

As figure 2 shows, the main concept of our testbench approach is to associate one testbench module (TM) or bus functional model with each design-under-test (DUT) interface. All those testbench modules are controlled by a single command file. Each testbench module provides interface-specific commands to its DUT interface. Furthermore, it implements a command loop process requesting the next command from the command file using a global testbench package. Thus a so called virtual interconnect layer is established. Structural interconnect is required only between testbench modules and DUT.

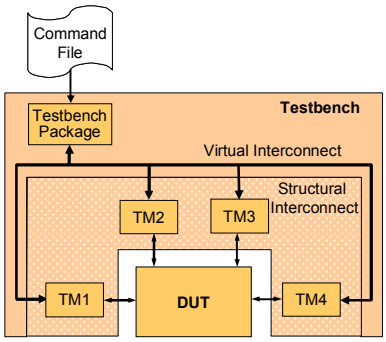

Figure 2 VHDL testbench approach

The command file is an ASCII file containing command lines for the particular testbench modules as well as control flow and synchronization statements. Due to its unified structure, this testbench approach allows to easily reuse existing testbench modules.

```
1 : CLK PERIOD 10 ns
2 : CLK RESET  0  12
3 : ALL SYNC ALL
4 : -- configure filter 1
5 : CFG WRITE h#F004          b#10011000
6 : CFG READ  dec1_cfg_addr1 h#98
7 : ALL SYNC A2M CFG
8 : -- configure audio channel 1
9 : A2M FREQ_CHANNEL  1 48 KHz
10: A2M START_CHANNEL 1
11: -- run filter for 1 ms
12: ALL WAIT deltime 1 ms
13: ALL QUIT
```

Figure 3 Command file example

Figure 3 gives an example of the command file syntax. Each line starts with either a testbench module identifier (e.g. CLK, CFG), the ALL identifier for addressing global commands (e.g. SYNC) or control flow statements. Command lines addressing testbench modules are followed by a module-specific command and optional parameters. Thus line 1 addresses the clock generation module CLK. The command PERIOD is implemented within this clock generation module for setting the clock period and requires two parameters: value and time unit. Line 3 contains a synchronization command to the testbench package. The parameter list of the synchronization command specifies the modules to be synchronized (all for line 3; A2M and CFG for line 7). Since in general all testbench modules are operating in parallel and thus requesting and executing commands independently, it is important to synchronize them at dedicated points within the command file. When receiving a synchronization command, the specified testbench modules will

stop requesting new commands until all of them have reached this synchronization point.

3.2 Introducing a SystemC-based approach

Motivation for applying SystemC is to enhance the existing VHDL-based testbench approach. The idea behind the initial VHDL testbench approach was to define a sequence of commands, which are executed by several testbench modules and thus to describe a testcase within a simple text file. This works fine with the implemented VHDL-based approach. But the application of this concept showed that it is also desirable to get more flexibility within the command file. Besides, VHDL itself lacks advanced verification features provided by Hardware Verification Languages (HVL) like e, SystemVerilog HVL and SystemC together with the *SystemC Verification Library* (SCV).

However, the basic concepts of applying a simple text file for defining testcases as well as the parallel command execution and synchronization of testbench modules have proved to be efficient. Therefore we decided to extend the existing approach. But a hardware description language like VHDL is not really suitable to implement a testbench controller which has to parse and execute an external command file. Hence, we decided to apply SystemC which provides a maximum flexibility, due to its C++ nature and the large variety of available libraries, especially the SCV. Using SystemC requires a mixed-language simulation approach. The DUT may still be implemented in VHDL, whereas the testbench moves towards SystemC. Available commercial simulators support mixed-language simulation.

The implemented SystemC testbench controller covers the full functionality of the VHDL testbench package and additionally supports several extensions of the command file syntax. This makes the usage of existing command files fully compliant to the new approach. The new SystemC controller enables to apply variables, arithmetic expressions, nested loops, control statements and especially random expressions to be defined within the command file. However, those features are intended to implement testcases more efficiently and flexibly. In general, the major testbench behavior should be implemented in VHDL or SystemC within the testbench modules. Thus the strategy is to implement more complex module commands rather than too complicated command files. But the SystemC approach does not only extend command syntax. It provides static script checks, more meaningful error messages and debugging features.

Implementing the testbench controller in C++ following an object-oriented structure allows improving the usability of the concept. A SystemC testbench module is inherited from a testbench module base class. Hence, only the module-specific features have to be implemented. For example, the VHDL-based approach required to implement a command loop process for each testbench module in order to fetch the next command. This is not required when applying the SystemC approach because the command thread is inherited from the base class. Only the command functions have to be implemented. For the implementation of new features like expression evaluation, the use of C++, with lots of libraries available, shows its strength in particular. Thus, we take advantage of using the Spirit Library [7] for resolving arithmetic expressions within the command file.

Another important requirement from a practical point of view is that existing VHDL-based testbench modules may be used unchanged within the new approach. Therefore, SystemC co-simulation wrappers need to be implemented. Generation of co-simulation wrappers is provided using a fully automated transformation approach which is described in [8]. Hence, a legacy VHDL-based testbench environment may be transferred to our new SystemC-based approach. All VHDL testbench modules are wrapped in SystemC and a new SystemC testbench top-level is built automatically. This allows taking advantage of the new command file syntax without re-implementing any testbench module in SystemC. Especially, the introduction of randomization within the command file gives the chance to enhance existing testcases with minimum effort.

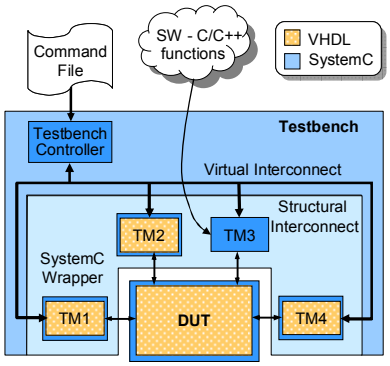

Figure 4 SystemC testbench approach

Figure 4 visualizes a testbench environment including a mixture of VHDL and SystemC testbench modules. As a first step, legacy testbench modules may be kept, like it's shown for TM1, TM2 and TM4. Some of the testbench modules, like TM3, may be replaced by more powerful SystemC modules later. Besides, SystemC modules allow the easy integration of C/C++ functions. Moreover, the testbench module provides the interface handling and correct timing for connecting a piece of software.

4 Design example

In the following, some extended and new verification features resulting out of our SystemC-based testbench approach will be shown and discussed in terms of an industrial design example, a configurable decimation filter, taken from a Bosch car infotainment application. The decimation filter is used for reducing the sampling frequency of audio data streams. Actually, the decimation filter consists of two independent filter cores. The first one may reduce the input

sample frequency of one stereo channel by a factor of three, while the second one may either work on two stereo channels with a decimation factor of two or on one stereo channel with a decimation factor of four. The decimation filter module possesses two interfaces with handshake protocols. One is applied for audio data transmission and the other one for accessing the configuration registers.

The original verification environment has been implemented in VHDL, based on the legacy testbench concept as described within chapter 3. Beside a clock generation module, two testbench modules for accessing both the data transmission and the configuration interface were required. For fulfilling the verification plan, a set of directed testcases (command files) has been created.

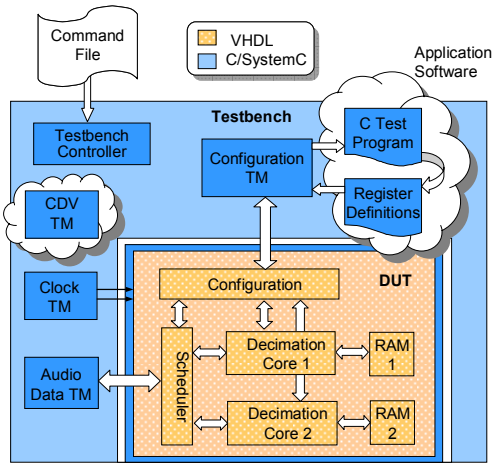

Figure 5 Decimation filter

Figure 5 shows the top-level architecture of the decimation filter embedded within a SystemC-based testbench. The example demonstrates the smooth transition towards our SystemC-based testbench approach as well as the application of constraint random and coverage-driven verification techniques. Furthermore, the presented approach proves its flexibility by providing an efficient hardware software co-verification method.

4.1 Constraint Random Verification

For the decimation filter example, the randomization mechanisms of our SystemC-based testbench approach have been extensively used. Thus, randomized regression tests have discovered some interesting corner cases. As a first step, the existing VHDL testbench modules have been implemented in SystemC. Thereby SystemC showed no significant difficulties nor did require more implementation time. In order to check the compliance with the legacy VHDL approach, all existing testcases have been re-simulated.

Since reference audio data is available for all filter configurations, a random simulation could be implemented quickly. Thereby randomization techniques have been applied to both the testbench modules and the command file. The command file has been split into a main file containing the general function and an include file holding randomized variable assignments. The main command file consists of a loop which applies all randomized variables from the include file in order to reconfigure and run the filter for a dedicated time.

```
10: -- number of reconfiguration iterations
11: A2M ASSIGN reconf_cnt = 20
12: -- randomized parameter (for each iteration)
13: -- define filter operation
15: A2M ASSIGN rand_filter_op_dec2 = ${ keep 20% , stop 20% ,
                                        fact_2 , fact_4 }
19: -- frequency in KHz
20: A2M ASSIGN rand_in_freq_dec1 = ${ 144 60% , 72 30% , 36}
22: -- run time for each iteration
23: A2M ASSIGN rand_run_time      = ${ 50000:60000 }
24: A2M ASSIGN rand_load          = ${ none , rand ,
                                       low_const, low_rand ,
                                       mid_const, mid_rand ,
                                       high_const, high_rand}
```

Figure 6 Constraint include file

Figure 6 illustrates an excerpt from the include file. For example line 24 describes the load scenario at the audio data interface. The variable #rand_load will be applied as parameter to a command of module A2M later within the main command file. A directed test will be enforced by assigning constant values instead of randomized items. Hence, the required tests, claimed by the verification plan, could be implemented more efficiently as constraint include files. After the verification plan has been fulfilled all parameters may be randomized for running overnight regressions and finding corner cases.

4.2 Coverage-Driven Verification

To get an idea of the verification progress, especially when applying random verification, coverage metrics are required. Analyzing the code coverage is necessary but not sufficient. For the given example, we implemented a set of functional coverage points using PSL [5]. Since PSL does not support cover groups and cross coverage, we developed a Perl [9] script generating those cross coverage points. Nevertheless, implementing coverage points has been a considerable effort, but as a result we recognized some verification holes within our VHDL directed testbench. Considering the fully randomized testcase, all coverage points will be eventually covered. In order to meet the coverage goals faster and thus reducing the required simulation time, a much more efficient approach is defining multiple randomized testcases using stronger constraints.

Automating this procedure of manually adapting constraints leads to another understanding of TBA which is the automatic adaptation of constraints due to the measured coverage results. Therefore it is necessary to manually define dependencies between constraints and coverage items. Such a testbench would hit all desired coverage points automatically. The disadvantage of this approach is the high implementation effort for the definition of constraints, cov-

erage items and their dependencies. Nevertheless, we discovered a method based on our SystemC testbench and PSL: First we need access to our coverage points. Therefore we assign coverage points to VHDL signals which may be observed from SystemC. Thereafter we define dependencies between those coverage results and constraints within either the command file or a SystemC testbench module. For automating this method we improved the above mentioned Perl script. Thus, we generate a CDV testbench module, which either passes coverage information to the command file or may be extended for adopting the constraints in SystemC.

4.3 HW/SW co-simulation

In the target application, the decimation filter is embedded within an SoC and thus controlled by a processor. In order to setup a system level simulation, a vendor-specific processor model is given in C and Verilog. Hence, the compiled and assembled target application software, implemented in C, may be executed as binary code on the given processor model. But due to this co-simulation approach, the simulation performance decreases notably, although the actual behavior of the processor model is not relevant in this case.

The application C code consists of a main function and several interrupt service routines. Controlling the audio processing module like the decimation filter is done by accessing memory-mapped registers. Thus the processor performs read and write accesses via its hardware interface.

To overcome the performance limitation, the idea is to omit the processor model and connect the C code directly to a testbench module, like illustrated by figure 5. Due to its C++ nature, our SystemC-based testbench approach offers a smart solution. The intention is to map our testbench modules read and write functions to register accesses within the application C code. Therefore, we re-implemented the existing register definitions, applying an object-oriented style. This allows overloading the assignment and implicit cast operators for those registers. Hence, reading a register and thus applying the implicit cast results in a read command being executed by the testbench module. Similarly, assigning a value to a register results in a write command being executed by the testbench module. Finally, we need a mechanism to initiate the execution of the main and interrupt functions from the application C-code. Therefore, we implemented module commands starting those C-functions. Hence, we were able to control and synchronize the execution of those functions within our command file. This is essential in order to control the audio testbench module, which is required to transmit and receive audio data with respect to the current configuration. In order to execute the interrupt functions we applied the interrupt mechanism of our testbench concept.

5 Conclusions

In this paper, difficulties introducing new verification methods into an existing company's design flow were worked out and critically discussed.

Taking a company-internal VHDL-based testbench approach as an example, we demonstrated a smooth transition towards advanced verification techniques based on SystemC. The presented approach allows us to reuse existing verification components and testcases. Therefore, we guarantee that running projects benefit from new techniques without the risk of losing design efficiency or quality. This results in a maximum of acceptance among the developers, which is essential for successfully introducing new methods. The effectiveness and potential of the developed approach was shown in terms of an industrial design example given by a configurable decimation filter.

6 Acknowledgements

This work was partially funded by the German BMBF (Bundesministerium für Bildung und Forschung) under grant 01M3078.

7 References

[1] Open SystemC Initiative (OSCI), *SystemC 2.1 Library*, www.systemc.org
[2] Open SystemC Initiative (OSCI), *SystemC Verification Library 1.0*, www.systemc.org
[3] IEEE Std 1800-2005, *IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language*
[4] IEEE Std 1647-2006, *IEEE Standard for the Functional Verification Language 'e'*
[5] IEEE Std 1850-2005, *IEEE Standard for Property Specification Language (PSL)*
[6] The MathWorks Homepage, www.mathworks.com
[7] Spirit Library, http://spirit.sourceforge.net
[8] J.H. Oetjens, J. Gerlach, W. Rosenstiel, *An XML Based Approach for Flexible Representation and Transformation of System Descriptions*, Forum on Specification & Design Languages (FDL) 2004, Lille, France.
[9] Wall, Larry, et.al., *Programming Perl (Second Edition)*, O'Reilly & Associates, Sebastopol, CA., 1996.
[10] IEEE Std 1076.3-1997, IEEE standard VHDL synthesis packages