Accelerating Validation of Time-Triggered Automotive Systems on FPGAs

Shanker Shreejith*†, Suhaib A. Fahmy* and Martin Lukasiewycz†
*School of Computer Engineering
Nanyang Technological University, Singapore
Email: {shreejit1,sfahmy}@ntu.edu.sg
†TUM CREATE, Singapore

***Abstract*—Automotive systems comprise a high number of networked safety-critical functions. Any design changes or addition of new functionality must be rigorously tested to ensure that no performance or safety issues are introduced, and this consumes a significant amount of time. Validation should be conducted using a faithful representation of the system, and so typically, a full subsystem is built for validation. We present a scalable scheme for emulating a complete cluster of automotive embedded compute units on an FPGA, with accelerated network communication using custom physical level interfaces. With these interfaces, we can achieve acceleration of system emulation by $8\times$ or more, with a systematic way of exploring real-world issues like jitter, network delays, and data corruption, among others. By using the same communication infrastructure as in a real deployed system, this validation is closer to the requirements of standards compliance. This approach also enables hardware-in-the-loop (HIL) validation, allowing rapid prototyping of distributed functions, including changes in network topology and parameters, and modification of time-triggered schedules without physical hardware modification. We present an implementation of this framework on the Xilinx ML605 evaluation board that integrates six FlexRay automotive functions to demonstrate the potential of the framework.**

I. Introduction

Modern vehicles can be considered as distributed mobile computing platforms, with top-of-the-range vehicles incorporating up to 100 embedded compute units with functions ranging from vehicle dynamics to passenger comfort. The embedded hardware is connected through a heterogeneous network communicating using multiple protocol specifications, with gateways between them. This growing complexity makes iterative design improvements more difficult, as any changes to the network must not impact the performance and safety of existing nodes. For more cutting-edge safety-critical systems, like drive-by-wire, the risks are even greater.

Relying on high-level simulations of such systems is insufficient, as they often fail to capture low-level network communication issues that can have a significant bearing on overall system performance. One common method is to integrate a hardware cluster consisting of multiple devices, as shown in Fig. 1, with the sensors and actuators in a hardware-in-the-loop (HIL) setup. HIL tests enable the design team to extensively test and certify the functions and to verify the network topology, parameters, and communication schedules by using a faithful representation of the cluster in a controlled setting. For top-of-the-range models, this can require setup of

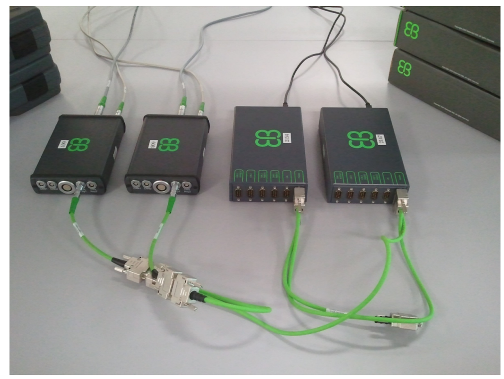

Fig. 1: Lab test setup for 4 ECUs.

a cluster comprising a high number of compute units (ECUs), sensors and actuators, along with the communication network.

As the number of nodes in vehicles continues to increase, these laboratory setups become more complex, and as a result, restrict the space of configurations that can be explored. Even small modifications to the setup can require a significant amount of reworking, e.g., reconfiguring the timing parameters of all distinct nodes on the network.

We propose to model an entire cluster of ECUs on a single FPGA, using real implementations of the compute units, and communication infrastructure. This allows us to conduct more extensive hardware in the loop validation, with easier modifications to the system setup, the ability to test for robustness to physical issues, and a guarantee of faithfulness to the actual implementation.

In this paper, we describe a verification platform using a single FPGA that integrates a complete compute cluster of embedded compute nodes along with their network infrastructure. We use the FlexRay time-triggered automotive network protocol, with interfaces we have built for use in discrete FPGA compute units, allowing for cycle-accurate simulation. In addition to this, we are able to leverage optimisations in the interfaces to offer $8\times$ or more speed-up in simulation, in addition to testing for real world issues like clock jitter and node failures. The approach can be adapted for other time-triggered protocols.

The remainder of this paper is organised as follows: In Section II, we give an overview of similar approaches described in the literature. Section III outlines the general idea of HIL testing and how our approach offers added advantages. Section IV details the software and hardware architecture of the proposed platform. We also describe our approach for achieving acceleration in the verification flow. In Section V, we present a case study that showcases the platform's capabilities. Finally, Section VI concludes and outlines our future work.

II. Background

Computational infrastructure in a modern vehicle is organised into different segments or domains based on bandwidth requirements and the criticality of the relevant functions. Each domain is served by a communication protocol that balances the reliability requirements and cost considerations for the intended tasks [1], [2], [3]. For simple tasks like door and window controls, a protocol like Local Interconnect Networks (LIN) is used, while more complex control functions might use Control Area Networks (CAN). Functions which are deemed safety-critical are served by high performance protocols which provide redundancy and timeliness like the FlexRay protocol, which is gaining ground as the de facto protocol for safety-critical applications. Similar protocols based on time-triggered Ethernet are also under development, and the proposed approach can be adapted to them. In-vehicle systems hence represent a heterogeneous and complex network of distributed functions.

Compute functions are usually implemented as software on commercial processors, to provide flexibility and easy upgradability. FPGAs have been used for evolving high performance applications such as computer vision, and this role is expected to grow as computational requirements increase [4]. Some potential advantages of FPGA-based computing for safety-critical and non-safety-critical vehicular applications are discussed in [5]. In [6], the authors describe approaches for implementing FPGA-based ECUs for general and safety-critical applications which are completely compliant with AUTOSAR standards. Dynamic and partial reconfiguration of FPGAs in automotive applications has also been discussed. In [7], the author proposes runtime reconfiguration (complete) of FPGAs in the presence of errors to enable hardware-level fault-tolerance. In [8], we leverage partial reconfiguration as part of a proposed fault-tolerant scheme for compute units on a FlexRay bus. In [9], the authors describe an FPGA-based framework which enables Ethernet-based networks for real-time applications with predictable latencies, while coexisting with standard non-deterministic traffic.

Even as they find more widespread use, with falling costs and improving capabilities, FPGAs have long been used for rapid prototyping and validation of complex applications. When simulating a system in software, high level models are typically used to enable these simulations to run in reasonable time. This, however, means that such models do not incorporate all the details that might be necessary for a truly accurate simulation. FPGAs offer us the opportunity to mimic the exact hardware system, offering cycle- and bit- accurate simulations orders of magnitude faster than would be possible in software. Hence, they are widely used for validating processors, systems-on-chip and other complex systems. Architectures for FPGA-based ASIC emulation and co-verification for embedded systems are described in great detail in [10], [11]. One significant challenge in such cases is the limited I/O pin count of FPGAs, which is addressed in [12], where the authors propose a technique called *virtual wires*. FPGA-based emulation is also widely employed in validation of critical communication networks, where the use of reconfigurable fabrics provides significant advantages in speed and accuracy [13].

In the automotive domain, FPGA-based computing units and interfaces with special test capabilities have been proposed for validating novel functions [14]. This allows injection of real-world errors during validation, which would otherwise require more complex hardware in the test setup. FPGAs have also been used for validating vehicle-to-vehicle (V2V) communication, serving as channel emulators for evaluating wireless transceivers [15].

To the best of the authors' knowledge, there is no prior work that investigates the integration of a network cluster on a single FPGA for validation purposes, despite the significant advantages it offers.

III. General Concept

Hardware-in-the-loop tests are used to profile the performance of a system under real-world conditions. This is done by mimicking the real world implementation scenario, including the network and other communicating devices. Conducting such tests is essential when modifying parts of the automotive network, as this establishes whether these changes can be tolerated within the context of the exiting system and its network properties, without impacting performance and reliability. As shown in Fig. 1, a typical setup would include multiple compute units and their network interfaces, with lengths of cable to mirror connectivity in the vehicle, along with sensors and actuators or models of them. For a reasonably sized subsystem, this setup can be cumbersome; analysing the state of the system involves collecting data from many discrete nodes, and any changes might involve complex re-assembly. With hybrid and full electric vehicles (EVs) gaining popularity, computational complexity in future vehicles is expected to increase substantially as more safety critical systems, like drive-by-wire, are incorporated. Existing validation approaches do not scale well for such systems.

Rather than build the test system from discrete components, we propose to model the subsystem using actual implementations of the compute functions, network interfaces, network topology, and sensor/actuator interfaces on a single large FPGA, as shown in Fig. 2. A unified external interface to the outside world allows sensors and actuators or models of them to be attached easily.

A key benefit of this approach is that multiple distinct configurations, and even subsystems can be evaluated on the same platform, simply by reconfiguring the FPGA. This results in reduced setup times and a faster turnaround for experiments.

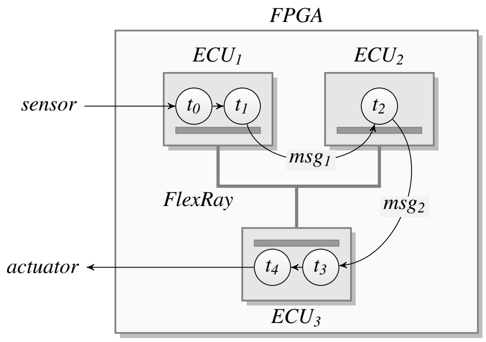

Fig. 2: Hardware in the loop test setup.

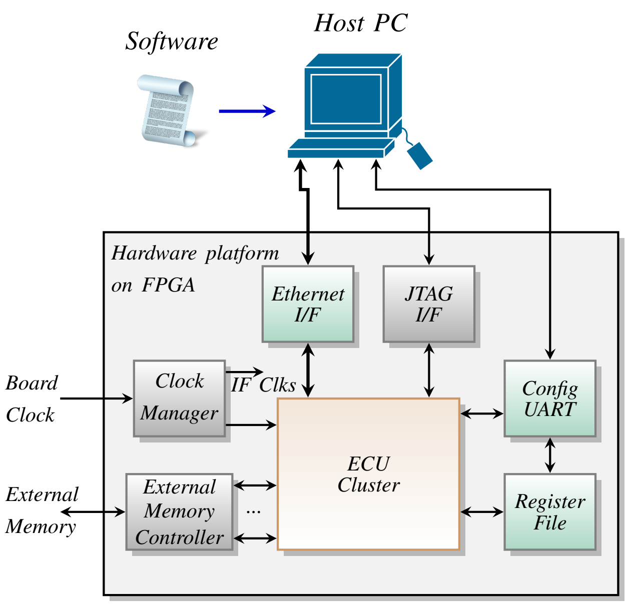

Fig. 3: System architecture of the validation platform.

The afforded flexibility also opens the door to more complex design space exploration and optimisation of ECU architectures, interconnect topologies, and communication schedules, allowing the designer to more confidently identify a robust combination for a given subsystem. In this paper, we focus on presenting a scalable and configurable platform on FPGAs that enables high speed verification of automotive systems with simplified software control. This framework can also address the rising challenge of architectural optimisation and design space exploration for evolving functions and networks in the automotive domain generally.

IV. System Architecture

The complete verification platform comprises a standard host PC connected to a commercial FPGA board, as shown in Fig. 3. Software on the host PC controls and configures the FPGA board over standard Ethernet, JTAG, and UART interfaces. The FPGA board integrates a cluster of isolated compute units or ECUs connected through a dual-channel FlexRay communication network. Each ECU may implement a specific automotive function, like park assist or adaptive cruise control, either as a hardware implementation or as software running on a soft processor. Interfaces within the FPGA are

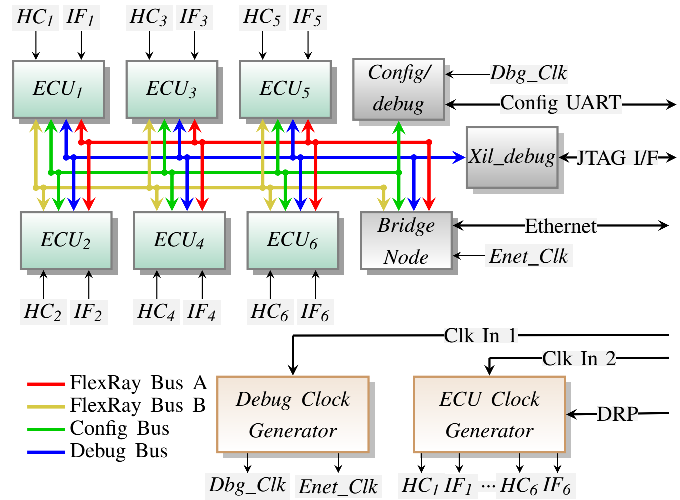

Fig. 4: Hardware architecture of the validation platform.

built using hard macros or optimised IP to minimise overhead, and allow for complex ECUs.

A. Hardware Architecture

Fig. 4 shows the architecture of the validation platform, with the external interfaces and clock domains. The figure describes an example design comprising 6 independent computing units marked ECU_1 to ECU_6. Each ECU is fed with independent isolated clocks, marked as HC_x for ECU_x generated by the hardware clock manager. Each IC_x clock is the interface clock for the FlexRay interface within ECU_x, enabling the interfaces to be clocked at a different frequency from the ECU core. The interfaces enable communication between ECUs (over the FlexRay bus) and can be controlled by the host PC.

1) Host Interfaces and Global Registers: The host PC can communicate with the ECUs over the shared UART *Config/debug* interface, the *Xil_debug* JTAG interface and the *Ethernet interface*, as shown in Figures 3 and 4. The host PC controls the FPGA platform by accessing the global registers (register file) over the *Config/debug* interface. The *Xil_debug* JTAG interface enables initialisation and debugging of Microblaze-based ECUs in the cluster, using the Microblaze debugger module (MDM). The host PC uses the Ethernet interface as a real-time debugger for monitoring the state of the FlexRay bus and selected control signals from the ECUs in the cluster.

The Register File implements a set of global registers that are used to configure the interfaces, set platform parameters, and control/configure the special test features. The functionality of each register is described in Table I. The control/configuration registers are used for enabling/disabling the platform, enabling test cases and to configure the operation modes. The *Config UART* is also mapped in the memory space of each ECU, and thus doubles as a debug interface. This enables host software to access debug data and registers using the *ECU Address Registers* as an indirect addressing register.

2) ECU architecture: Compute units implement automotive functions either as hardware functions, or as software on

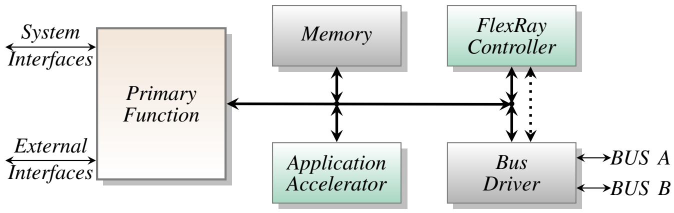

Fig. 5: Example ECU architecture.

a soft processor. Fig. 5 illustrates one such model for an ECU. As would be the case in a vehicle, each ECU is completely independent and implemented in isolation. The primary function represents the functional implementation of an automotive algorithm in either software or hardware. The application accelerators are dedicated hardware units like FFT modules or FIR filters that leverage FPGA specific hardware like DSP blocks, or other custom hardware designed for a specific application. The ECU memory is built using Block-RAMs. ECUs also incorporate a dual channel FlexRay Communication Controller (CC) which implements the FlexRay communication protocol. A key aspect of this platform is that each ECU uses a fully featured communication controller for verification, just as would be the case in the final deployment.

The ECUs may interface with subsystems like sensor modules over standard interfaces like SPI, I2C, or other system interfaces. ECUs may also interface with external storage elements like non-volatile memories or high-speed DRAMs. The external memory controller provides multi-channel access to such storage elements, and ECUs can connect to it over a

TABLE I: Register file description.

Address	Function	Description
0x00	Version register	H/W version number
0x01	Platform control	Controls the interfaces and operative modes of the H/W platform
0x02	Platform status	Interface and mode status register for the H/W platform
0x03	Debug control	Operative modes & parameters for the debug module
0x04	Error injection	Enable/disable error modes - bit error, frame error, frame delay, frame drops
0x05	Bit error config	Specifies the bit/byte(s) to insert error
0x06	Error slot config	Specifies the Slot-ID to insert error
0x07	Delay rate config	Specifies the delay value in ticks
0x08	Frame drop config	Specifies the FlexRay cycle to drop Frame
0x09	ECU addr reg [31:16]	ECU debug : Indirect address register (upper DWORD)
0x0A	ECU addr reg [15:0]	ECU debug : Indirect address register (lower DWORD)
0x0B	ECU access control	Selects ECU and read/write for indirect access
0x0C	ECU data reg [31:16]	ECU debug data (write/read)
0x0D	ECU data reg [15:0]	ECU debug data (write/read)
0x0E	Clock jitter	Select the clock offset for IC_x - frequency/phase and offset value

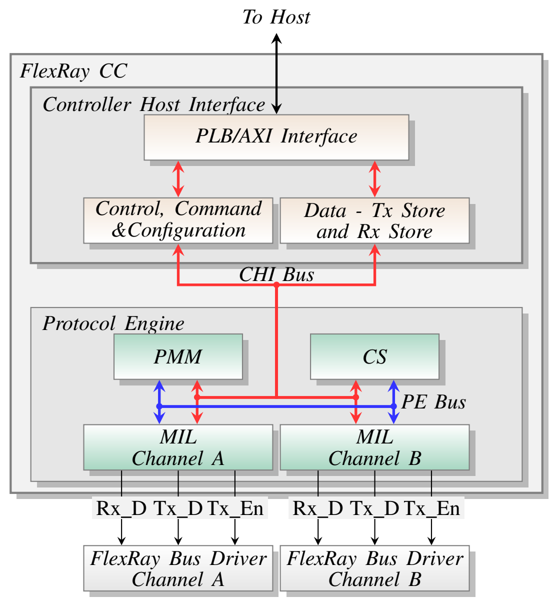

Fig. 6: Custom CC architecture.

streaming interface. It is configured to ensure that the memory spaces are isolated between the different channels.

3) The FlexRay bus and FlexRay Communication Controller (CC): The heart of the verification platform is the FlexRay bus and FlexRay CC. FlexRay is a state-of-the-art serial communication protocol for interconnecting automotive ECUs [16]. It uses a periodic communication scheme called the FlexRay cycle, where each cycle is composed of a sequence of communication slots that are statically allocated to the different ECUs at design-time. FlexRay offers support for deterministic and volume data transfers within the same cycle using a combination of fixed-size and variable-size slot-based medium access. The communication configuration of the individual ECUs determines the FlexRay parameters (global and node-level) and communication slots assigned to them. This information is embedded in each ECU and the primary function (processor or logic) initialises the controller with these values during startup.

Communication occurs over the shared medium when an ECU uses the allocated slot(s) to transfer/receive data to/from another compute unit, sensor or actuator. The FlexRay encoding scheme ensures redundancy for the transmitted data, providing protection against possible interference in the harsh automotive environment. The FlexRay CC implements the protocol specification, abstracting this information from the rest of the system.

To enable a compact implementation of the ECUs, we have developed an FPGA-optimised FlexRay communication controller that leverages FPGA-specific resources like BRAMs and DSP blocks and enables a tight integration with the primary processing function or algorithm. The controller also features a configurable set of extended capabilities which are not available in commercial off-the-shelf implementations of the FlexRay protocol. We leverage the capabilities of this controller to verify the ECU design against real world scenarios like frame errors, jitters, delays, and more. A block diagram of the custom controller is shown in Fig. 6. The controller comprises two major blocks; the *Controller Host Interface (CHI)* which interfaces to the primary function over industry standard AXI/PLB bus with burst capability, and the *Protocol Engine (PE)*, which implements the protocol specifics. The configuration, control and status space is embedded in the CHI module along with an isolated data store. To provide independence from the clock requirements of the FlexRay logic, the CHI module uses the host function clock as its primary clock.

The PE module implements the communication specification described by the FlexRay protocol. It comprises three functional sub-blocks:

1) *Clock Synchronisation (CS)* responsible for maintaining a steady local clock and periodic synchronisation with the global timebase.
2) *Medium Interface Layer (MIL)* which implements datapath functions like encoding/decoding of data streams, medium access control and framing/processing of data.
3) *Protocol Management Module* which controls and coordinates the CS and MIL modules by controlling their modes in response to host commands or network conditions, ensuring protocol compliance at all times.

The custom PE also offers configurable extended functionality like time-stamping of transmitted and received data, multicycle data handling, extensive filtering schemes for received data and security features like automatic rejection of untimely data, without intervention from the processing logic.

4) Error injection and live-debugging: The *Bridge Node* is a special ECU in the cluster that implements two distinct functions; a bus replay module (BRM) function and a bus debug module (BDM) function. The BRM allows logged data from a real experimental setup to be stored in its CC and played back within the FPGA network in a cycle accurate manner. The BRM also enables special test capabilities like bit-error injection, delay and jitter adjustment, as well as preconfigured frame drops during transmission, using the extended capabilities of the custom FlexRay CC. The BRM supports specific (like delay slot x in cycle y by t units) and pseudo random specifications for the parameters. Errors and delays can also be injected in transmissions by other ECUs in the cluster. Bus buffers controlled by the BRM are used to drop frames (specific or random) transmitted by other ECUs while bit-errors are injected by directly modifying the transmitted values on the bus.

The bus debug module enables real-time debugging of the FlexRay bus and the ECU control signals selected at design-time. The BDM captures the selected signals with a timestamp and encapsulates them into Ethernet frames which are transmitted over the Gigabit Ethernet interface. The BDM also features a trigger mode which can be configured to perform a trigger-based capture of selected signals. The Ethernet frames are decoded into value change dump (VCD) format by the debug tools and can be viewed on the host PC in real-time.

B. Software support on the host PC

The initialisation and management of the verification platform is handled on the host PC using the Python function calls listed in Table II. The individual *init_* functions can be used to initialise the platform by configuring all the ECUs, one specific ECU or the BRM module. The *mode_platform* function control the error injection capabilities of the platform, while the *mode_debug* function configures and trigger the live-debugger module (BDM).

The code for an example design comprising multiple ECUs is shown in Fig. 7. The initialisation segment creates a merged bitstream file from the tool-generated bit file by invoking the *init_platform* API call. The function merges the software (elf) for ECU_1 (mb_0) into the tp_top bit file, which is used in subsequent calls that initialise the remaining ECUs. For the final ECU, the *done_flag* is set to 1, triggering the initialisation of the FPGA with the integrated bitstream. Subsequently, tests are performed on the cluster by using the *mode_debug* and *mode_platform* function calls. A change in software or communication schedule is triggered by altering the software routine in the specific ECU, which is triggered by the *init_processor* API call in the example.

C. Accelerated Mode

A normal FlexRay bus provides a serial datapath over an unshielded twisted pair cable. To ensure data is protected in the harsh automotive environment, the FlexRay protocol imposes bit-level redundancy for all transmissions. The decoder must sample these redundant bits and majority vote over a predefined 8-bit window to decide bit polarity. This results in an actual transmission rate of 80 Mbps for the maximum FlexRay data rate of 10 Mbps. Hence, an 80 MHz sampling and transmission clock are required for the FlexRay CC. This would limit the potential overclocking possible on an FPGA to around 3 $\times$ (a frequency of 240 MHz is considered high for complex designs).

TABLE II: Software APIs.

Function	Arguements	Description
init_platform()	elf_file, processor_tag, bit_file, done_flag	Initialises the processor *processor_tag* with code *elf_file*, creates bitstream *bit_file*, downloads it to FPGA
mode_platform()	reg_file_address, reg_file_value	Modifies the register file content at address *reg_file_address* with value *reg_file_value*; alters the operating mode of platform
init_processor()	elf_file, processor_tag	Downloads the modified software *elf_file* to the processor *processor_tag* and resets it
init_BRM()	brm_file	Initialises the BRM memory with the captured FlexRay bus data and enables the BRM
mode_debug()	reg_file_value, capture_flag	Alters the behaviour of the Bus Debug Module. Can choose debug signals using *reg_file_value* and alter host data capture using *capture_flag*

```python
from init_platform import init_platform
from init_BRAM import init_BRAM
from mode_debug import mode_debug
from mode_platform import mode_platform
from init_processor import init_processor

# define global values
debug_config = #value
test_config0 = #value
...
test_confign = #value
...

# Initialisation
# merge ECU software with bitstream
# generate a merged file for first ECU
init_platform("ecu0.elf","mb_0","tp_top",0)
# use merged file for further ECUs
init_platform("ecu1.elf","mb_1","tp_top",0)
...
# set done_flag to 1 for the final ECU
init_platform(...,1)

# Run Tests
# write debug_configuration and start debug
mode_debug(debug_config,1,1)
mode_platform(reg_addr, test_config0)
...
# stop current test
mode_platform(reg_addr, 0,0)

# Modify ECU S/W
init_processor("elf_new1.elf","mb_1")
# Re-run Tests
# write debug_configuration and start debug
mode_debug(debug_config,1,1)
mode_platform(reg_addr, test_config1)
...
```

Fig. 7: Python Software Flow.

Since the entire FlexRay bus is contained within the FPGA, and we can be sure of transmission robustness, we take advantage of the 8 times serial redundancy to make the bus byte-wide. In order not to affect the protocol constraints, only the coder-decoder module within the FlexRay PHY is altered to support byte-wide transmission and reception. This relaxes the clock frequency required for the interface to 10 MHz. Alternatively, the modification allows data transmission to be overclocked to $8\times$ or more, enabling faster progression of the emulation. With *fast_mode* selected, the achievable cluster acceleration is limited primarily by the acceleration possible for the compute nodes, rather than the communication infrastructure.

For functional validation, the FlexRay CC switches its *operating mode* from normal-serial mode to fast-parallel mode when *fast_mode* is enabled in the *Platform Control* register. The ECUs are issued with a reset signal and the FlexRay interface switches to the parallel mode. The ECUs' software reads the state change and enables faster local clock configuration for the interface, thus accelerating the validation process. However, for HIL tests which should progress at line-speeds, *normal_mode* should be enabled. For *normal_mode*, the FlexRay CC switches to the normal-serial mode which offers complete compliance with the FlexRay interface specification at all levels and is configured with the actual local clock configuration. The verification/HIL tests then progresses at normal hardware speeds.

V. Test Setup and Results

To evaluate the capabilities of the platform we have implemented it on a Xilinx ML605 board that incorporates a Virtex-6 LX240T device. This implementation incorporates 6 compute units including the bridge node. Four of them (ECU_1 to ECU_4) use a Microblaze running specific algorithms as the primary function. ECU_1 and ECU_2 form part of a park-assist system with ECU_1 forming the sensor interface and ECU_2 forming the compute and actuator interface. ECU_3 and ECU_4 represent prime and standby logic for an adaptive cruise control system with dedicated hardware accelerators. ECU_5 consists of a hardware implementation of a radar interface for the adaptive cruise control system which passes sampled radar data over the FlexRay bus for processing by the primary and standby units. The *Bridge Node* mimics a centralised fault detection unit which can trigger a switch between the primary and redundant functions. The resource utilisation of the setup is shown in Table III. The commands to/from the brake and throttle actuators/sensors are collected/generated by the *Bridge Node*. This can be replaced by actual models in an HIL test setup, or connections to real actuators/sensors.

In this test setup, we have configured the ECUs to utilise on chip memory (BRAMs) rather than external memory, resulting in 88% BRAM utilisation. With this exception, the overall FPGA utilisation is below 50% and hence it is possible to integrate more ECU functions. As FPGAs continue to grow, larger clusters of ECUs can be integrated and tested. Initial experiments confirm that 10 or more ECUs can be integrated on the Xilinx VC707 evaluation board that incorporates a Virtex-7 XC7VX485T device.

A. Test Cases

In error injection tests, we inject bit-errors, and frame drops modelling disturbances and loss of communication on the cluster network. Clock drift within the system is tested by dynamically altering the clock phase and frequency of the clock generator module. We can also inject special frames to trigger specific responses from ECUs. The effects of these tests are validated against expected behaviours of the functional modules in the cluster. The FlexRay communication parameters for the system are detailed in Table IV. The consolidated test results for the different test cases described below are tabulated in Table V.

TABLE III: Resource utilisation on XC6VLX240T.

Function	LUTs	FFs	BRAMs	DSPs
ECU_1	11339	7614	77	5
ECU_2	11334	7614	29	5
ECU_3	13837	11766	87	47
ECU_4	13844	11771	87	47
ECU_5	9006	5604	13	2
Bridge ECU	12121	8479	79	5
Debug Logic	791	1010	10	0
Total	74186	56184	729	111
(%)	49%	18%	87.6%	14%

TABLE IV: Communication schedule for the cluster.

Parameters	Assigned Values
Number of Cycles	64, 1 ms per cycle
Number of Static Slots	15 at 32 macroticks each
Payload Length (Static)	2 words
Number of Dynamic Slots	71 (max)
ECU_1 Data Txn	Cycles 0 to 31 on multiple slots
ECU_2 Data Txn	Cycles 32 to 63 on multiple slots
ECU_3 Data Txn	Cycles 0 to 15,48 to 63 on multiple slots
ECU_4 Data Txn	Cycles 16 to 47 on multiple slots
ECU_5 Data Txn	All Cycles on Slot 16
Bridge_Node Data Txn	All Cycles on Slot 5

1) Park Assist System: ECU_2 of the park-assist system samples data from the sensor ECU (ECU_1) over 64 ms and computes the adjustments required to the throttle, steering and brake controls to complete the parking operation, once the park-mode is chosen by the user. The module should stop if it receives a consistent stream of erroneous sensor data (64 samples) or it fails to receive valid data over 4 consecutive communication cycles. In our first set of tests, we inject bit-errors into the sensor data transmitted by ECU_1 by altering a selected bit in the data segment of the frame for 64 data samples. It was observed that on receiving the 64th consecutive sample with error, ECU_2 transmits an error code in its data segment indicating a sensor malfunction, and ceases to provide control information to the throttle and steering ECUs. ECU_1 stays halted even if it receives a stream of error free packets, and must be restarted by re-enabling park-mode signal, as in the specification.

For the second set of tests, we drop frames from ECU_1 modelling a communication break-up between the two modules. It was observed that ECU_2 issues the error code in its data segment as in the previous case and ceases to provide control information. However, once the communication is re-established, ECU_2 resumes normal operation after receiving a complete set of error-free sensor data, as required by the design.

2) Radar front-end system: The radar front-end system is based on a 64ms frequency modulated continuous wave (FMCW) radar scanner. When enabled, ECU_5 models the radar system and transmits the received samples from the radar interface for processing by the target detection ECUs (ECU_3 and ECU_4). These ECUs detect and estimate the distance and velocity of targets from the sample data and issue control signals to the throttle and brake ECUs. The target ECUs must reject a complete data-set if any chunk of the received data is detected to have errors. In such a cycle, the ECUs must not issue any control commands corresponding to the erroneous data. Even with single bit-errors in a dataset, ECU_3 and ECU_4 flagged erroneous communication in their respective data, while providing no control input to the throttle/brake ECUs. With persistent errors, both ECUs went into fall-back mode which provides minimal functionality using available data segments.

When a communication outage was modelled by dropping frames from ECU_5, both ECUs first went into fallback mode and then to halt mode, as expected. When communication was restored, they returned to normal mode. The Bridge_Node was used to send special fault state frames addressed to the ECUs, forcing them into fallback state and performing recovery.

3) Babbling Idiot Test: A common fault in time-triggered systems is the babbling idiot fault, whereby a node transmits messages at arbitrary points in time, corrupting the bus transmission. This is modelled by forcing the Bridge_Node to transmit frames in slots not assigned to it, disrupting the communication sequence. During this test, the other nodes fail to stay in synchronisation due to the inconsistency of transmission and as required by the FlexRay protocol, they switch to the halt mode and flag a clock synchronisation error. Once restarted under normal conditions, the ECUs reset the FlexRay interface and re-integrate into the cluster, resuming normal communication.

4) Clock Drifts/Jitter: A distributed system often faces the issue of lack of synchronisation between participating nodes due to drifts of their individual clocks. In our test setup, this is modelled by altering the phase and/or frequency of the interface clocks dynamically using the dynamic reconfiguration port (DRP) found in Xilinx Clock Managers. When enabled, a predefined set of test cases that perform phase drifts and frequency drifts can be performed on the nodes. These include shifting any selected clock by 45/90/180 degrees or altering the frequency by fixed steps. Any chosen set will cause all ECUs to be reset and restarted with the selected clock combination.

During our experiments, we observed that phase variations on the clock were absorbed by the FlexRay clock correction mechanisms, while frequency variations of more than 10% caused nodes to go out of sync, without being able to recover from the error.

B. Acceleration Tests

Accelerating the emulation process is one of the key advantages of our platform. In normal mode, the ECUs are run at 50 MHz (Microblaze clock), while the FlexRay network is running at full capacity (10 Mbps data rate). The *platform control* register enables the clock frequency to the FlexRay interfaces and ECUs to be modified. When *fast_mode* is selected, the debug waveform shows the ECUs being reset, and clocked with the new frequencies, with the parallel communication interface enabled. Normal communication was established over the parallel bus, as required by the FlexRay protocol, with the interface now being clocked at 10 MHz. By choosing an acceleration of $2\times$ in the *platform control* register, the ECUs were fed with $2\times$ clocks by the clock manager, enabling the function and interfaces to run twice as fast as in the normal case. With $4\times$ acceleration, Microblaze-based ECUs, now clocked at 200 MHz, were at the limits of what is supported. At this speed, reliable and error-free communication was established between all ECUs. Microblaze-based ECUs were limited from further acceleration, and only the logic-based ECU_5 was able to run at $8\times$.

TABLE V: Test results.

Test Case	Expected	Observed
Bit-error	Park assist: ECU_2 fault message	ECU_2 transmits error code indicating sensor fault
	Radar system: reject dataset	Rejects data & flags error; fall-back mode with persistent errors
Frame drop	Park assist: ECU_2 fault message	Transmits error code, recovers when communication re-established.
	Radar system: fall-back then halt	Switches to fall-back mode & halts with persistent errors
Special Frames	Radar ECU: fall-back, then recovery mode	Radar ECU decodes error, triggers fall-back mode & initiates recovery
Random Txn	Synchronisation error	All ECU's flag sync error & halts; recovers on reset
Clock Drift: Phase	Phase drifts absorbed by FlexRay clock synchronisation	
Clock Drift: Frequency	For drifts $> 10\%$ interface loses synchronisation & halts, unable to recover	
Acceleration	Fast_mode ($1\times$)	Parallel data mode enabled, normal communication
	Fast_mode ($2\times$)	Two times acceleration in response & communication
	Fast_mode ($4\times$)	Four times acceleration over normal mode
	Fast_mode ($8\times$)	ECU_1 to ECU_4 fail, ECU_5 operates at $8\times$ speed

VI. Conclusion

Complex test setups, comprising multiple ECUs and large amounts of communication infrastructure, are required to validate and certify the advanced computational functions in modern vehicles within a realistic environment. Such setups are cumbersome, and are not amenable to exploration of a wide range of system architectures.

In this paper, we have presented a modular scheme for implementing a cluster of ECUs on a commercial FPGA, with special testing capabilities for emulating real world conditions. By replicating the network and interfaces on the FPGA, we ensure that our test setup is completely compliant with automotive specifications. We have presented a proof-of-concept implementation on a Xilinx ML605 development board, integrating six computing units. The cluster/network is completely contained within the FPGA, allowing us to achieve $8\times$ or more speedup in functional validation, using custom modifications to the ECU network interface. We also have the ability to introduce common network errors like clock jitter, frame drops, bit-errors among other possibilities, all of which can be controlled and monitored from the host PC using software APIs. Moreover, any infrastructure modification can be made with a complete reconfiguration on the FPGA, enabling easier migration to evolving standards.

We are working on extending this setup to support larger FPGAs (Virtex-7 series) and multi-FPGA setups that would enable integration of larger clusters. We are also investigating how to integrate standardised interfaces to external high performance processors, compute units and clusters, using the interfaces available on the FPGA. Finally, we aim to adapt our FlexRay-based scheme to evolving automotive networks like synchronous Ethernet, for which the higher layer protocols have yet to be defined, and we would like to add support for heterogeneous network setups.

Acknowledgment

This work was supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

References

[1] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proc. of the IEEE*, vol. 93 Issue 6, p. 1204 to 1223, 2005.

[2] T. Forest, A. Ferrari, G. Audisio, M. Sabatini, A. Sangiovanni-Vincentelli, and M. Di Natale, "Physical Architectures of Automotive Systems," in *Proc. of the Design Automation and Test in Europe (DATE) Conference*, 2008, p. 391 to 395.

[3] C. A. Lupini, "In-Vehicle Networking Technology for 2010 and Beyond," *SAE2010 - World Conference & Exhibition*, 2010.

[4] M. Lukasiewycz, S. Steinhorst, S. Andalam, F. Sagstetter, P. Waszecki, W. Chang, M. Kauer, P. Mundhenk, S. A. Fahmy, S. Shanker, and S. Chakraborty, "System Architecture and Software Design for Electric Vehicles," in *Proc. of the Design Automation Conference (DAC)*, 2013.

[5] S. Shreejith, S. Fahmy, and M. Lukasiewycz, "Reconfigurable Computing in Next-Generation Automotive Networks," *Embedded Systems Letters, IEEE*, vol. 5, no. 1, p. 12 to 15, 2013.

[6] F. Fons and M. Fons, "FPGA-based Automotive ECU Design Addresses AUTOSAR and ISO 26262 Standards," *Xcell journal*, vol. Issue 78, p. 20 to 31, 2012.

[7] N. Chujo, "Fail-safe ECU System Using Dynamic Reconfiguration of FPGA," *R & D Review of Toyota CRDL*, vol. 37, p. 54 to 60, 2002.

[8] S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz, "An Approach for Redundancy in FlexRay Networks Using FPGA Partial Reconfiguration," in *Proc. of the Design, Automation and Test in Europe (DATE) Conference*, 2013, p. 721 to 724.

[9] G. Carvajal, M. Figueroa, R. Trausmuth, and S. Fischmeister, "Atacama: An Open FPGA-Based Platform for Mixed-Criticality Communication in Multi-segmented Ethernet Networks," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, p. 121 to 128.

[10] U. Khan, H. L. Owen, and J. Hughes, "FPGA architectures for ASIC hardware emulators," in *Proc. of the ASIC Conference and Exhibit*, 1993, p. 336 to 340.

[11] L. Jianhua, Z. Ming, B. Jinian, and X. Hongxi, "A debug sub-system for embedded-system co-verification," in *Proc. of the Conference on ASIC*, 2001, p. 777 to 780.

[12] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: overcoming pin limitations in FPGA-based logic emulators," in *Proc. of the Workshop on FPGAs for Custom Computing Machines (FCCM)*, 1993, p. 142 to 151.

[13] S. Saponara, N. L'Insalata, T. Bacchillone, E. Petri, I. Del Corona, and L. Fanucci, "Hardware/Software FPGA-based Network Emulator for High-speed On-board Communications," in *Proc. of the Conference on Digital System Design Architectures, Methods and Tools (DSD)*, 2008, p. 353 to 359.

[14] J. Sobotka and J. Novak, "FlexRay controller with special testing capabilities," in *in Proc. of the Conference on Applied Electronics (AE)*, 2012, p. 269 to 272.

[15] T. Fernandez-Carames, M. Gonzalez-Lopez, and L. Castedo, "FPGA-based vehicular channel emulator for evaluation of IEEE 802.11p transceivers," in *Proc. of the Conference on Intelligent Transport Systems Telecommunications,(ITST)*, 2009, p. 592 to 597.

[16] *FlexRay Communications System, Protocol Specification Version 2.1 Revision A*, FlexRay Consortium Std., December 2005.