On the Verification of Automotive Protocols

G. Zarri, F. Colucci, F. Dupuis, R. Mariani, M. Pasquariello, G. Risaliti, C. Tibaldi*

YOGITECH SpA, San Martino Ulmiano (Pisa), Italia - www.yogitech.com

*Politecnico di Torino - Dipartimento di Informatica e Automatica- Torino, Italia

Abstract

Verification quality is a must for functional safety in electronic systems. In automotive, the verification flow is historically based on a layered approach, where each level (model, design and system) has its proper verification and validation methodology. Very often, these methodologies are badly or not interconnected at all one to another, and it's still common to see some of the most critical verification tasks confined to post-silicon validation, where costs to solve issues could be a killing factor for deeply integrated electronic systems.

This paper presents the architecture of verification components that can be applied in all the different levels and shows how they have been successfully applied to the verification of systems integrating LIN, CAN and FlexRay protocols.

1. Introduction

Electric and electronic systems account for 49.2% of automobile breakdowns, according to Germany's Allgemeiner Deutscher Automobil Club in data for 2003. Recent statistics on ICs say that 71% of System-on-Chip (SoC) re-spins are due to logic bugs, and 47% of these are because of incorrect or incomplete specifications. Moreover, 14% of failing SoCs show bugs in reused components or IPs. Such statistics show why 60-70% of the entire product cycle for a complex logic chip is dedicated to verification tasks [1]. The same message is stated in the IEC-61508 standard for functional safety of electronic systems, which classifies as mandatory functional testing and fault injection [2].

The verification flow of electronic systems for automotive is historically based on a layered approach, where each level (model/module, design/chip and system/application) has its proper analysis or verification or validation methodology. Very often, these methodologies are badly or not interconnected at all one to another. It's still common to see some of the most critical verification tasks (such as protocol compliance or fault injection) confined to post-silicon validation, where costs to solve issues could be a killing factor for deeply integrated electronic systems.

To reach the best verification quality, a more integrated approach is suggested, where verification components and related methodologies applied at module level can be re-used in other levels to better cover particular scenarios or hard-to reach "corner" cases in the early development stage. Modern verification approaches such as constraint-driven random test generation and functional coverage analysis, can be applied at different level of abstractions guaranteeing a better match of verification plans.

This paper shows how this integrated approach can be realized and applied to state-of-art automotive protocols such LIN, CAN and FlexRay, showing examples of the use of verification components at different abstraction levels based on the characteristics of each of these protocols.

2. LIN protocol verification

Local Interconnect Network protocol (LIN, [3]) is one of the most used serial protocol in automotive: it is a single wire serial protocol, single master multi slave with self synchronization mechanism. The specification of a LIN network covers different layers (figure 1), and therefore the architecture of a verification component for LIN should be able to be reused in all of them. Each of these layers is specified by different files such for instance LIN description File (LDF) and Node Capability File (NCF).

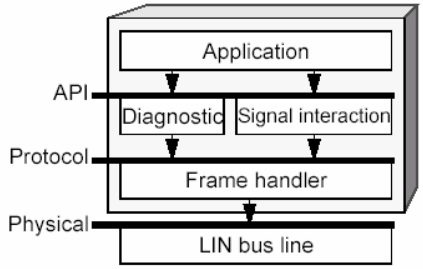

Figure 1: the layered structure of LIN

2.1. LIN verification component

Figure 2 shows the proposed architecture of a verification component to be reused in all the different levels. Generally speaking, a verification component is an IP that performs the verification tasks at design stage. For the most popular verification methodology called "coverage-driven dynamic verification", a verification

3-9810801-0-6/DATE06 © 2006 EDAA

component is composed by four main parts: agents, monitors (including checkers), coverage and configuration. The LIN verification component proposed in this paper is composed by a master *agent* that can be configured to be active or passive. When is active, based on existing frame, the agent can send the network configuration (assign PID, etc..) and/or send randomly requests for all the available frames. This is done in the sequence driver with a dedicated configuration sequence. The sequence driver is responsible of the master task and the slave task of the master: it selects a header, and if publisher of the frame, it sends the response as well. When passive, the Bus Functional Module (i.e. the module that is in charge of interacting with the transmission lines) is not present, but the monitor is able to analyse frames to check if the configuration is well analysed by all the slaves.

The verification component includes a slave agent that can be configured as an active or passive agent. When passive, it cannot transmit on the LIN network. But it will check the behaviour of the slaves after their configuration (check the response to the related PID, hook for the scoreboards in order to check that reception of a subscribed frame well updated the corresponding signals). When is active, it analyses the configuration, and reacts when it's subscriber or publisher of a frame.

The verification component is completed by monitors. The monitors include the protocol checker and some additional checks for higher level: time between two wake-up request, response space in the range defined by the node capability file, etc….The link between the BFM (Bus Functional Module) and the monitors is needed to check if the value sent on the LIN network lines is the correct one: if not, the transmission must be stopped.

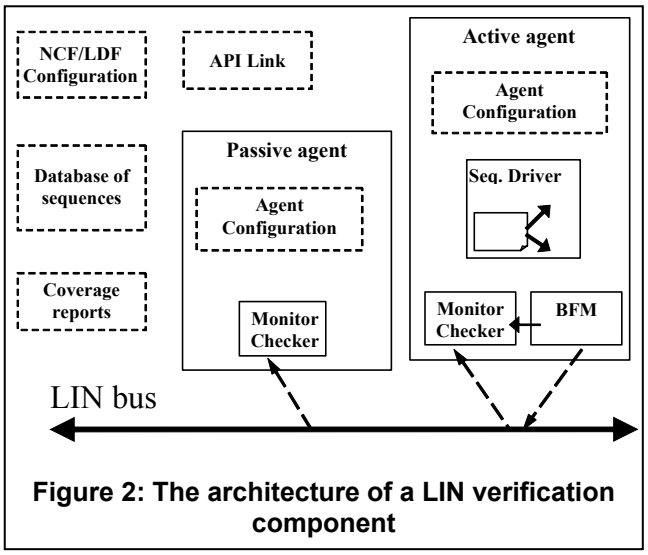

Figure 2: The architecture of a LIN verification component

The verification component should be easily configurable: all pieces of information relative to a node (message id, NAD, published/subscribed frames and so on...) must be randomly generated, to allow the checking of all possible configuration of a node. To be linked with upper layers, the LIN verification component architecture should offer the possibility to extract from the NCF (Node Capability File) all the parameter of a node. This allows simulating a complete network with only the description available and not the real IP, by instantiating a verification component node with the same characteristic. Moreover, the LIN verification component must includes a co-verification link in order that a real application program can be run for each slave and/or for the master. In such a way, a software engineer could start writing application code even if the design is not yet available, and debug it.

A database of sequence must be also available. This database should include standard sequences of test but also some fault sequences to easily check the most common faults (error in parity, in protocol etc…).

The proposed verification component has been realized and successfully used in the verification of LIN designs [4], as described afterwards. It is worth to note that such component, as the others described in this paper, has been implemented using the "e" language, under standardization with the IEEE initiative P1647 [5].

2.2. Coverage

Coverage is one of the most important aspects of a verification environment since it is the metric to assess the quality of the verification task [6].

Particular events in the simulation induce the embedded coverage unit of the verification component to collect information on the conceptual items that are defined as "coverage groups". They represent the functional coverage items that can be used for test program ranking. This is a critical measure to understand if the validation task can be considered fully completed or vice versa the verification engineer still needs to develop new tests to cover the "holes" in his environment. In fact only when LIN basic coverage items, user defined coverage definitions and the HDL code coverage percentages reach 100%, the verification engineer can consider himself safe about having exhaustively stressed the module under test.

For example for a specific set of LIN tests, it is important to know if all the possible response lengths (that can vary from 1 to 8 bytes) have been submitted/received to/from the IP, if the sequence has been interrupted at any time during the transmission (i.e. during sync, PID and response phases), if the node has been at least once publisher and/or subscriber of a frame, if different inter-bytes spaces and/or response spaces have been tested, if break fields with different length have been submitted to the IP in order to cover specific predefined ranges of values, and if the sent data have been properly read back on the reception line.

Moreover, coverage items will give feedback on specific corner cases: if the frames had good or bad checksum and parity, they were compliant to 2.0 or 1.3 specifications, the PID was valid or invalid, etc...

2.3. Module-level verification

The following is an example of module-level verification using the LIN verification component described in previous paragraph, taken from a real project. The goal was to verify a LIN Controller IP having an AMBA APB interface on one side, and a LIN interface on the other side. This IP being a slave, it has been used the master agent of the LIN verification

component in order to send LIN frames over the network.

As shown in figure 3, the LIN verification component is connected to the LIN side of the IP and it is used to send frame headers and responses on LIN RX input, collect responses on LIN TX output, monitor the traffic, and check the LIN protocol and collect coverage information. An AHB Verification component [7] has been used to emulate the "system" controlling the IP itself.

Coverage information are improved for the specific environment by adding items like the interrupts set in an interrupt register each time the AHB verification component will read such an address on the interface in order to confirm the all the possible interrupts have been checked during the test. Finally the scoreboard collects data information from both APB and LIN bus and compare them together in order to check if all the TX/RX frames are correctly passed through the LIN IP.

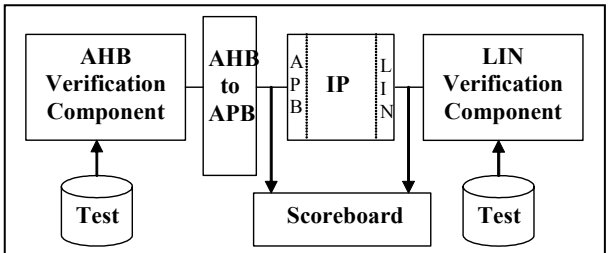

Figure 3: LIN module-level verification

The following is an example of bugs that can be easily detected with this environment. According to the LIN specifications, the end of the synchronization falls together with the 4th falling edge of the synch-break field, each transmission should start with a start bit (i.e. the line is set to '0'), and stop with a stop bit (i.e. the line is set to '1') and errors that might occur will cause the nodes to ignore the current transmission.

In the verified LIN IP, the checks on the start bit and stop bit were properly done by the decoder, which however was erroneously disabled during the synchronization phase. Since the last 2 bits of the synchronization field are not useful for the synchronization, no specific checks were done. Therefore, if a collision occurs on the physical interface such that the stop bit of the synchronization field is corrupted, the LIN IP will not be able to see the error condition. Since there will be no transmission issues detected, the LIN IP will wait for the next byte synchronizing itself on the next start bit of the Identifier field byte (PID), instead of ignoring the following transmission until the beginning of the next frame.

Using the proposed architecture of the LIN verification component this bug was easily found by sending a random sequence having wrong stop bits in the header.

Such a scenario is extremely simple to produce, and the same thing would have been more difficult to be verified with a standard testbench environment, due to difficulty with which corner-cases can be implemented.

Another typical example of scenario that takes advantage of the proposed verification architecture, it is to interrupt the transmission in a specific part of the frame by sending a new frame in order to test the break detection even during a reception. In fact, one of the crucial aspects of the environment is the capability to randomly generate the constraints: this allows the user to have both the simplicity of a test that randomly generates the desired scenarios, and the easy control of all the generated parameters in order to generate very specific tests.

The same idea can be applied in order to send some frames with bad checksum, bad parity, bad start/stop bits, or to limit the break field size to a specific set of values (for example always greater then 15 Tbits).

2.4. System-level verification

The following figure 4 shows how the architecture used at module level is fully reusable at system level. If many verification components are available, each for any interface of the LIN, they can generate independent transaction objects or structures to work on different sides of the LIN network. In such a system, the check of the AHB/APB interface is not needed anymore. The AHB verification component can be easily replaced by a piece of software that will handle the good LIN IP transmission/reception of the frames; different slaves verification components (active or passive) will be connected to the LIN network in order to reach the desired level of system complexity.

Possibility to easily move from module to system level is make easier by the use of the *e*RM methodology and the object oriented philosophy [8]. *e*RM methodology guarantees that all the verification component that are instantiated in the verification environment can work in the same way (when different types of protocols need to be checked) and that similar structures can be generated and merged in sequences very easily (leveraging on compatibility and absence of conflicts in resource sharing).

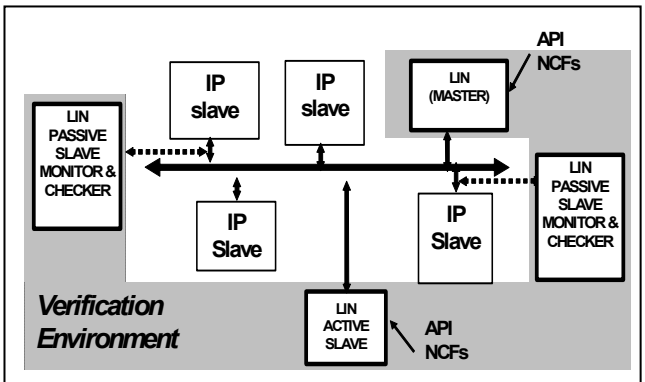

Figure 4: LIN system-level verification

3. CAN protocol verification

The same concepts used for LIN verification can be used with CAN, a well-known protocol [9] performing serial broadcasting communication with multi-master bus access, time synchronization, high error detection capability and a layered structure similar to the LIN case.

Respect other commonly used test benches such [10], the distinguish elements to be addressed by a CAN verification component are protocol compliance (e.g. the respect of ISO 11898-1, ISO 16845 [11]), random noise injection and the extension of the verification to chip and system level.

A verification environment combining all the previous mentioned aspects has been implemented and successfully used to prove CAN-based designs [12-14]. The model-level verification of a CAN node is done (see figure 5) by using a CPU model or a BFM (Bus Functional Model) to create meaningful scenarios for the CAN Protocol Engine (PE).

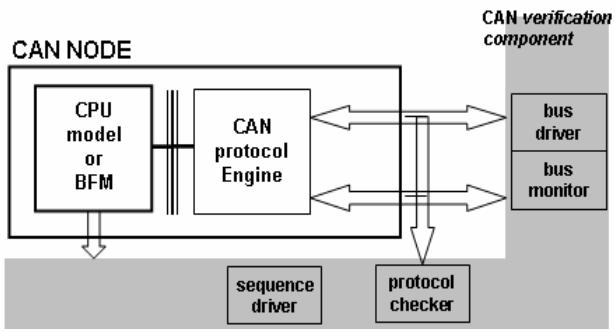

Figure 5: CAN model-level verification

The CAN verification component is used to drive the CAN bus and to monitor CAN PE's answers. The monitor logs all traffic information and collects items for test functional coverage. The protocol checker is a runtime tool checking CAN rules of the current bus traffic. If some problem conditions are detected during the simulation phase, the checker prompts the user about the error printing a message on the violation.

These rules can be extended and customized by the user. A sequence driver coordinates all CAN verification component functions based on a test suite following, the ISO 16845 and ISO11898-1 norms.

At chip level (figure 6), a full CAN node, including a CPU, the CAN PE, and the CAN RX/TX transceiver, is verified re-using the CAN verification component, and adding a top-level verification environment to control the CPU and monitor the CPU bus.

A model of the channel is provided and fault injection is done to verify the robustness of CAN protocol: the CAN verification component includes a *random noise injector* that is particular important to verify CAN synchronization. In fact, most of electronic systems require at least a noise analysis (i.e. the injection of a disturbance in the control lines and the measurement of the consequent degradation of the system mission).

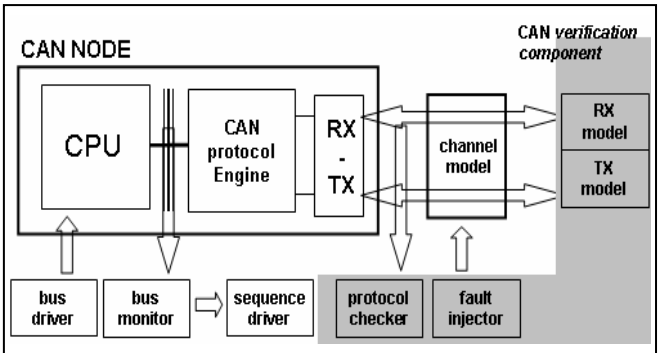

Figure 6: CAN chip-level verification

A typical example of this is the verification of robustness of CAN protocol units against line noise (the injection of bit errors in the packets). In this case CAN devices should be able to deal with these kinds of errors and re-transmit until they are successful. At the same time they should not clog the bus with re-transmissions due to faulty or marginal designs and these must be identified before silicon goes into production.

Therefore, an important function of a chip and system-level verification environment is the ability to inject permanent or intermittent errors, with configurable fault models. In addition, all the failures foreseen by the International Transceiver Conformal Test should be applied, such shorts between CANH and CANL, and shorts with VBAT (battery voltage) or ground.

At system level (figure 7), many CAN nodes can be verified in a real system, also taking into account the API (Application Programmer's Interface) to provide a co-simulation link between the sequence driver and the application software. Functional patterns of a set of stimuli and expected values can be generated out of this environment to link system-level verification with post-silicon test.

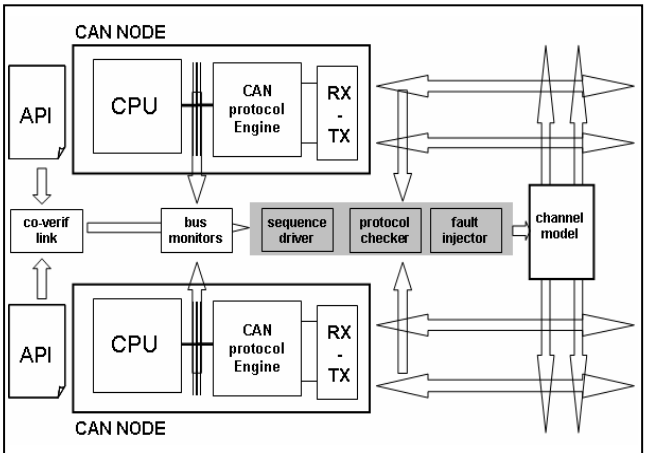

Figure 7: CAN system-level verification

4. FlexRay protocol verification

FlexRay is an emerging protocol for high-reliability communications with high data rate [15]. Also in this case, the crucial aspect of verification is to have a

modular verification component that can be used at the different levels of abstraction.

Figure 8 shows the schematic architecture of the proposed verification component for the FlexRay node. The main block is the PE verification component that contains a model of the Protocol Engine based on what described in FlexRay specifications. Moreover, a PE-CHI (Control Host Interface) verification component must be implemented: it contains BFM, monitor and protocol checker to easily handle the interface between PE and CHI. With the included blocks is possible to link the PE verification component with the CHI part of the node that can be described in HDL or in *e*/SC. An API should be also included for this purpose, to have the possibility to co-verify software tests. The independent PE-CHI monitor can be reused when the PE-CHI active part (BFM) isn't instantiated in the environment.

The other block of the verification component is the Channel, that contains a model of the FlexRay channel, able to add delays between each node participating in the cluster. An indipendent TX/RX Monitor is also included to monitor the two TX-RX lines of the FlexRay interface of the PE. The monitor is able to report the frames seen and to check the basic rules on the structure of the frame (CRC, lenght of field, etc..). The verification component architecture is completed by a Config structure that contains all the FlexRay global and Node specific configurations parameters. All the parameters are chosen following and satisfying the rules specified in the FlexRay Specification.

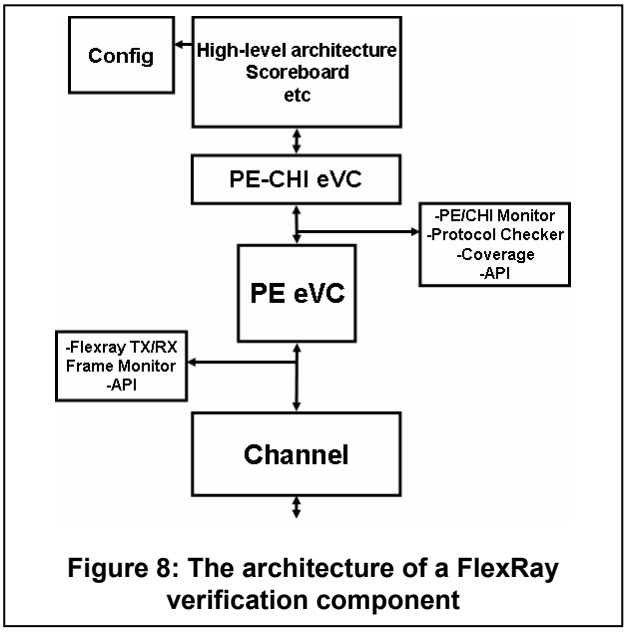

Figure 8: The architecture of a FlexRay verification component

The verification component proposed in this paper can be used at node level both to verify the Protocol Engine (figure 9a) and the upper levels of the Flexray node (figure 9b) such the CHI.

At system level (figure 10), the FlexRay verification component can be used both to emulate a FlexRay network and therefore to verify a complete RTL implementation of a FlexRay node. It is worth to note that to start a FlexRay communication two nodes are enough, except during startup where for fault tolerance requirements at least three nodes need to be there. Then the user can constraint its global environment and drives the full verification scenario through the host driver and adapting the scoreboards to extract useful coverage information.

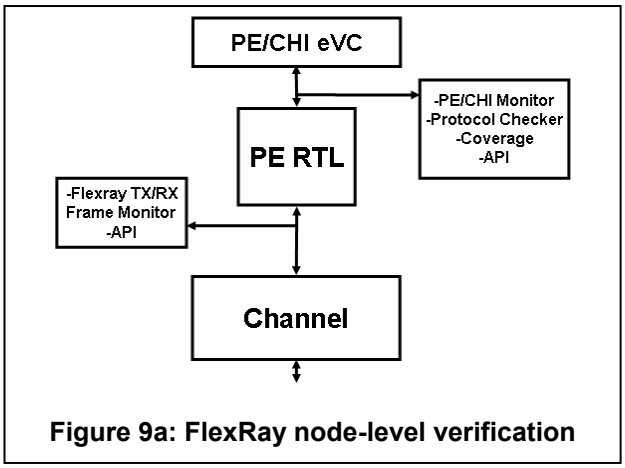

Figure 9a: FlexRay node-level verification

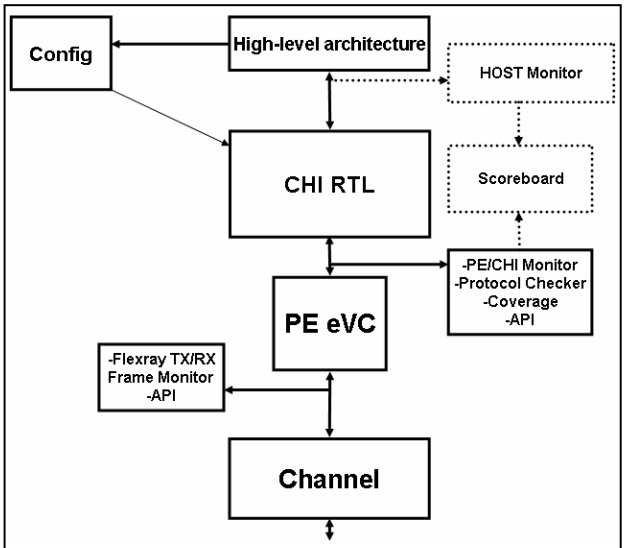

Figure 9b: FlexRay node-level verification

5. Comparison with other methodologies

The verification components proposed in this paper introduce significant benefits respect traditional verification methodologies (such as RTL testbench or pure software driven tests), summarized in the following:

- Use of random-constrained stimuli generation to reach particular corner-cases, really important for protocols
- Coverage-driven verification to precisely specify the verification goals and reach the highest verification quality
- Embedded capability of run-time protocol checking, fully aligned with protocol specs
- High-level of reusability in all the different level of abstractions and possibility to easy customize the verification environment to the specific application

need. This is particular important for verification blocks such monitors and checkers.

- Inclusive of random fault-injection to model bus noise
- To allow a co-verification link to be used together with software test

All these benefits result in practical advantages in terms of bugs found (tenths of bugs – someone critical - found also in mature IPs) and drastically reduction of the verification effort.

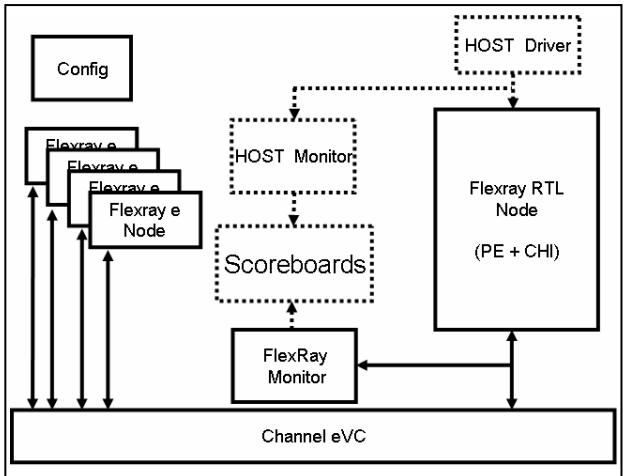

Figure 10: FlexRay network-level verification

6. Conclusions

There are two issues still pending: "Who's guarding the guards," in other words, how to assess the quality of the verification flow, and determining the link with post-silicon and on-field testing. Verification automation's quality is based both on the maturity of the tools and on the experience of the engineers using them. High-level languages and the use of detailed methodologies help system engineers. However, the crucial point remains the verification plan where each coverage item should be defined and pondered. Efficiency and execution time of verification tools allow the use of massive regressions enabling "think & try" approaches. However blind verification is a practice of the past and a modern verification engineer should always be fully aware of what he or she is verifying.

Concerning the link with post-silicon and on-field testing, as seen in the CAN example, a first answer is the generation of functional patterns out of the system-level verification environment. Latest innovations in verification automation foresee a tighter integration providing the system engineer with hardware boxes that emulate, accelerate, and drive the real silicon with the same inputs of the verification environment. However, system engineers are still looking forward to a seamless complete integration.

In brief, only a combined use of modern verification automation tools and high quality verification components can surmount verification barriers and cross the layer boundaries, providing system engineers with a systematic and controlled verification flow, able to fulfill the strict functional testing rules required by safety norms, such as IEC61508.

7. References

[1] R.Mariani, “Breaking the verification barriers “, Automotive Design Line, Feb 2005

[2] CEI International Standard IEC 61508, 1998-2000

[3] LIN Specifications 2.0, LIN Consortium, http://www.lin-subbus.org/

[4] LIN eVC datasheet, www.yogitech.com

[5] IEEE 1647: http://www.ieee1647.org/index.html

[6] A. Pizali, “Functional Verification Coverage Meaurement and Analysis”, Springer Edition, 2004

[7] AHB verification component, www.cadence.com

[8] eRM Methodology, www.cadence.com

[9] CAN specification, CAN CIA, http://www.can-cia.org/

[10] CAN Reference Model and Testbench, BOSCH, http://www.semiconductors.bosch.de/de/20/can/index.asp

[11] ISO Norms, http://www.iso.ch

[12] A.Di Blasi, F.Colucci, R.Mariani, “Y-CAN Platform: A Re-usable Platform for Design, Verification and Validation of CAN-Based Systems On a Chip”, ETS-2003 Symposium, May2003

[13] C.Turner, C.Mueller, “Re-usable CAN IP block”, CAN newsletter 3/2002 CIA

[14] CAN eVC datasheet, www.yogitech.com

[15] FlexRay consortium, http://www.flexray.com