# An Efficient Test and Diagnosis Environment for Communication Controllers

Eric Armengaud, Andreas Steininger
Vienna University of Technology
Embedded Computing Systems Group E182-2
Treitlstr. 3, A-1040 Vienna
{armengaud, steininger}@ecs.tuwien.ac.at

Martin Horauer
University of Applied Sciences Technikum Wien
Dept. of Embedded Systems
Höchstädtplatz 5, A-1200 Vienna
horauer@technikum-wien.at

## Abstract

*Testing is of utmost importance at several development stages of electronic systems; this is especially true for tightly coupled distributed embedded systems. This paper presents a test environment that allows to log bus-traffic of an automotive cluster relying on the emerging FlexRay bus protocol. While a real-life cluster allows for fast test pattern application, only limited insight can be shed onto the internals of the nodes for diagnosis. Our approach allows to efficiently switch between hardware and simulation by mapping the logged real-life data to a HDL simulation testbench for the system-under-test. This allows us to gain deeper insights and eventually reveal the causes of the faults observed and recorded on the real hardware.*

## 1 Introduction

Electronics, nowadays, is the driving factor for almost all innovations in the automotive domain. To manage the traffic of these emerging distributed embedded systems an industrial consortium of leading automotive and electronic OEM suppliers introduced the FlexRay protocol, see [1]. Relying on both the time- and event-triggered paradigms, FlexRay promises reliability and fault-tolerance aspects with the bandwidth to serve the needs of a communication backbone and the flexibility for the coupling of sensor/actuator systems required for future automotive solutions.

Since FlexRay will be introduced for safety-critical applications, e.g. *X-by-wire* systems, where a failure can lead to severe consequences, means for the evaluation of dependability properties are required. Clearly, testing is essential in order to evaluate whether the system is correctly implemented and will react as expected in its future field environment. However, while methods for testing of the computing nodes themselves on the one side and the bus on the other side do exist, a unified, accurate and systematic test approach on the system level is required that does not only consider the function of these singular components in isolation. Experience shows that problems with interaction of "fault-free" components are becoming increasingly relevant in practice. The problem is further aggravated by the large number of new product variants.

This paper presents a method for the efficient verification and diagnosis of communication controller hardware that was developed in the course of our STEACS[1] project. In particular, our structured method enables fast and efficient switching between a physical prototype cluster and a HDL simulation model. This is of utmost importance since the physical prototype on one hand permits the validation of hardware models using fast and extensive quantitative tests, while on the other hand the HDL simulator provides detailed insight into the module in selected problem cases and thus enables accurate diagnosis.

Sec. 2 details the setup of our system under test and the environment we have developed for prototype tests. Next, in Sec. 3 we present our test environment for the HDL simulation and explain the efficient mapping we created between the hardware and simulation environments. In Sec. 4 we present some experimental results conducted with our setup before we conclude our paper in Sec. 5.

## 2 Test Environment - Prototype

Our system-under-test (SUT) is a COTS FlexRay cluster running a dedicated application, cf. Fig. 1. To enable remote debugging and testing, we further attached two customized tester nodes termed BusDoctor to the FlexRay bus. This embedded test solution provides capabilities for monitoring FlexRay bus-traffic on one hand and allows to inject stimuli into the data stream on the other hand. In particular, our FPGA-based solution employed in the BusDoctor nodes allows to record and/or manipulate data at different levels of abstraction (e.g. by handling encoded/decoded bits next to the physical layer or entire frames at the data link layer).

Each of these two embedded testers is controlled and configured by a remote host using a standard Ethernet connection. In particular, when monitoring is activated the host creates a logfile with a trace of the bus traffic at the abstraction levels of interest. Similarly, via the replay functionality it is possible to inject stimuli stored in a logfile directly into the ongoing bus traffic. Notice that a bus traffic scenario can be completely described by such a trace in a logfile. Hereby, our FPGA implementation extracts relevant information to decide at which particular layer the traffic shall be inserted

[1]The STEACS-project received support from the Austrian "FIT-IT[embedded systems" initiative, funded by the Austrian Ministry for Traffic, Innovation and Technology (BMVIT) and managed by the Austrian Research Promotion Agency (FFG) under grant 807146.
See http://embsys.technikum-wien.at/steacs.html for further information.

and when this shall be done. The latter trigger functionality is essential when valid data is to be inserted into the time-triggered portion of the bus traffic and hence needs to be aligned appropriately.

Finally, a central tester controls the hosts in turn, configures and controls the system-under-test, and allows to start and stop the experiments. Additionally, it provides tools for automatic test reporting and statistical analysis. Furthermore the central tester may provide a "BDFileGenerator" (cf. [2]) to artificially generate a logfile instead of recording it, and a "LogChanger" to alter an existing logfile, which for example can be used to inject faults in order to activate and test fault tolerance mechanisms.

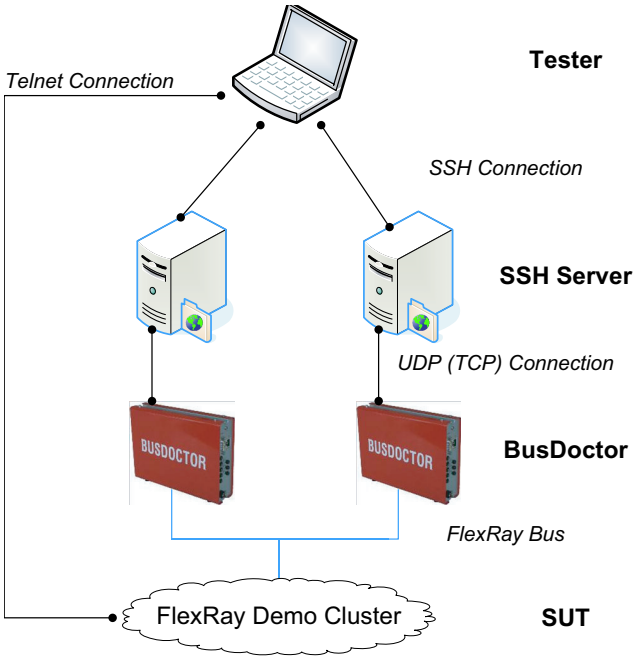


Figure 1: Prototype Test Environment

The BusDoctor's internal architecture is basically an SoC solution implemented on an Altera Excalibur FPGA. The FPGA resembles a hardwired processor stripe centered around an ARM processor core with several interfaces (UART, memory interfaces, etc.). The remaining configurable FPGA resources host our monitoring and injection hardware solution. The latter implements the FlexRay protocol stack and provides "by-paths" to enable the recording of incoming data streams and to inject data into the outgoing data-streams at different levels of abstraction, respectively. All recorded and injected data are encapsulated in simple, dedicated frames that are transferred to the ARM processor via a DPRAM interface. A dedicated bus arbiter controls the interface to/from the monitoring and injection modules towards the DPRAM, see [4]. An RTAI Linux operating system firmware executing on top of the ARM processor handles the transfer to/from the DPRAM onwards either to large flash disks and/or a remote host via a standard Ethernet interface employing the TCP/IP protocol, cf. 2.

# 3 Test Environment - HDL Simulator

The modified FlexRay controller that forms the core of the BusDoctor's hardware functionality is implemented in VHDL. Therefore, a COTS HDL simulator has been used to test and debug the design. The main problem we faced was to develop a model that provides a realistic and representative environment of the FlexRay bus on one side and the host on the other side. For the purpose of testing the FlexRay controller portion the interface to the host could be moved down to the configuration and data interface (see Fig. 2), thus reducing the simulation complexity. (By doing this we in fact assumed the hardwired ARM core to be fault free at this point.)

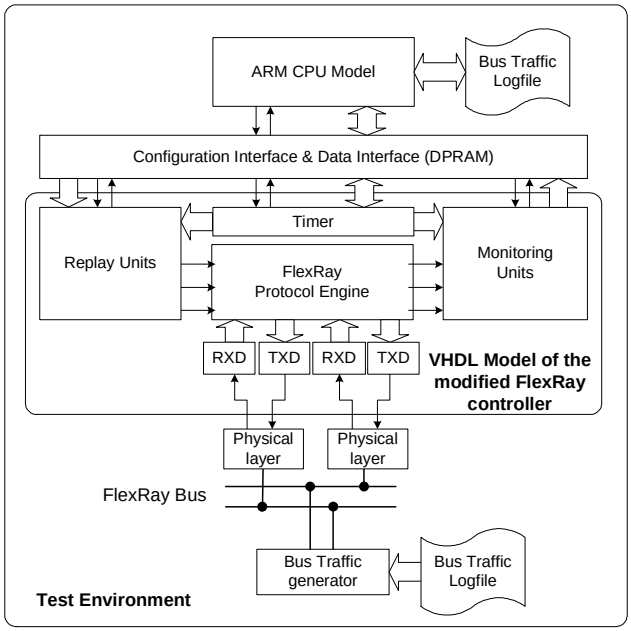


Figure 2: Simulation Test Environment

Consequently, we developed simulation models for the host side (ARM CPU and configuration and data interface) and for the FlexRay side (physical layer and bus traffic generator). The ARM CPU model could quite conveniently be reduced to its memory access for configuration and control of the VHDL design. In addition, a function to transfer the data between a logfile and the VHDL model was designed. This (1) provides a well defined bound between configuration and data, and (2) eases the generation of scenarios by the use of automated tools. On the FlexRay side, a bus traffic generator module was developed to stimulate the FlexRay bus. To make the tedious process of test pattern generation more efficient, we have developed a converter that allows us to feed this generator from a bus traffic logfile.

During the development of the BusDoctor we set emphasis on an efficient data encapsulation. This led us to a strict separation between control information and data. Data values are stored in a logfile and describe the information exchanged at the communication medium completely defining

an application scenario that can be transparently used at every stage for verification. In particular, a bus traffic logfile consists of the complete description of the bus traffic behavior. The communication scenario is divided into packets, each consisting of an identifier, a packet length descriptor, a timestamp and the actual packet contents, see also [3]. The packet identifier uniquely describes the considered channel and abstraction level. Next, the length information is useful when packets have to be filtered out. The timestamp depicts the starting point of the packet, and finally the packet content describes the transmitted message with a specified granularity. Tab. 1 shows an example of a bus traffic logfile: The first packet illustrates a packet at the frame contents level and a second one at the bit level.

| | |
|---|---|
| Identifier: | "Frame level identifier channel A" |
| Length: | length of the frame in bytes ($0 - 2^8$) |
| Timestamp: | 32 bit timestamp; granularity 40 ns |
| Content: | data payload |
| Identifier: | "Bit level identifier channel B" |
| Length: | length of the frame in bytes ($0 - 2^{16}$) |
| Timestamp: | 32 bit timestamp; granularity 40 ns |
| Content: | sampled binary values |
| ... | ... |

Table 1: Example of a bus traffic logfile

Within the STEACS project we developed different tools to generate new bus traffic scenarios where the resulting trace can be freely used either for simulation purposes or for direct tests on a physical prototype cluster. This common scenario description builds a very comfortable bridge between simulation and prototype test. Indeed, situations captured with the BusDoctor hardware can be easily transferred to the simulation environment for closer analysis as required for debugging, and in the reverse direction, simulated behaviors can be easily validated in hardware as well. This common interface consequently enables efficient interaction between prototype tests (fast and quantitative tests) and simulation (more accurate tests that are useful for detailed offline debugging).

# 4 Test Application

To illustrate the usefulness of our approach we conducted several test campaigns on the FlexRay protocol. The bus traffic for this communication protocol is divided into "communication cycles" that represent a repeating medium access scheme according to a configured schedule. A communication cycle is further divided into a static part and a dynamic part to enable both static TDMA-based and dynamic arbitrated bus communication within one single communication cycle. More information is available in the FlexRay protocol specification [1].

The aims of our campaigns were (1) to test the error detection mechanisms available in every standard FlexRay communication controller according to the FlexRay protocol specification, see [1], and (2) to test the additional services provided by the BusDoctor hardware for efficient debugging. Four flags are defined within the FlexRay specification, while our BusDoctor implements 16 flags in order to provide more accurate fault diagnosis. Tab. 2 lists the flags that are available both within a standard communication controller and our BusDoctor, respectively.

| **Standard FlexRay controller** | |
|---|---|
| vSS!ValidFrame | indicates the validity of a frame |
| vSS!SyntaxError | signals a syntax error: e.g. a coding error; an erroneous frame start or end sequence |
| vSS!ContentError | reports a frame content error: e.g. an erroneous frame identifier or a wrong cycle length |
| vSS!BViolation | indicates a boundary violation |
| **BusDoctor** | |
| CODERR | Coding Error |
| TSSVIOL | Transmit Start Sequence Error |
| HCRCERR | Header CRC Error |
| FCRCERR | Frame CRC Error |
| FESERR | Frame End Sequence Error |
| SYMB | Symbol |
| VCE | Valid Communication Element |
| BVIOL | Boundary Violation |
| SWVIOL | Symbol Window Violation |
| NITVIOL | Network Idle Time Violation |
| SOVERR | Slot Overbooked Error |
| NERR | Null Frame Error |
| SSERR | Sync or Start-up flag Error |
| FIDERR | Frame Identifier Error |
| CCERR | Communication Cycle Error |
| SPLERR | Static Payload Length Error |

Table 2: Fault Detection Flags overview

For the generation of the bus traffic scenario logfile we engineered two solutions; (1) using the BusDoctor to record bus-traffic or (2) using the "BDFileGenerator" tool to efficiently generate artificial, specification conformous bus-traffic. Afterwards, these logfiles can be easily altered by means of our "LogChanger" to violate certain conditions and thus stimulate/emulate faults.

The presented test campaign consisted of 15 experiments where each one consisted of 3.000 to 5.000 communication cycles and, hence, represented 9 to 15 seconds of bus traffic. For every experiment we injected something between 46 and 1092 deviations. A dedicated test application was developed to analyze the monitored bus traffic and report the error flags. Additionally, the test flags were made visible at external pins for the standard FlexRay controller implementation as well. Tab. 3 summarizes the test campaign and the obtained results.

As can be seen in Table 3 all error flags from the listing in Table 2 were activated at least once both for the standard FlexRay communication controller and the BusDoctor. The first five experiments aimed at altering the frame structure (defined bit sequences acting as delimiters or pro-

| Exp. | # of deviations | # of cycles | Description | FlexRay status | BusDoctor status |
|---|---|---|---|---|---|
| 1 | 234 | 5000 | Byte Start Sequence error | vSS!SyntaxError | CODERR |
| 2 | 234 | 5000 | Transmit Start Sequence error | vSS!SyntaxError | TSSVIOL |
| 3 | 234 | 5000 | Header CRC error | vSS!SyntaxError | HCRCERR |
| 4 | 234 | 5000 | Frame CRC error | vSS!SyntaxError | FCRCERR |
| 5 | 234 | 5000 | Frame End Sequence error | vSS!SyntaxError | FESERR |
| 6 | 1092 | 5000 | Boundary violation (pos. offset) | vSS!BViolation | BVIOL |
| 7 | 468 | 5000 | Boundary violation (neg. offset) | vSS!BViolation | BVIOL |
| 8 | 46 | 3000 | Symbol Window Violation | vSS!BViolation | SWVIOL |
| 9 | 46 | 3000 | Network Idle Time Violation | vSS!BViolation | NITVIOL |
| 10 | 78 | 5000 | Several frames within a slot | vSS!BViolation | SOVERR |
| 11 | 78 | 5000 | Null frames within dynamic slots | vSS!ContentError | NERR |
| 12 | 156 | 5000 | Sync or startup bit in dynamic slots | vSS!ContentError | SSERR |
| 13 | 78 | 5000 | Frame Identifier error | vSS!ContentError | FIDERR |
| 14 | 46 | 3000 | Cycle counter error | vSS!ContentError | CCERR |
| 15 | 234 | 5000 | Static Payload Length error | vSS!ContentError | SPLERR |

Table 3: Test Campaign Summary

tection within the frame) to produce syntax errors. Then, the goal of experiments 6 to 10 was to produce boundary violations in the time domain, i.e. to shift communication elements above the boundaries predefined by the communication schedule. Finally, experiments 11 to 15 aimed at altering frame contents to produce errors at higher protocol layers (such as, e.g., frame identifier mismatch). Our BusDoctor provided us with more accurate fault diagnosis due to the larger number of status flags available.

The whole campaign lasted approximately 5 minutes in hardware and can be easily repeated to verify a new chip implementation. In our case this campaign permitted us to identify and fix a problem in the implementation of the fault detection mechanisms of the BusDoctor. The presented test environment enabled us to directly re-use the bus traffic for the HDL simulation. Thus, the fault could be easily traced and fixed.

# 5 Conclusion

This paper presented a test environment that allows to capture real-life bus-traffic of a distributed automotive cluster with the help of dedicated diagnosis nodes. Typically, the logged data can be (1) used for replay to enable debugging with the cluster at hand and (2) applied to a HDL simulation model to gain deeper insights on the effects that caused an error. Optionally, this logfile can be artificially generated and/or modified. The strength of our approach is to allow fast application of test patterns in hardware while at the same time facilitating diagnosis by means of simulation.

# References


[1] Flexray Communications Systems - Protocol Specification Version 2.0. FlexRay Consortium, 2004.

[2] E. Armengaud, A. Steininger, and M. Horauer. Efficient Stimulus Generation for Testing Embedded Distributed Systems – The FlexRay Example. $10^{th}$ *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA05)*, September 2005. (to appear).

[3] E. Armengaud, A. Steininger, M. Horauer, R. Pallierer, and H. Friedl. A Monitoring Concept for an Automotive Distributed Network - The FlexRay Example. $7^{th}$ *IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS'04)*, pages 173–178, April 2004.

[4] M. Horauer, F. Rothensteiner, M. Zauner, E. Armengaud, A. Steininger, H. Friedl, and R. Pallierer. An FPGA based SoC Design for Testing Embedded Automotive Communication Systems employing the FlexRay Protocol. *Proceedings of the Austrochip 2004 Conference*, pages 119–125, September 2004.